On the Composition of Secure Multi-Party Protocols

Thesis for the Ph.D. Degree

by

Yehuda Lindell

Under the Supervision of
Professors Oded Goldreich and Moni Naor

Department of Computer Science and Applied Mathematics
The Weizmann Institute of Science

Submitted to the Feinberg Graduate School of
the Weizmann Institute of Science
Rehovot 76100, Israel

July 16, 2002

Abstract

In the setting of multi-party computation, sets of two or more parties with private inputs wish to jointly compute some (predetermined) function of their inputs. This computation should be such that the outputs received by the parties are correctly distributed, and none of the parties learn anything beyond their prescribed output. This encompasses any distributed computing task and includes computations as simple as coin-tossing and broadcast, and as complex as electronic voting, electronic auctions, electronic cash schemes and anonymous transactions. The feasibility (and infeasibility) of multi-party computation has been extensively studied, resulting in a seemingly comprehensive understanding of what can and cannot be securely computed, and under what assumptions. However, most of this research relates only to the *stand-alone* setting, in which a single set of parties execute a single protocol in isolation. In contrast, in modern network settings, it is usually the case that many parties run many protocol executions. In this thesis, we study the feasibility of secure multi-party computation under this more realistic setting of general *composition* of protocols and executions. The main results presented are as follows:

1. We show that the basic task of achieving secure broadcast is strictly harder under composition. Specifically, in the stand-alone model, it is known that digital signatures can be used in order to obtain broadcast that is secure *for any number of corrupted parties.* In contrast, we show that it is *impossible* to obtain a secure broadcast protocol that composes in parallel or concurrently (even assuming digital signatures), unless one assumes that strictly less than one third of the parties are corrupted.

2. The above impossibility result has a serious impact on the composition of secure multi-party protocols (when a third or more of the parties may be corrupted). This is because all known protocols for secure multi-party computation assume a broadcast channel, and implement this channel in a point-to-point network using a protocol for secure broadcast. Therefore, these multi-party protocols do not compose. We bypass this problem as follows. First, we provide a new definition of secure multi-party computation that is a mild relaxation of previous definitions. Next, we show that under this definition, secure computation of any function can be achieved *without broadcast* by adapting known protocols. As a corollary we obtain secure multi-party protocols that compose (concurrently, when an honest majority is assumed, and in parallel when any number of parties may be corrupted).

3. Finally, we study the feasibility of obtaining "universally composable" secure multi-party computation. The composition operation considered in the framework of universal composability is very general. In particular, it guarantees that a secure protocol remains secure even when run concurrently with other arbitrary protocols. This captures the security requirements of protocols in real settings (like that of the Internet). Previously, it has been shown that when a majority of the parties are assumed to be honest, universally composable protocols can be constructed for any functionality (in the standard model). It has also been

shown that when only a minority of the parties are honest, there are important functionalities that *cannot* be securely realized in a universally composable way in the standard model. Thus, an important open question is whether or not there exists a reasonable model in which universally composable multi-party computation of general functionalities can be achieved, for an honest minority. We show that in the common reference string model, universally composable protocols exist for any two-party and multi-party functionality and for any number of corrupted parties. This result establishes the feasibility of obtaining secure two-party and multi-party computation (without an honest majority) under the most stringent security definition currently known.

Acknowledgements

First and foremost, I would like to thank my thesis advisors Oded Goldreich and Moni Naor. I am most grateful to Moni for helping me find my way in my first year. Although I spent most of this year studying (rather than researching), Moni continually threw research questions at me from many different fields of theoretical computer science. Eventually I found my way to Cryptography, and by the time Moni went away for sabbatical I already had a large stack of problems to look at.

My work with Oded began in the second year of my studies and his contribution to me on both a personal and research level has been enormous. It was always a pleasure to work with Oded; his demand for excellence on the one hand and positive encouragement on the other served as great motivating factors throughout the ups and downs of the last few years. Oded has served as a role model for me and I am sure that he will continue to be a source of inspiration. I have learnt many things from Oded; both technically and also regarding ones attitude and approach to research. I hope that I will have many opportunities to continue working with Oded in the future.

I feel greatly privileged to have been given the opportunity to carry out my Ph.D. research at the Weizmann Institute. I am convinced that the atmosphere of the Computer Science department, generated by both the faculty and students, significantly contributed to the success of my studies. My time at Weizmann has been most enjoyable, and I believe this to be a necessary condition for successful research. Throughout my graduate studies, I learnt much from the faculty members of the department and from fellow students. I would like to especially mention Shafi Goldwasser, with whom I was very fortunate to work with. My work together with Shafi was both enjoyable and enlightening. I would also like to thank my office-mate Alon Rosen for the many discussions that we had, ranging from the "philosophy of science" to technical issues in our research. Although I have no joint paper with Alon, I feel that he has been one of my closest research collaborators. I hope that in the future we will have the opportunity to work together. Special thanks also to Boaz Barak for fruitful joint work and many helpful discussions.

Many thanks to my other co-authors: Ran Canetti, Anna Lysyanskaya, Benny Pinkas and Tal Rabin. I enjoyed working with them all and learnt much from each of them. I would like to especially thank Ran, who was always quick and thorough in answering my questions. Ran was of great help to me even before I met him and our ensuing joint work (although being torturous to write) was very enjoyable. It was also a great pleasure to work together with Tal. Beyond our joint work together, I feel much gratitude to Tal from my Summer Internship at the cryptography group at IBM T.J.Watson in 2001. Beyond being a surprisingly productive summer, the atmosphere in the group was fun and social. Table tennis (or, ping-pong) after lunch and working in the park turned out to actually improve productivity (which is of course the only reason we did it). Tal was also most helpful with all the technical side of moving and settling in. Many thanks! Other people that I worked with and learnt from include Marc Fischlin, Rosario Gennaro, Shai Halevi, Nick Howgrave-Graham, Jonathan Katz, Eyal Kushilevitz, Michael Langberg, Daniele Micciancio Kobbi Nissim, Omer Reingold and Salil Vadhan. Thanks to you all.

Finally, I wish to acknowledge my co-authors on the results that make up this thesis. Chapter 2 is due to joint work with Anna Lysyanskaya and Tal Rabin [66], Chapter 3 is joint work with Shafi Goldwasser [54], and Chapter 4 is joint work with Ran Canetti, Rafi Ostrovsky and Amit Sahai [20].

Last and not least, I would like to thank my beautiful wife Yael for all her support. The pressures of a Ph.D. are sometimes taxing, and Yael's understanding throughout was always felt and appreciated.

Contents

Chapter 1

Introduction

1.1 Secure Computation

In a secure multi-party computation, a set of n parties with private inputs wish to jointly and securely perform a computation based on their individual inputs. This computation should be such that each party receives its correct output (*correctness*), and none of the parties learn anything beyond their prescribed output (*privacy*). For example, in an election protocol, correctness ensures that no coalition of parties can influence the outcome of the election beyond just voting for their preferred candidate, whereas privacy ensures that no parties learn anything about the individual votes of other parties. Secure multi-party computation can be viewed as the task of carrying out a distributed computation, while protecting honest parties from the malicious intents of dishonest (or corrupted) parties. Specifically, a protocol execution involves the n participating parties and an adversary who has control over a subset of t of these parties. The adversary is therefore an active, inside attacker who attempts to somehow "harm" the honest participants (e.g., in an election, the adversary may attempt to sway the outcome or learn the honest parties' private votes).

Security in multi-party computation. A number of different definitions have been proposed for secure multi-party computation. These definitions aim to ensure a number of important security properties. The most central of these are:

- *Privacy:* No party should learn anything more than its prescribed output. In particular, the only information that should be learned about other parties' inputs is what can be derived from the output itself.
- *Correctness:* Each party is guaranteed that the output that it receives is correct.
- *Independence of Inputs:* Corrupted parties must choose their inputs independently of the honest parties' inputs.
- *Guaranteed Output Delivery:* Corrupted parties should not be able to prevent honest parties from receiving their output. In other words, the adversary should not be able to carry out a denial of service attack.
- *Fairness:* Corrupted parties should receive their outputs if and only if the honest parties also receive their outputs. The scenario where a corrupted party obtains output and an honest party does not should not be allowed to occur.

The standard definition today (cf. [13] following [53, 4, 69]) formalizes the above requirements (and others) in the following general way. Consider an ideal world in which an external trusted party is willing to help the parties carry out their computation. An ideal computation takes place in the ideal world by having the parties simply send their inputs to the trusted party who computes the desired function and passes each party its prescribed output. Notice that all of the above security properties (and more) are ensured in this ideal computation. The security of a real protocol is established by comparing the outcome of the protocol to the outcome of an ideal computation. Specifically, a real protocol that is run by the parties (in a world where no trusted party exists) is said to be secure, if no adversary controlling a coalition of corrupted parties can do more harm in a real execution that in the above ideal computation. This is stated more formally as follows: for any adversary attacking a real protocol execution there exists an adversary attacking an ideal execution such that the input/output distributions of the parties in the real and ideal executions are essentially the same. Since the adversary is unable to cause any harm in an ideal execution, this means that security is also guaranteed in a real protocol execution.

We remark that the above informal description is "overly ideal" in the following sense. It is a known fact that unless an honest majority is assumed, it is impossible to obtain general protocols for secure multi-party computation that guarantee output delivery and fairness (e.g. [58, 23]). These requirements are therefore relaxed when no honest majority is assumed. In particular, under certain circumstances, honest parties may not receive their prescribed output, and fairness is not always guaranteed. In fact, a number of different levels of fairness have been considered. On one extreme, *complete fairness* states that corrupted parties receive their outputs if and only if honest parties receive their outputs (this is the notion of fairness described above). On the other extreme, *no fairness* states that corrupted parties may receive their outputs even if honest parties do not. An intermediate notion called *partial fairness* has the following property. There exists a specified party (say P_1) such that if P_1 is honest then complete fairness is achieved. However, if P_1 is corrupt then no fairness is achieved. That is, sometimes fairness is obtained and sometimes it is not (and the fairness is thus partial).

Feasibility of secure multi-party computation. Wide reaching feasibility results regarding secure multi-party computation were presented in the mid to late 1980's. The most central of these are as follows (recall that t equals the number of corrupted parties):

1. For $t < n/3$, secure multi-party protocols (with complete fairness and guaranteed output delivery) can be achieved for any function in a point-to-point network and without any setup assumptions. This can be achieved both in the computational setting [52] (assuming the existence of trapdoor permutations), and in the information theoretic (private channel) setting [10, 22].[1]

2. For $t < n/2$, secure multi-party protocols (with complete fairness and guaranteed output delivery) can be achieved for any function assuming that the parties have access to a broadcast

[1]In the information theoretic setting of secure computation, the adversary is not bound to any complexity class (and in particular, is not assumed to be polynomial-time). Results in this model require no complexity or cryptographic assumptions and hold unconditionally. The only assumption used is that parties are connected via ideally private channels (i.e., it is assumed that the adversary cannot eavesdrop on the communication between honest parties).

In contrast, in the computational setting the adversary is assumed to be a probabilistic polynomial-time machine (or a polynomial-size family of circuits). Results in this model typically assume cryptographic assumptions like the existence of trapdoor permutations. We note that in this model, it is not necessary to assume that the parties have access to ideally private channels (because such channels can be implemented using public-key encryption).

channel. This can be achieved in the computational setting [52] (with the same assumptions as above), and in the information theoretic setting [76].

3. For $t \geq n/2$, secure multi-party protocols with partial fairness (and without guaranteeing output delivery) can be achieved assuming that the parties have access to a broadcast channel and in addition assuming the existence of trapdoor permutations [81, 52, 44]. Some works attempting to provide higher levels of fairness have also appeared [81, 42, 53, 8].

We stress that all of the above results have been proven in the *stand-alone* model of computation only. However, the fact that a protocol is secure in the stand-alone model does *not* (necessarily) mean that it remains secure when run in a multi-execution setting. This leads us to ask the following fundamental question: Do analogous feasibility results hold in more general multi-execution settings (i.e., under composition)?

1.2 Protocol Composition

In general, the notion of "protocol composition" refers to a setting where the participating parties are involved in many protocol executions. Furthermore, the honest parties participate in each execution as if it is running in isolation (and therefore obliviously of the other executions taking place). In particular, this means that honest parties are not required to coordinate between different executions or remember the history of past executions. Requiring parties to coordinate between executions is highly undesirable and sometimes may even be impossible (e.g., it is hard to imagine successful coordination between protocols that are designed independently of each other). We note that in contrast to the honest parties, we assume that the adversary may coordinate its actions between the protocol executions. This asymmetry is due to the fact that some level of coordination is clearly possible. Thus, although it is undesirable to rely on it in the construction of protocols, it would be careless to assume that the adversary cannot utilize it to some extent.

1.2.1 Types of Protocol Composition

As we have described, the notion of protocol composition relates to settings in which many protocol executions take place. However, there are actually many ways to interpret this notion. We describe two important interpretations here:

1. Self-composition: In the stand-alone model, a single set of parties carry out a single protocol execution. Furthermore, these parties are assumed to be the only parties in the network. In contrast, a protocol is said to *self-compose* (or compose with respect to itself) if it remains secure when a single set of parties carry out *many* protocol executions. We stress that the honest parties run each protocol execution obliviously of the other executions. In summary, in self-composition, the same set of parties run many independent copies of the same protocol (and it is assumed that they are the only parties in the network).[2]

2. General-composition: In this type of composition, many sets of parties carry out many executions of arbitrary protocols. Thus, general-composition differs from self-composition in the following two ways. First, different sets of (possibly overlapping) parties interact in the

[2]Another type of self-composition considers a scenario where many sets of parties carry out executions of the same protocol. The model studied in work on non-malleability [32] falls under this category. In particular, they consider two pairs of parties running the same protocol concurrently and an adversary who controls the 2nd party in the 1st execution and the 1st party in the 2nd execution.

network. Second, the parties run arbitrary protocols that may be designed independently of each other. A protocol is said to maintain security under *general-composition* if its security (as defined by comparing it to an ideal execution) is maintained even when it is run in such a scenario.

In both of the above notions of composition, many protocol executions take place. However the *scheduling* of these executions is also an important factor when considering composition. We now describe the three main types of scheduling that appear in the literature:

1. Sequential: a new execution begins strictly after the previous one terminates.
2. Parallel: all executions begin at the same time and proceed at the same rate (i.e., in a synchronous fashion).
3. Concurrent: the scheduling of the protocol executions, including when they start and the rate at which they proceed, is maliciously determined by the adversary. That is, the adversary has full control over when messages sent by the parties are delivered (as is typical in an asynchronous network).

Another type of scheduling, called concurrent with timing [33], lies somewhere between parallel and concurrent scheduling. Here, the adversary can determine when protocols begin but is limited in its ability to vary the rate at which the protocols proceed. (In particular, the adversary cannot delay the delivery of messages for an arbitrary long period of time.)

1.2.2 Feasibility of Protocol Composition

Most of the research on protocol composition considers the question of obtaining security under self-composition. Much of this research has been focused on zero-knowledge. Specifically, it has been shown that sequential composition of (auxiliary input) zero-knowledge always holds [51], whereas parallel composition does not necessarily hold [48]. In contrast, zero-knowledge protocols that do compose in parallel have been demonstrated [47, 46]. The question of zero-knowledge that composes *concurrently* was initiated by [33] who considered the above-mentioned timing model. The feasibility of concurrent zero-knowledge with arbitrary scheduling was later demonstrated by [77]. In general, the issue of concurrent zero-knowledge has received much attention, culminating in a rather exact characterization of the round complexity of black-box concurrent zero-knowledge [61, 79, 17, 60, 74] (work on non black-box concurrent zero-knowledge appears in [1]). Another specific problem that has been studied in the context of concurrent self-composition is that of oblivious transfer [43]. We stress that all of the above work deals with self-composition only.

In addition to the above work on specific protocol problems, research has been carried out regarding the composition of protocols for (general) secure multi-party computation. This research is typically more definitional in nature. That is, papers present a definition, show that protocols compose under this definition, and then (hopefully) proceed to present protocols that achieve the definition.[3] The sequential composition of secure multi-party protocols was considered both regarding *self*-composition [5], and *general*-composition [69, 13]. It seems safe to say that we have a good understanding of the sequential composition of protocols. For example, it is known that any multi-party functionality can be securely computed under sequential general-composition, and for any number of corrupted parties [52, 13].

[3]This is in contrast to the work on the composition of zero-knowledge, for example, where the definition is generally accepted and the challenge is in finding protocols that meet the definition.

Recently, a robust notion of security, called universal composability, was put forward in [14]. This definition follows the standard paradigm of comparing a real protocol execution to an ideal model involving a trusted party. However, it also differs in a very important way. The traditional model considered for secure computation includes the parties running the protocol, plus an adversary $\mathcal{A}$ that controls a set of corrupted parties. However, in the universally composable framework, an additional adversarial entity called the environment $\mathcal{Z}$ is introduced. This environment generates the inputs to all parties, reads all outputs, and in addition interacts with the adversary in an arbitrary way throughout the computation. (As is hinted by its name, $\mathcal{Z}$ represents the external environment that consists of arbitrary protocol executions that may be running concurrently with the given protocol.) A protocol is said to securely realize a given functionality f if for any real-model adversary $\mathcal{A}$ that interacts with the protocol, there exists an ideal-model adversary $\mathcal{A}'$, such that *no environment* $\mathcal{Z}$ can tell whether it is interacting with $\mathcal{A}$ and parties running the protocol, or with $\mathcal{A}'$ and parties that are running in the ideal model for f. Such a protocol is called universally composable. (In a sense, $\mathcal{Z}$ serves as an "interactive distinguisher" between a run of the protocol and an ideal execution involving a trusted party.) The importance of this new definition is due to a "composition theorem" that states that any universally composable protocol remains secure under *concurrent general-composition* [14]. An important advantage of universally composable security is that it enables the modular design of protocols. In particular, it is not necessary to design an entire "secure system". Rather, protocols are designed independently and once proven secure (under this definition), can be safely run in any system.

It has been shown that in the case of an honest majority, universally composable protocols exist for any multi-party functionality [14] (building on [10, 76]). However, prior to our work, very little was known of the honest minority and two-party cases. Additional work on parallel and concurrent general-composition in more limited settings appears in [30] who deal specifically with the information theoretic setting and assume ideally private channels, and in [73] who relate to a setting where a single protocol execution takes place concurrently to another arbitrary protocol (that is, the "amount" of concurrency is strictly limited).

We conclude that the question of general and self composition of multi-party protocols when an honest majority is not assumed was largely unanswered. In particular, it was not known how to achieve concurrent general-composition, or even concurrent self-composition. The parallel case has also received almost no attention. Specifically, nothing was known regarding parallel general-composition and there is no protocol for which security under parallel self-composition has been proved.[4] In this thesis, we solve some of these questions.

1.3 Our Results

This thesis studies the feasibility of protocols under different types of composition. We present lower bounds regarding the fault tolerance of broadcast protocols that remain secure under parallel self-composition. This result has a significant impact on the composability of secure multi-party protocols as they typically rely extensively on the use of a broadcast primitive. Nevertheless, we show that the lower bound can be bypassed in the context of secure multi-party computation such that concurrent and parallel self-composition can be achieved. Finally, we present universally composable protocols for securely realizing any multi-party functionality in the common reference

[4]We conjecture that the protocol of [52] self composes in parallel when the zero-knowledge proofs and proofs of knowledge used are such that simulation and knowledge extraction can be executed under parallel composition. We have not verified this conjecture.

string model.[5] (Recall that such protocols achieve *concurrent general-composition.*) Each chapter of this thesis is preceded by a detailed introduction. Here, we briefly state the results and explain how they relate to each other.

1.3.1 The Composition of Authenticated Byzantine Agreement

We begin by studying the specific and basic problem of achieving secure broadcast in a point-to-point network. This problem is known as Byzantine Agreement (or Generals) and has been the focus of much research. Pease et al. [71, 62] showed that Byzantine Agreement can be achieved in the standard model *if and only if* strictly less than 1/3 of the parties may be corrupted. They further showed that by augmenting the network with a public-key infrastructure for digital signatures, it is possible to obtain secure protocols for any number of corrupted parties. This augmented problem is called "authenticated Byzantine Agreement".

We ask whether protocols for authenticated Byzantine Agreement self-compose in parallel or concurrently. This question is answered in the negative. In fact, we prove an impossibility result stating that authenticated Byzantine Agreement *cannot be self-composed in parallel* (even twice), if 1/3 or more of the parties are corrupted. In contrast, it is easily shown that any protocol for Byzantine Agreement in the standard model self-composes concurrently. Thus, on the one hand, if less than 1/3 of the parties are corrupted, protocols for Byzantine Agreement in the standard (i.e., "non-authenticated") model can be used, and composition is achieved. On the other hand, if 1/3 or more of the parties are corrupted (and parallel or concurrent composition is required), then even authenticated Byzantine Agreement cannot be used. We conclude that the augmentation of the standard model with a public-key infrastructure is of no benefit when parallel or concurrent composition is needed.

Conceptually, this result demonstrates that achieving security under composition (even a relatively weak type of composition such as two execution parallel self-composition) can be strictly harder than achieving security in the stand-alone model. It also shows that any protocol that achieves composition will have to necessarily do so without the help of authenticated Byzantine Agreement. This conclusion is bothersome because all known protocols for secure multi-party computation in the case that a third or more of the parties may be corrupted (e.g., [52, 76, 9]) rely extensively on broadcast, and indeed implement it using authenticated Byzantine Agreement. Thus, none of these protocols compose in the point-to-point model.

1.3.2 Secure Computation Without Agreement

As we have mentioned, an immediate concern that arises out of the previous result relates to the feasibility of obtaining composition of secure multi-party protocols in a point-to-point network. This concern motivates the following question: Is it possible to obtain secure multi-party computation without relying on a broadcast channel (and thus without using authenticated Byzantine Agreement) when a third or more of the parties are corrupted?

We answer this question in the affirmative. Specifically, we mildly relax the definition of secure computation allowing abort (i.e., when output delivery is not guaranteed), and show how this definition can be achieved without the use of a broadcast channel, and for any number of corrupted parties. The novelty of our definition is in decoupling the issue of *agreement* from the central security issues of privacy and correctness in secure computation. The only difference between our definition and previous ones is as follows. Previously, it was required that if one honest party

[5]In the common reference string model, all parties are given a common, public reference string that is ideally chosen from a given distribution.

aborted then all other honest parties also abort. Thus, the parties *agree* on whether or not the protocol execution terminated successfully or not. In our new definition, it is possible that some honest parties abort while others receive output. Thus, there is no agreement regarding the success of the protocol execution. We stress that in all other aspects, our definition remains the same (in particular, the properties of privacy, correctness and independence of inputs are all preserved). An important corollary of this result is the existence of multi-party protocols that self-compose for the case that a third or more of the parties may be corrupted.

1.3.3 Universally Composable Multi-Party Computation

The broadest type of composition described above is that of concurrent general-composition. As we have mentioned, universally composable protocols remain secure under this type of composition. The feasibility of obtaining universally composable secure multi-party computation is therefore of great interest. The following results are known regarding universal composability in the standard model (where no setup phase is assumed). On the one hand, it has been shown that in the case of an honest majority, universally composable protocols exist for any multi-party functionality [14]. On the other hand, it has also been shown that universally composable protocols for the basic two-party functionalities of commitment, zero-knowledge and coin-tossing *cannot* be obtained [16, 14]. These impossibility results can actually be extended to include a very large class of natural functionalities [19]. (We note that two-party computation is a special case of multi-party computation when there is no honest majority, and thus these results also rule out the possibility of obtaining universally composable multi-party computation for an honest minority.) Thus, a natural and important question to ask is whether or not there exists a reasonable model in which it is possible to obtain general universally composable secure computation for the *two-party* and *honest-minority multi-party* cases?

Universally composable protocols for the *specific* two-party functionalities of commitment and zero-knowledge have been constructed in the common reference string model [16, 25]. In this model, all parties are given access to some string that is ideally chosen from a given distribution. The string must be chosen in a trusted manner, and therefore this setup assumption is definitely undesirable. However, due to the impossibility results of the standard model mentioned above, some setup assumption is required, and that of a common reference string seems rather minimal. We note that there are some scenarios where the use of a common reference string is very reasonable. Take for example the case of a large organization that wishes to have its employees use secure protocols for a number of different tasks. Then, in this context, the organization itself is trusted to properly choose a common reference string.

We ask whether or not there exist universally composable protocols for general functionalities in the common reference string model, in the two-party case, and in the multi-party case when no honest majority is assumed. We answer this question in the affirmative and prove the following feasibility result:

> *There exist universally composable protocols in the common reference string model for securely computing any multi-party functionality and that tolerate any number of corrupted parties.*

An important corollary of this result is the feasibility of obtaining universally composable *two-party* computation for any functionality. We stress that previous feasibility results for secure computation (in the case of an honest minority) related to the stand-alone model only. Thus, our result is the first general construction to guarantee security under *concurrent general-composition,* without assuming an honest majority.

Remark: Another important feature of our construction relates to the specific adversarial model used. We differentiate between two types of adversaries: static and adaptive. A static adversary controls a set of corrupted parties that is fixed at the onset of the computation (but whose identities are of course unknown to the honest parties). In contrast, an adaptive adversary actively corrupts parties throughout the computation. We stress that an adaptive adversary has full power to choose who to corrupt and when, and can base this choice on its view of the computation so far. We present universally composable protocols for both the static and adaptive adversarial models. Our protocols are actually the first general constructions to guarantee security against adaptive adversaries in the case of an honest minority. (All previous protocols for the adaptive adversarial model [15, 6, 24], even in the stand-alone model, assumed an honest majority.)

1.3.4 An Apparent Contradiction

We conclude with a remark regarding the connection between the results described in Sections 1.3.1 and 1.3.3. On the one hand, in Section 1.3.1 we claim that it is impossible to construct a protocol for the broadcast functionality (i.e., Byzantine Agreement) that self-composes in parallel, when 1/3 or more of the parties are corrupted. (This impossibility result holds also in the common reference string model, see Section 2.2.2.) On the other hand, in Section 1.3.3 we claim that for any functionality (and, in particular, for the broadcast functionality), there exists a protocol in the common reference string model that remains secure under concurrent general-composition and for *any* number of corrupted parties. Thus, it would seem that these claims contradict each other.

However, there is actually no contradiction because the results are in different models. Specifically, the impossibility result of Section 1.3.1 is demonstrated in a model where guaranteed output delivery is required (i.e., all honest parties must receive their output and the adversary cannot carry out a denial of service attack). In contrast, the definition of universal composability, similarly to the definitions of Section 1.3.2, does not guarantee output delivery and thus honest parties may abort (the outline of the definition of universal composability in Section 1.2.2 is very brief and therefore does not include this point). We stress that the proof of the impossibility result for composing (authenticated) Byzantine Agreement relies heavily on the requirement that parties *must* receive output; whereas the fact that output delivery is *not* required is essential for achieving universal composability in the case of an honest minority. Thus, this difference in modeling is actually central to both of the results. See the formal definitions of Byzantine Agreement and universal composability in Sections 2.2 and 4.3, respectively, for a full description of the model used in each result.[6]

1.4 Organization

Although the results of this thesis are all on the subject of the composition of secure protocols, each result can be viewed in its own right. Therefore, despite the risk of a small amount of repetition, each chapter is presented in a self-contained manner, enabling a reader to focus on one chapter without having to read the others. We also provide an appendix containing abstracts of other work done by the author during graduate studies at the Weizmann Institute of Science.

[6]There is another technical difference between the models in that the current formalization of universal composability assumes that each execution is assigned a unique session identifier. In such a case, it is actually possible to achieve authenticated Byzantine Agreement that self-composes concurrently, for any number of corrupted parties; see Section 2.4. Nevertheless, we believe that the feasibility result for universal composability holds even when unique session identifiers are not assumed.

Chapter 2

The Composition of Authenticated Byzantine Agreement

In this chapter we show that the basic problem of achieving secure broadcast in a distributed network is strictly harder when composition is required. In particular, assuming a public-key infrastructure for digital signatures, it is known that secure broadcast can be achieved for *any* number of corrupted parties in the stand-alone model. In this chapter, we show that an analogous result that composes cannot be achieved. That is, when a third or more of the parties may be corrupted, it is *impossible* to obtain secure broadcast that remains secure under parallel self-composition, (even given a public-key infrastructure). Notice that our impossibility result holds even for the relatively weak notion of parallel self-composition.

2.1 Introduction

The Byzantine Generals (Byzantine Agreement[1]) problem is one of the most researched areas in distributed computing. Numerous variations of the problem have been considered under different communication models, and both positive results (i.e., protocols) and negative results (i.e., lower bounds on efficiency and fault tolerance) have been established. The reason for this vast interest is the fact that the Byzantine Generals problem is the algorithmic implementation of a broadcast channel within a point-to-point network. In addition to its importance as a primitive in its own right, broadcast is a key tool in the design of secure protocols for multi-party computation.

The problem of Byzantine Generals is (informally) defined as follows. The setting is that of n parties connected via a point-to-point network, where one party is designated as the General (or dealer) who holds an input message x. In addition, there is an adversary who controls up to t of the parties and can arbitrarily deviate from the designated protocol specification. The aim of the protocol is to securely simulate a broadcast channel. Thus, first and foremost, all (honest) parties must receive the same message. Furthermore, if the General is honest, then the message received by the honest parties must be x (i.e., the adversary is unable to prevent an honest General from successfully broadcasting its given input message).

Pease et al. [71, 62] provided a solution to the Byzantine Generals problem in the standard model, i.e. the information-theoretic model with point-to-point communication lines (and no setup assumptions). For their solution, the number of corrupted parties, t, must be less than $n/3$. Furthermore, they complemented this result by showing that the requirement for $t < n/3$ is in fact

[1]These two problems are essentially equivalent.

inherent. That is, no protocol which solves the Byzantine Generals problem in the standard model can tolerate a third or more corrupted parties.

The above bound on the number of corrupted parties in the standard model is a severe limitation. It is therefore of great importance to find a different (and realistic) model in which it is possible to achieve higher fault tolerance. One possibility involves augmenting the standard model such that messages sent can be authenticated. By authentication, we mean the ability to ascertain that a message was in fact sent by a specific party, even when not directly received from that party. This can be achieved using a trusted preprocessing phase in which a public-key infrastructure for digital signatures (e.g. [78, 56]) is set up. (We note that this requires that the adversary be computationally bounded. However, there exist preprocessing phases which do not require any computational assumptions; see [72].) Indeed, Pease et al. [71, 62] use such an augmentation and obtain a protocol for the Byzantine Generals problem which can tolerate *any* number of corrupted parties (this is very dramatic considering the limitation to 1/3 corrupted in the standard model). The Byzantine Generals problem in this model is called *authenticated* Byzantine Generals. We often informally refer to this model as the "authenticated model".

A common use of Byzantine Generals is to substitute a broadcast channel in multi-party protocols. As such, it is likely to be executed many times. The question of whether these protocols remain secure when executed concurrently, in parallel or sequentially is thus an important one. However, existing work on this problem (in both the standard and authenticated models) focused on the security and correctness of protocols in the stand-alone model only.

It is not difficult to show that the unauthenticated protocol of Pease et al. [71], and in fact all protocols in the standard model, do compose concurrently (and hence in parallel and sequentially). However, this is not the case with respect to *authenticated* Byzantine Generals. The first to notice that composition in this model is problematic were Gong et al. [57], who also suggest methods for overcoming the problem. Our work shows that these suggestions and any others are in fact futile because composition in this model is impossible (as long as 1/3 or more of the parties may be corrupted). (We note that by composition, we actually refer to stateless composition; see Section 2.2.3 for a formal discussion.)

Our Results. The main result of this chapter is a proof that authenticated Byzantine Generals protocols, both deterministic and randomized, cannot be self-composed in parallel (and thus concurrently). This is a powerful statement with respect to the value of enhancing the standard model by the addition of authentication. Indeed, it shows that despite popular belief, this enhancement *does not* provide the ability to improve fault tolerance when composition is required. That is, if there is a need for parallel composition, then the number of corrupted players cannot be $n/3$ or more, and hence the authenticated model provides no advantage over the standard model. This result is summarized in the following theorem:

Theorem 2.1.1 *No protocol for authenticated Byzantine Agreement that self-composes in parallel (even twice) can tolerate $n/3$ or more corrupted parties.*

On the Use of Unique Session Identifiers. As will be apparent from the proof of Theorem 2.1.1, the obstacle to achieving agreement in this setting is the fact that honest parties cannot tell in which execution of the protocol a given message was authenticated. This allows the adversary to "borrow" messages from one execution to another, and by that attack the system. In Section 2.4, we show that if we further augment the authenticated model so that unique and common indices are assigned to each execution, then security under many concurrent executions can be achieved for any number of corrupted parties.

Thus, on the one hand, our results strengthen the common belief that session identifiers are necessary for achieving authenticated Byzantine Generals. On the other hand, we show that such identifiers *cannot be generated within the system.* Typical suggestions for generating session identifiers in practice include having the General choose one, or having the parties exchange random strings and then set the identifier to be the concatenation of all these strings. However, Theorem 2.1.1 rules out all such solutions. Rather, one must assume the existence of some *trusted external means* for coming up with unique and common indices. This seems to be a very difficult, if not impossible, assumption to realize.

A natural question to ask here relates to the fact that unique and common session identifiers seem to be necessary for carrying out any form of concurrent composition. This is because parties need to be able to allocate messages to protocol executions, and this requires a way of distinguishing executions from each other. Indeed, a natural solution to this problem is to provide unique and global session identifiers to every execution. However, in such a case, these identifiers can also be used for achieving broadcast. Nevertheless, it is possible to correctly allocate messages to protocol executions *without* the use of of global identifiers. This can be achieved as follows. When a party P_i begins a new execution, it chooses a unique, *local* identifier $sid(P_i)$ and sends it to all the other parties. Then, whenever a party P_j wishes to send a message to P_i in this execution, it concatenates the identifier $sid(P_i)$ to the message. In this way, P_i is able to route all messages that it receives to the correct executions. In general, it is guaranteed that all the honest parties in a given execution successfully differentiate this execution from others.[2] Most importantly, this is achieved without any global identifiers whatsoever.

Implications for Secure Multi-Party Computation. As we have stated above, one important use for Byzantine Generals protocols is to substitute the broadcast channel in a multi-party protocol. In fact, all known solutions for general multi-party computation assume a broadcast channel. The implicit claim is that this channel can be substituted by a Byzantine Generals protocol without any complications. However, our results show that the use of authenticated Byzantine Generals in such a way prevents the composition of the larger protocol (even if this protocol does compose when using a physical broadcast channel).

Another important implication of our result is due to the fact that any secure protocol for solving general multi-party tasks can be used to solve Byzantine Generals (by the standard definition of secure computation for honest majority). Therefore, none of these protocols can be composed in parallel or concurrently, unless more than 2/3 of the parties are honest or a physical broadcast channel is available. This issue is dealt with in more depth in Chapter 3.

A Comparison to Zero-Knowledge. It is instructive to compare our results to the work of Goldreich and Krawzcyk [48] on zero-knowledge. They show that there exist protocols that are zero-knowledge when executed stand-alone, and yet do not remain zero-knowledge when composed in parallel (even twice). Thus, they show that zero-knowledge does not *necessarily* compose in parallel. Completing this picture, we note that there exist zero-knowledge protocols that do compose in parallel (for example, see [47, 46]). In contrast, we show that it is impossible to obtain *any protocol* for Byzantine Agreement that will compose in parallel (even twice).

[2]There are no guarantees regarding messages sent by corrupted parties. However, corrupted parties can anyway send arbitrary messages, so this makes no difference.

2.2 Definitions

2.2.1 Computational Model

We consider a setting involving n parties, $P_1, \ldots, P_n$, that interact in a *synchronous* point-to-point network. In such a network, each pair of parties is directly connected, and it is assumed that the adversary cannot modify messages sent between honest parties. Each party is formally modeled by an interactive Turing machine with $n-1$ pairs of communication tapes. The communication of the network proceeds in synchronized rounds, where each round consists of a *send* phase followed by a *receive* phase. In the *send* phase of each round, the parties write messages onto their output tapes, and in the *receive* phase, the parties read the contents of their input tapes.

This paper refers to the *authenticated* model, where some type of *trusted preprocessing phase* is assumed. This is modeled by all parties also having an additional setup-tape that is generated during the preprocessing phase. Typically, in such a preprocessing phase, a public-key infrastructure of signature keys is generated. That is, each party receives its own secret signing key, and in addition, public verification keys associated with all other parties. (This enables parties to use the signature scheme to authenticate messages that they receive, and is thus the source of the name "authenticated".) However, we stress that our lower bound holds for all preprocessing phases (even those that cannot be efficiently generated).

In this model, a t-adversary is a party that controls up to $t < n$ of the parties $P_1, \ldots, P_n$ (these parties are said to be corrupted). The adversary receives the corrupted parties' views and determines the messages that they send. These messages need not be according to the protocol execution, but rather can be computed by the adversary as an arbitrary function of its view. In this work, we consider static adversaries for whom the set of corrupted parties is fixed before the execution begins. (By taking a weaker adversary, we strengthen our impossibility result.) Finally, our impossibility results hold for adversaries (and honest parties) whose running time is of any complexity. In fact, our lower bound holds as long as the adversary is allowed to be of the same complexity as the honest parties.

2.2.2 Byzantine Generals/Agreement

The existing literature defines two related problems: Byzantine Generals and Byzantine Agreement. In the first problem, there is one designated party, the General or dealer, who wishes to broadcast its value to all the other parties. In the second problem, each party has an input and the parties wish to agree on a value, with a validity condition that if a majority of honest parties begin with the same value, then they must terminate with that value. These problems are equivalent in the sense that any protocol solving one can be used to construct a protocol solving the other, while tolerating the same number of corrupted parties. We relax the standard requirements on protocols for the above Byzantine problems in that we allow a protocol to fail with probability that is negligible in some security parameter. This relaxation is needed for the case of authenticated Byzantine protocols where signature schemes are used (and can always be forged with some negligible probability). We present the formal definition for the Byzantine Agreement problem only and all references from here on will be with respect to this problem.

Definition 2.2.1 (Byzantine Agreement): *Let $P_1, \ldots, P_n$ be n parties with associated inputs $x_1, \ldots, x_n$ and let $\mathcal{A}$ be an adversary who controls up to t of the parties. Then, a protocol solves the* Byzantine Agreement *problem, tolerating t corruptions, if for any adversary $\mathcal{A}$ the following two properties hold (except with negligible probability):*

1. Agreement: *All honest parties output the same value.*
2. Validity: *If more than $n/2$ honest parties have the same input value x, then all honest parties output x.*

We note that the validity requirement is sometimes stated so that it must hold only when more than $n/3$ honest parties have the same input value.

Authenticated Byzantine Agreement: In the model for authenticated Byzantine Agreement, some trusted preprocessing phase is run before any executions begin. In this phase, a trusted party distributes keys to every participating party. Formally,

Definition 2.2.2 (Authenticated Byzantine Agreement): *A protocol for* authenticated Byzantine Agreement *is a Byzantine Agreement protocol with the following augmentation:*

- *Each party has an additional* setup-tape.
- *Prior to any protocol execution, an ideal (trusted) party chooses a series of strings $s_1, \ldots, s_n$ according to some distribution, and sets party P_i's* setup-tape *to equal s_i (for every $i = 1, \ldots, n$).*

Following the above preprocessing stage, the protocol is run in the standard communication model for Byzantine Generals/Agreement protocols.

As we have mentioned, a natural example of such a preprocessing phase is one where the strings $s_1, \ldots, s_n$ constitute a public-key infrastructure. That is, the trusted party chooses verification and signing key-pairs $(vk_1, sk_1), \ldots, (vk_n, sk_n)$ from a secure signature scheme, and sets the contents of party P_i's tape to equal $s_i = (vk_1, \ldots, vk_{i-1}, sk_i, vk_{i+1}, \ldots, vk_n)$. In other words, all parties are given their own signing key and the verification keys of all the other parties. We note that this preprocessing phase can also be used to setup a *common reference string* to be accessed by all parties (in this case, all the s_i's are set to the desired common reference string).

We remark that the above-defined preprocessing phase is very strong. First, it is assumed that it is run completely by a trusted party. Furthermore, there is no computational bound on the power of the trusted party generating the keys. Nevertheless, our impossibility results hold even for such a preprocessing phase.

2.2.3 Composition of Protocols

This paper deals with the security of authenticated Byzantine Agreement protocols, when the protocol is executed many times (rather than just once). We define the composition of protocols to be *stateless*. This means that the honest parties act upon their view in a single execution only. In particular, this means that the honest parties do not store in memory their views from previous executions or coordinate between different executions occurring at the current time. Furthermore, in stateless composition, there is no unique session identifier that is common to all participating parties. (See the Introduction for a discussion on session identifiers and their role.) We note that although the parties are stateless, the adversary is allowed to maliciously coordinate between executions and record its view from previous executions. Formally, parallel composition is captured by the following process:

Definition 2.2.3 (parallel self-composition): *Let $P_1, \ldots, P_n$ be parties for an authenticated Byzantine Agreement protocol Π and let $I \subseteq [n]$ ($|I| \leq t$) be the set of corrupted parties controlled by an adversary $\mathcal{A}$. Then, the* parallel self-composition *of Π involves the following process:*

- *Run the preprocessing phase associated with* Π *and obtain the strings* $s_1, \ldots, s_n$. *Then, for every* j, *set the* setup-tape *of* P_j *to equal* s_j.
- *Repeat the following process a polynomial number of times in parallel:*
  1. *The adversary* $\mathcal{A}$ *chooses an input vector* $x_1, \ldots, x_n$.
  2. *Fix the input tape of every honest* P_j *to be* x_j *and the random-tape to be a uniformly (and independently) chosen random string.*
  3. *Invoke all parties for an execution of* Π *(using the strings generated in the preprocessing phase above). The execution is such that for* $i \in I$, *the messages sent by party* P_i *are determined by* $\mathcal{A}$ *(who also sees* P_i*'s view). On the other hand, all other parties follow the instructions as defined in* Π.

Protocol Π *is said to* remain secure under parallel self-composition *if each of the parallel executions constitutes a secure Byzantine Agreement protocol tolerating* t *corruptions.*

We stress that the preprocessing phase is executed only once and all executions use the strings distributed in this phase. Furthermore, we note that Definition 2.2.3 implies that all honest parties are oblivious of the other executions that have taken place (or that are taking place in parallel). This is implicit in the fact that in each execution the parties are invoked with no additional state information, beyond the contents of their input, random and setup tapes (i.e., the composition is *stateless*). In contrast, the adversary $\mathcal{A}$ *can* coordinate between the executions, and its view at any given time includes all the messages received in all other executions.[3]

Concurrent composition is defined analogously, with the only difference being that the adversary can determine the scheduling of the executions (i.e., when they begin and at what rate they proceed). Before continuing, we show that any Byzantine Generals (or Agreement) protocol *in the standard model* composes concurrently.

Proposition 2.2.4 *Let* Π *be a protocol that solves the Byzantine Agreement problem in the* standard model *and tolerates* t *corruptions. Then,* Π *remains secure under concurrent self-composition.*[4]

Proof (sketch): We reduce the security of Π under concurrent composition to its security for a single execution. Assume by contradiction that there exists an adversary $\mathcal{A}$ who runs N concurrent executions of Π, such that with non-negligible probability, in one of the executions the outputs of the parties do not meet the requirement on a Byzantine Agreement protocol. Then, we construct an adversary $\mathcal{A}'$ who internally incorporates $\mathcal{A}$ and attacks a single execution of Π. Intuitively, $\mathcal{A}'$ simulates all executions apart from the one in which $\mathcal{A}$ succeeds in its attack. Formally, $\mathcal{A}'$ begins by choosing an index $i \in_R \{1, \ldots, N\}$. Then, for all but the i^{th} execution of the protocol, $\mathcal{A}'$ plays the roles of the honest parties in an interaction with $\mathcal{A}$ (this simulation is internal to $\mathcal{A}'$). However, for the i^{th} execution, $\mathcal{A}'$ externally interacts with the honest parties and passes messages between them and $\mathcal{A}$. The key point in the proof is that the honest parties hold no secret information (and do not coordinate between executions). Therefore, the simulation of the concurrent setting by $\mathcal{A}'$ for $\mathcal{A}$ is *perfect*. Thus, with probability $1/N$, the i^{th} execution is the one in which $\mathcal{A}$ succeeds. However, this means that $\mathcal{A}'$ succeeds in breaking the protocol for a single execution (where $\mathcal{A}'$'s success probability equals $1/N$ times the success probability of $\mathcal{A}$). This contradicts the stand-alone security of Π. ■

[3]The analogous definition for the composition of *unauthenticated* Byzantine Agreement is derived from Definition 2.2.3 by removing the reference to the preprocessing stage and setup-tapes.

[4]It can actually be shown that Π remains secure even under concurrent *general*-composition.

2.3 Impossibility Result

In this section we show that it is impossible to construct an authenticated Byzantine Agreement protocol that self-composes in parallel (or concurrently), and is secure when $n/3$ or more parties are corrupted. This result is analogous to the [71, 62] lower bounds for Byzantine Agreement in the standard model (i.e., without authentication). We stress that our result does not merely show that authenticated Byzantine Agreement protocols do not necessarily compose; rather, we show that one *cannot* construct protocols that will compose. Since there exist protocols for unauthenticated Byzantine Agreement that are resilient for any $t < n/3$ corrupted parties and compose concurrently, this shows that the advantage gained by the preprocessing step in authenticated Byzantine Agreement protocols is lost when composition is required.

Intuition. Let us first provide some intuition into why the added power of the preprocessing step in authenticated Byzantine Agreement does not help when composition is required. (Recall that on the one hand in the stand-alone setting, there exist authenticated Byzantine Agreement protocols that tolerate any number of corrupted parties. On the other hand, under parallel composition, only $t < n/3$ corruptions can be tolerated.) An instructive step is to first see how authenticated Byzantine Agreement protocols typically utilize a public-key infrastructure (set up in the preprocessing step) in order to increase fault tolerance. Consider three parties A, B and C participating in a standard (unauthenticated) Byzantine Agreement protocol. Furthermore, assume that during the execution A claims to B that C sent it some message x. Then, B cannot differentiate between the case that C actually sent x to A, and the case that C did not send this value and A is corrupted. Thus, B cannot be sure that A really received x from C. Indeed, such a model has been called the "oral message" model, in contrast to the "signed message" model of authenticated Byzantine Agreement [62]. The use of signature schemes helps to overcome this exact problem: If C had *signed* on the message x and sent this signature to A, then A could forward the signature to B. Since A cannot forge C's signature, this would then constitute a proof that C had indeed sent x to A. Utilizing the unforgeability property of signatures, it is thus possible to achieve Byzantine Agreement for any number of corrupted parties.

However, the above intuition holds only in a setting where a single execution of the agreement protocol takes place. Specifically, if a number of executions were to take place, then A may send B a value x along with C's signature on x, yet B would still not know whether C signed on x in this execution, or in a different (concurrent or parallel) execution. Thus, the mere fact that A produces C's signature on a value does not provide proof that C signed *this* value in *this* execution. As we will see in the proof, this is enough to render the public-key infrastructure useless under parallel composition.

Theorem 2.3.1 (Theorem 2.1.1 – restated): *No protocol for authenticated Byzantine Agreement that self-composes in parallel (even twice) can tolerate $n/3$ or more corrupted parties.*

Proof: Our proof of Theorem 2.1.1 uses ideas from the proof by Fischer et al. [38] that no unauthenticated Byzantine Agreement protocol can tolerate $n/3$ or more corrupted parties. We begin by proving the following lemma:

Lemma 2.3.2 *There exists no protocol for authenticated Byzantine Agreement for three parties, that composes in parallel (even twice) and can tolerate one corrupted party.*

Proof: Assume, by contradiction, that there exists a protocol Π that solves the Byzantine Agreement problem for three parties A, B and C, where one may be corrupt. Furthermore, Π remains secure even when composed in parallel twice. Exactly as in the proof of Fischer et al. [38], we define a hexagonal system S that intertwines two independent copies of Π. That is, let A_0, B_0, C_0 and A_1, B_1 and C_1 be independent copies of the three parties participating in Π. By independent copies, we mean that A_0 and A_1 are the same party A with the same key tape, that runs in two different parallel executions of Π, as defined in Definition 2.2.3. The system S is defined by connecting party A_0 to C_1 and B_0 (rather than to C_0 and B_0); party B_0 to A_0 and C_0; party C_0 to B_0 and A_1; and so on, as in Figure 2.1.

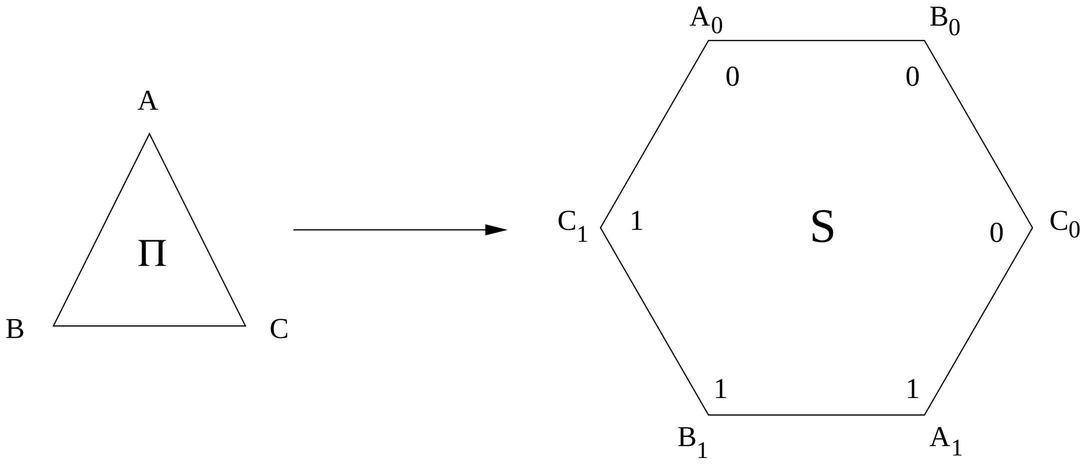

Figure 2.1: Combining two copies of Π in a hexagonal system S.

In the system S, parties A_0, B_0, and C_0 have input 0; while parties A_1, B_1 and C_1 have input 1. Note that within S, all parties follow the instructions of Π exactly. We stress that S is *not* a Byzantine Agreement setting (where the parties are joined in a complete graph on three nodes), and therefore the definitions of Byzantine Agreement tell us nothing directly of what the parties' outputs should be. However, S *is* a well-defined system and this implies that the parties have well-defined output distributions. The proof proceeds by showing that if Π is a correct Byzantine Agreement protocol, then we arrive at a contradiction regarding the output distribution in S. We begin by showing that B_0 and C_0 output 0 in S. We denote by $\mathsf{rounds}(\Pi)$ the upper bound on the number of rounds of Π (when run in a Byzantine Agreement setting).

Claim 2.3.3 *Except with negligible probability, parties B_0 and C_0 halt within* $\mathsf{rounds}(\Pi)$ *steps and output* 0 *in the system S.*

Proof: We prove this claim by showing that there exists an adversary $\mathcal{A}$ controlling A_1 and A_2 who participates in two parallel copies of Π and simulates the system S, with respect to B_0 and C_0's view. (We stress that $\mathcal{A}$ controlling A_1 and A_2 is considered one corruption because these are both copies of the same party.) The adversary $\mathcal{A}$ (and the other honest parties participating in the parallel executions) work within a Byzantine Agreement setting where there are well-defined requirements on their output distribution. Therefore, by analyzing their output in this parallel execution setting, we are able to make claims regarding their output in the system S.

Let A_0, B_0 and C_0 be parties running an execution of Π, denoted Π_0, where B_0 and C_0 both have input 0. Furthermore, let A_1, B_1 and C_1 be running a parallel execution of Π, denoted Π_1, where B_1 and C_1 both have input 1. Recall that B_0 and B_1 are independent copies of the party B with the same key tape (as in Definition 2.2.3); likewise for C_0 and C_1.

Now, let $\mathcal{A}$ be an adversary who controls both A_0 in Π_0 and A_1 in Π_1 (recall that the corrupted party can coordinate between the different executions). Party $\mathcal{A}$'s strategy is to maliciously generate an execution in which B_0's and C_0's view in Π_0 is *identical* to their view in S. $\mathcal{A}$ achieves this by redirecting edges of the two parallel triangles (representing the parallel execution), so that the overall system has the same behavior as S; see Figure 2.2.

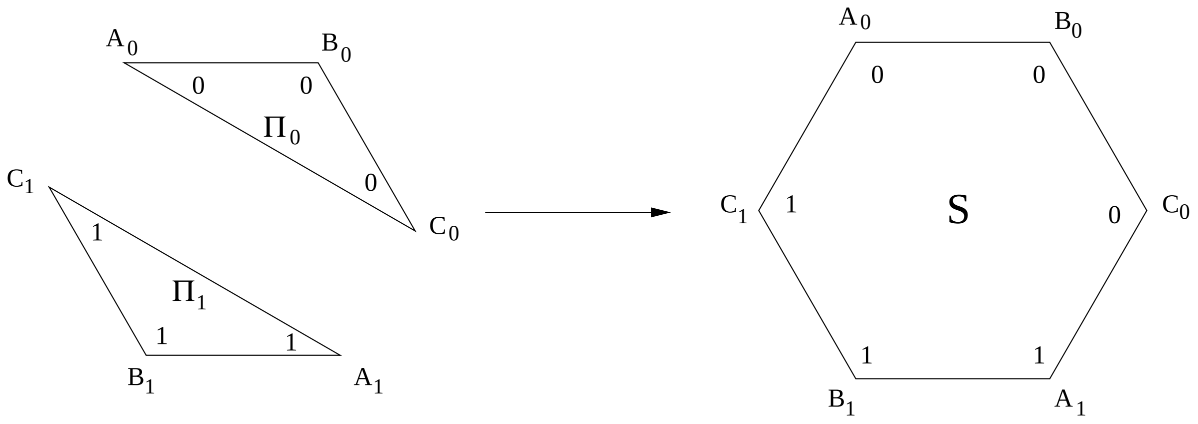

Figure 2.2: Redirecting edges of Π_0 and Π_1 to make a hexagon.

Specifically, the (A_0, C_0) and (A_1, C_1) edges of Π_0 and Π_1 respectively are removed, and the (A_0, C_1) and (A_1, C_0) edges of S are added in their place. $\mathcal{A}$ is able to make such a modification because it only involves redirecting messages to and from parties that it controls (i.e., A_0 and A_1).

Before proceeding, we present the following notation: let $\mathsf{msg}_i(A_0, B_0)$ denote the message sent from A_0 to B_0 in the i^{th} round of the protocol execution. We now formally show how the adversary $\mathcal{A}$ works. $\mathcal{A}$ invokes parties A_0 and A_1, upon inputs 0 and 1 respectively. We stress that A_0 and A_1 follow the instructions of protocol Π exactly. However, $\mathcal{A}$ provides them with their incoming messages and sends their outgoing messages for them. The only malicious behavior of $\mathcal{A}$ is in the redirection of messages to and from A_0 and A_1. A full description of $\mathcal{A}$'s code is as follows (we recommend the reader to refer to Figure 2.2 in order to clarify the following):

1. *Send outgoing messages of round i:* $\mathcal{A}$ obtains messages $\mathsf{msg}_i(A_0, B_0)$ and $\mathsf{msg}_i(A_0, C_0)$ from A_0 in Π_0, and messages $\mathsf{msg}_i(A_1, B_1)$ and $\mathsf{msg}_i(A_1, C_1)$ from A_1 in Π_1 (these are the round i messages sent by A_0 and A_1 to the other parties; as we have mentioned, A_0 and A_1 compute these messages according to the protocol definition and based on their view).
   - In Π_0, $\mathcal{A}$ sends B_0 the message $\mathsf{msg}_i(A_0, B_0)$ and sends C_0 the message $\mathsf{msg}_i(A_1, C_1)$ (and thus the (A_1, C_1) directed edge is replaced by the directed edge (A_1, C_0)).
   - In Π_1, $\mathcal{A}$ sends B_1 the message $\mathsf{msg}_i(A_1, B_1)$ and sends C_1 the message $\mathsf{msg}_i(A_0, C_0)$ (and thus the (A_0, C_0) directed edge is replaced by the directed edge (A_0, C_1)).
2. *Obtain incoming messages from round i:* $\mathcal{A}$ receives messages $\mathsf{msg}_i(B_0, A_0)$ and $\mathsf{msg}_i(C_0, A_0)$ from B_0 and C_0 in round i of Π_0, and messages $\mathsf{msg}_i(B_1, A_1)$ and $\mathsf{msg}_i(C_1, A_1)$ from B_1 and C_1 in round i of Π_1.
   - $\mathcal{A}$ passes A_0 in Π_0 the messages $\mathsf{msg}_i(B_0, A_0)$ and $\mathsf{msg}_i(C_1, A_1)$ (and thus the (C_1, A_1) directed edge is replaced by the directed edge (C_1, A_0)).
   - $\mathcal{A}$ passes A_1 in Π_1 the messages $\mathsf{msg}_i(B_1, A_1)$ and $\mathsf{msg}_i(C_0, A_0)$ (and thus the (C_0, A_0) directed edge is replaced by the directed edge (C_0, A_1)).

We now claim that B_0 and C_0's view in Π_0 is identical to B_0 and C_0's view in S.[5] This holds because in the parallel execution of Π_0 and Π_1, all parties follow the protocol definition (including A_0 and A_1). The same is true in the system S, except that party A_0 is connected to B_0 and C_1 instead of to B_0 and C_0. Likewise, A_1 is connected to B_1 and C_0 instead of to B_1 and C_1. Furthermore, by the definition of $\mathcal{A}$, the messages seen by all parties in the parallel execution of Π_0 and Π_1 are exactly the same as the messages seen by the parties in S (e.g., the messages seen by C_0 in Π_0 are those sent by B_0 and A_1, exactly as in S). Therefore, the views of B_0 and C_0 in the parallel execution maliciously controlled by $\mathcal{A}$, are *identical* to their views in S.[6]

By the assumption that Π is a correct Byzantine Agreement protocol that composes twice in parallel, we have that in Π_0 both B_0 and C_0 halt within $\mathsf{rounds}(\Pi)$ steps and output 0 (except with negligible probability). The fact that they both output 0 is derived from the fact that B_0 and C_0 are an honest majority with the same input value 0. Therefore, they must output 0 in the face of *any* adversarial A_0; in particular this holds with respect to the specific adversary $\mathcal{A}$ described above. Since the views of B_0 and C_0 in S are identical to their views in Π_0, we conclude that in the system S they also halt within $\mathsf{rounds}(\Pi)$ steps and output 0 (except with negligible probability). This completes the proof of the claim. ■

Using analogous arguments, we obtain the following two claims:

Claim 2.3.4 *Except with negligible probability, parties A_1 and B_1 halt within* $\mathsf{rounds}(\Pi)$ *steps and output* 1 *in the system S.*

In order to prove this claim, the adversary is $\mathcal{C}$ who controls C_1 and C_2 and works in a similar way to $\mathcal{A}$ in the proof of Claim 2.3.3 above. (The only difference is regarding the edges that are redirected.)

Claim 2.3.5 *Except with negligible probability, parties A_1 and C_0 halt within* $\mathsf{rounds}(\Pi)$ *steps and output the same value in the system S.*

Similarly, this claim is proven by defining an adversary $\mathcal{B}$ who controls B_1 and B_2 and follows a similar strategy to $\mathcal{A}$ in the proof of Claim 2.3.3 above.

Combining Claims 2.3.3, 2.3.4 and 2.3.5 we obtain a contradiction. This is because, on the one hand C_0 must output 0 in S (Claim 2.3.3), and A_1 must output 1 in S (Claim 2.3.4). On the other hand, by Claim 2.3.5, parties A_1 and C_0 must output the same value. We conclude that there does not exist a 3-party protocol for Byzantine Agreement that tolerates one corruption and composes twice in parallel. This concludes the proof of the lemma. ■

Theorem 2.1.1 is derived from Lemma 2.3.2 in the standard way [71, 62] by showing that if there exists a protocol that is correct for any $n \geq 3$ and $n/3$ corrupted parties, then one can construct a protocol for 3 parties that can tolerate one corrupted party. This is in contradiction to Lemma 2.3.2, and thus Theorem 2.1.1 is implied. ■

The following corollary, referring to concurrent composition, is immediately derived from the fact that parallel composition (where the scheduling of the messages is fixed and synchronized) is merely a special case of concurrent composition (where the adversary controls the scheduling).

[5]In fact, the views of *all* the parties in the parallel execution with $\mathcal{A}$ are identical to their views in the system S. However, in order to obtain Claim 2.3.3, we need only analyze the views of B_0 and C_0.

[6]We note the crucial difference between this proof and that of Fischer et al. [38]. In [38], the corrupted party $\mathcal{A}$ is able to simulate the entire A_0–C_1–B_1–A_1 segment of the hexagon system S by itself. Thus, in a *single* execution of Π with B_0 and C_0, party $\mathcal{A}$ can simulate the hexagon. Here, due to the fact that the parties B_1 and C_1 have secret information that $\mathcal{A}$ does not have access to, $\mathcal{A}$ is unable to simulate their behavior itself. Rather, $\mathcal{A}$ needs to redirect messages from the parallel execution of Π_1 in order to complete the hexagon.

Corollary 2.3.6 *No protocol for authenticated Byzantine Agreement that self-composes concurrently (even twice) can tolerate $n/3$ or more corrupted parties.*

Sequential composition. The above results relate to the feasibility of parallel and concurrent composition. However, sequential composition is also an important concern. We note that lower bounds also hold for deterministic protocols. In particular, a deterministic protocol for Byzantine Agreement that runs for r rounds and tolerates $t \geq n/3$ corrupted parties, can be composed sequentially at most $2r{-}1$ times. On the other hand, randomized protocols can be used to overcome this lower bound. We refer the interested reader to [66] for further details.

2.4 Authenticated Byzantine Agreement using Unique Identifiers

In this section we consider an augmentation to the authenticated model in which each execution is assigned a unique and common identifier. We show that in such a model, it is possible to achieve Byzantine Agreement that composes concurrently, for *any* number of corrupted parties. We stress that in the authenticated model itself, it is not possible for the parties to agree on unique and common identifiers, without some external help. This is because by the results of this section, agreeing on a common identifier amounts to solving the Byzantine Agreement problem, and we have proven that this cannot be achieved for $t \geq n/3$ when composition is required. Therefore, these identifiers must come from outside the system (and as such, assuming their existence is an augmentation to the authenticated model).

Intuitively, the existence of unique identifiers helps in the authenticated model for the following reason. Recall that our lower bound is based on the ability of the adversary to *borrow* signed messages from one execution to another. Now, if each signature also includes the session identifier, then the honest parties can easily distinguish between messages signed in this execution and messages signed in a different execution. It turns out that this is enough. That is, we give a transformation from stand-alone Byzantine Agreement protocols based on signature schemes, to protocols that compose concurrently when unique identifiers exist. Our transformation holds for protocols that utilize the signature scheme in a natural way (as will be seen below).

The Transformation: Let Π be a protocol for authenticated Byzantine Agreement that uses a secure signature scheme. We define a modified protocol $\Pi(id)$ that works as follows:

- Each party is given the identifier id as auxiliary input.
- If a party P_i has an instruction in Π to sign a given message m with its signing key sk_i, then P_i signs upon $id \circ m$ instead (where $\circ$ denotes concatenation).
- If a party P_i has an instruction in Π to verify a given signature σ on a message m with a verification key vk_j, then P_i verifies that σ is a valid signature for the message $id \circ m$.

Secure signature schemes. Before proceeding, we present an informal definition of secure signature schemes. A signature scheme is a triplet of algorithms (G, S, V), where G is a probabilistic generator that outputs a pair of signing and verification keys (sk, vk), S is a signing algorithm and V is a verification algorithm. The validity requirement for signature scheme states that for every message m, $V(vk, m, S(sk, m)) = 1$, where $(vk, sk) \leftarrow G(1^n)$ (i.e., honestly generated signatures are always accepted). The validity can be relaxed so that it holds with overwhelming probability, where the probability is taken over the random coin tosses of the different algorithms.

The security requirement of a signature scheme states that the probability that an efficient forging algorithm S^* succeeds in generating any forgery, even when given oracle access to the signing oracle, is negligible. Of course, a successful forgery is only with respect to a message for which S^* did not receive a signature from its oracle. More formally, the following experiment is defined: The generator G is run, outputting a key-pair (vk, sk). Then, S^* is given vk and oracle access to the signing oracle $S(sk, \cdot)$. At the conclusion of the experiment, S^* outputs a pair (m^*, σ^*). Let Q_m be the set of oracle queries by S^*. Then, we say that S^* succeeds if $V(vk, m^*, \sigma^*) = 1$ and $m^* \notin Q_m$. (That is, S^* output a message along with a valid signature, and S^* did not query its oracle with this message.) A signature scheme is existentially secure against chosen-message attacks if for every probabilistic polynomial-time S^*, the probability that S^* succeeds is negligible.

Modifying the definition. For our purposes here, we consider a modification of the above definition where we somewhat restrict the forging algorithm S^*. That is, let id be any string. Then, the modification is such that S^* is given access to a restricted signing algorithm $S^{\neg id}(sk, \cdot)$ that signs on any message m that does not have prefix id. Furthermore, S^* must output a pair (m^*, σ^*) where the prefix of m^* equals id. As before, S^* is said to succeed if $V(vk, m^*, \sigma^*) = 1$ and $m^* \notin Q_m$. It is not hard to see that any signature scheme that is existentially secure against chosen-message attacks is also secure with this modification. This is because a successful forgery here contains a message m^* with prefix id. Since S^* only has access to $S^{\neg id}(sk, \cdot)$, it cannot have queried its oracle with m^*. Thus, a regular forging algorithm can also use S^* to generate a forgery.

It is also not hard to see that protocols that use a signature scheme in a natural way, remain secure when the modified definition is used and the appropriate id is required on all messages that are verified (as in our transformation of Byzantine Agreement protocols). This is because the infeasibility of forging messages containing id is preserved and this is the only property of signature schemes usually utilized. We are now ready to state the theorem:

Theorem 2.4.1 *Let Π be a secure protocol for authenticated Byzantine Agreement which uses a public-key infrastructure for a signature scheme that is existentially secure against chosen message attacks. Furthermore, the parties use their secret keys for signing on messages only. Let $\Pi(id)$ be obtained from Π as in the above transformation, and let $id_1, \ldots, id_\ell$ be a series of ℓ unique strings. If the stand-alone protocol $\Pi(id)$ remains secure even when the adversary is given access to restricted signing oracles $S^{\neg id}(sk, \cdot)$ for the secret keys of* all the honest parties, *then the protocols $\Pi(id_1), \ldots, \Pi(id_\ell)$ all solve the Byzantine Agreement problem, even when run concurrently.*

Proof: Intuitively, the security of the protocols $\Pi(id_1), \ldots, \Pi(id_\ell)$ is due to the fact that signatures from $\Pi(id_i)$ cannot be of any help to the adversary in $\Pi(id_j)$. This is because in $\Pi(id_j)$, the honest parties ignore any signature that includes an identifier that is not id_j. Since $id_i \neq id_j$, we have that signatures sent in $\Pi(id_i)$ are useless in $\Pi(id_j)$. Our formal proof of this intuition proceeds by showing how an adversary for a single execution of $\Pi(id)$ can internally simulate the concurrent executions of $\Pi(id_1), \ldots, \Pi(id_\ell)$, thereby reducing the security of the concurrent setting to the stand-alone setting.

Let $\mathcal{A}$ be an adversary who attacks the concurrent executions $\Pi(id_1), \ldots, \Pi(id_\ell)$. By contradiction, assume that $\mathcal{A}$ succeeds in "breaking" one of the $\Pi(id_i)$ executions with non-negligible probability. We construct an adversary $\mathcal{A}'$ who internally incorporates $\mathcal{A}$ and attacks a single execution of $\Pi(id)$ as follows. Intuitively, $\mathcal{A}'$ simulates all executions apart from the one in which $\mathcal{A}$ succeeds in its attack. Formally, $\mathcal{A}'$ first randomly selects an execution $i \in_R \{1, \ldots, \ell\}$. Then, $\mathcal{A}'$ sets $id_i = id$ and chooses unique identifiers id_j (for every $j \neq i$). Next, $\mathcal{A}'$ invokes $\mathcal{A}$ and emulates the concurrent executions of $\Pi(id_1), \ldots, \Pi(id_\ell)$ for $\mathcal{A}$. Adversary $\mathcal{A}'$ does this by playing the roles of the honest parties in all but the i^{th} execution $\Pi(id_i)$. In contrast, in $\Pi(id_i)$ adversary $\mathcal{A}'$ externally interacts with the honest parties and passes messages between them and $\mathcal{A}$. Since $\mathcal{A}'$ is given access to the restricted signing oracles of all the honest parties *and* the honest parties use their signing keys to generate signatures only, $\mathcal{A}'$ is able to generate the honest parties' messages in all the executions $\Pi(id_j)$ for $j \neq i$. (Recall that in these executions, the prefix of every signed message is $id_j \neq id_i$ and thus the restricted oracle suffices.) Therefore, the emulation by $\mathcal{A}'$ of the concurrent executions for $\mathcal{A}$ is perfect. This implies that $\mathcal{A}'$ succeeds in "breaking" $\Pi(id)$ with success probability that equals $1/\ell$ times $\mathcal{A}$'s success probability in the concurrent setting. Thus, $\mathcal{A}'$ succeeds with non-negligible probability and this contradicts the assumed stand-alone security of $\Pi(id)$ even when $\mathcal{A}'$ is given access to the restricted oracles. ■

It is easy to verify that the protocols of [71, 62, 34] for authenticated Byzantine Agreement all fulfill the requirements in the assumption of Theorem 2.4.1. We therefore obtain the following corollary:

Corollary 2.4.2 *There exist protocols for authenticated Byzantine Agreement that tolerate any $t < n$ corruptions and remain secure under concurrent executions, assuming that global unique identifiers are allocated to each execution.*

We conclude by noting that it is not at all clear how the augmentation to the authenticated model of unique identifiers can be achieved in practice. In particular, requiring the on-line participation of a trusted party who assigns identifiers to every execution is clearly impractical. (Furthermore, such a party could just be used to directly implement broadcast.) However, we do note one important scenario where Theorem 2.4.1 *can* be applied. As we have mentioned, secure protocols often use many invocations of a broadcast primitive. Furthermore, in order to improve round efficiency, in any given round many broadcasts may be simultaneously executed. The key point here is that *within* the secure protocol, unique identifiers can be allocated to each broadcast by the protocol designer. Therefore, authenticated Byzantine Agreement can be used. Of course, this does not change the fact that the secure protocol itself will not compose in parallel or concurrently. However, it does mean that its security is guaranteed in the stand-alone setting, and a physical broadcast channel is not necessary.

Chapter 3

Secure Computation Without Agreement

In the previous chapter, we have shown that authenticated Byzantine Agreement protocols cannot be composed concurrently (or even in parallel) when a third or more of the parties are corrupted. An immediate and important ramification of this result relates to the self-composition of secure multi-party computation. All known protocols for general secure multi-party computation strongly rely on the extensive use of a broadcast primitive. When a third or more of the parties are corrupted, this broadcast is implemented using authenticated Byzantine Agreement. Essentially, this use of Byzantine Agreement cannot be eliminated since the standard definition of secure computation (for the case that less than 1/2 of the parties are corrupted) actually implies Byzantine Agreement. Moreover, it is accepted folklore that the use of a broadcast channel is essential for achieving secure multiparty computation, in the case that 1/3 or more of the parties are corrupted. Due to the above state of affairs, *there are currently no known composable protocols* for secure multi-party computation in the point-to-point network model.

In this chapter we show that secure computation can be achieved without a broadcast channel, and thus without authenticated Byzantine Agreement. Specifically, we present a new definition of secure computation that only mildly relaxes previous definitions and that can be achieved without using a broadcast channel. The new definition separates the issue of agreement from the central security issues of privacy and correctness in secure computation. As a result the lower bounds of Byzantine Agreement no longer apply to secure computation. Indeed, we prove that secure multi-party computation can be achieved for any number of corrupted parties and without a broadcast channel (or trusted preprocessing phase as required for running authenticated Byzantine Agreement). An important corollary of the results in this chapter is the ability to obtain multi-party protocols in a point-to-point network that self-compose.

3.1 Introduction

3.1.1 Background

Security in multi-party computation. A secure multi-party protocol should protect honest parties from the malicious behavior of corrupted parties. In particular, malicious parties should not be able to learn anything more than their prescribed output (*privacy*) and the outputs of the computation should be as presribed (*correctness*). The security of multi-party computation [53, 4, 69, 13] is formalized in the following way. Consider an ideal world in which an external trusted party is willing to help the parties carry out their computation. An *ideal computation* takes place in the ideal world by having the parties simply send their inputs to the trusted party. This trusted party then computes the desired function and passes each party its prescribed output. A *real protocol* that is run by the parties (in a world where no trusted party exists) is said to be secure, if no adversary controlling a coalition of corrupted parties can do more harm in a real execution that in the above ideal computation. More formally, security is said to hold if a real execution can be emulated within the ideal model.

Notice that in an ideal computation, the above-mentioned properties of privacy and correctness are trivially guaranteed. Additional important properties that are implied are *independence of inputs* (meaning that corrupted parties must choose their inputs independently of the honest parties), *guaranteed output delivery* (meaning that all parties receive output and the adversary cannot conduct a denial of service attack), and *fairness* (meaning that corrupted parties receive output if and only if honest parties do). Since these properties all hold in the ideal model, and since real executions of secure protocols can be emulated in the ideal model, it follows that these properties also all hold in a real execution of a secure protocol. We remark that guaranteeing output delivery and fairness is actually impossible unless a strict majority of the parties are honest. Therefore, when this is not the case (i.e., when $t \geq n/2$), the ideal model is modified so that these properties are not implied. Specifically, under certain conditions, parties are allowed to abort (in which case they output a special symbol $\perp$ rather than their specified output). In addition, fairness takes on different meanings for different values of t. We will single out a few forms of fairness. On the one extreme, we have "complete fairness" that guarantees that if a corrupt party gets its output then all honest parties also get their output. (As we have mentioned, this is only possible when $t < n/2$.) On the other extreme, we have "no fairness" in which the adversary always gets its output and has the power to decide whether or not the honest parties also get output. An intermediate notion that we call "partial fairness" singles out a specified party such that if this specified party is honest then complete fairness is achieved. On the other hand, if the specified party is corrupt, then no fairness is achieved. Thus, fairness is partial.

Byzantine agreement and secure multi-party computation. There is a close connection between Byzantine agreement and secure multi-party computation. First, Byzantine agreement (or broadcast) is used as a basic and central tool in the construction of secure protocols. In particular, all known protocols for general multi-party computation use a broadcast channel (and implement it using Byzantine agreement or authenticated Byzantine agreement). Second, Byzantine agreement is actually a special case of secure computation (this holds by the standard definition taken for the case that $t < n/2$ where output delivery is guaranteed). Therefore, all the lower bounds relating to Byzantine agreement immediately apply to secure multi-party computation. In particular, the Byzantine agreement problem cannot be solved for any $t \geq n/3$ [71]. Thus, it is also impossible to achieve secure computation with guaranteed output delivery in a point-to-point network for $t \geq n/3$. On the other hand, for $t < n/2$ it *is* possible to obtain secure computation with guaranteed output delivery assuming a broadcast channel. This means that in order to achieve such secure computation for the range of $n/3 \leq t < n/2$, either a physical broadcast channel or a trusted pre-processing phase for running authenticated Byzantine agreement *must* be assumed.

In the previous chapter, it was shown that authenticated Byzantine agreement cannot be composed (concurrently or even in parallel), unless $t < n/3$. This has the following ramifications. On the one hand, in the range of $n/3 \leq t < n/2$, it is *impossible* to obtain secure computation that composes without using a physical broadcast channel. This is because such a protocol in the point-to-point network model and with trusted pre-processing would imply authenticated

Byzantine agreement that composes. On the other hand, as we have mentioned, in the range of $t \geq n/2$ the definitions of secure computation do not imply Byzantine agreement. Nevertheless, all protocols for secure computation in this range make extensive use of a broadcast primitive. The impossibility of composing authenticated Byzantine agreement puts this whole body of work into question when composition is required. Specifically without using a physical broadcast channel, none of these protocols compose (even in parallel). In summary, there are no known protocols for secure computation in a point-to-point network that compose in parallel or concurrently, for any $t \geq n/3$. Needless to say, the requirement of a physical broadcast channel is very undesirable (and often unrealistic).

3.1.2 Our Results

We present a mild relaxation of the standard definition of secure multi-party computation that decouples the issue of *agreement* from the issue of *secure multi-party computation.* In particular, our definition focuses on the central issues of privacy and correctness. Loosely speaking, our definition is different in the following way. As we have mentioned, for the case of $t \geq n/2$, it is impossible to guarantee output delivery and therefore some parties may conclude with a special abort symbol $\perp$, and not with their output. Previously [44], it was required that either *all* honest parties receive their outputs or *all* honest parties output $\perp$.[1] Thus the parties all *agree* on whether or not output was received. On the other hand, in our definition some honest parties may receive output while some receive $\perp$, and the requirement of agreement is removed. We stress that this is the only difference between our definition and the previous ones.

We show that it is possible to achieve secure computation according to the new definition for *any* $t < n$ and *without* a broadcast channel or setup assumption (assuming the same computational assumptions made, if any, by corresponding protocols that did use broadcast channels.) Thus, the lower bounds for Byzantine agreement indeed do not imply lower bounds for secure multi-party computation. We note that our result holds in both the information theoretic and the computational models.

A hierarchy of definitions. In order to describe our results in more detail, we present a hierarchy of definitions for secure computation. All the definition fulfill the properties of privacy and correctness. The hierarchy that we present here relates to the issues of abort (or failure to receive output) and fairness.

1. *Secure computation without abort:* According to this definition, all parties are guaranteed to receive their output. (This is what we previously called "guaranteed output delivery".) This is the standard definition for the case of honest majority (i.e., $t < n/2$). Since all honest parties receive output, complete fairness is always obtained here.

2. *Secure computation with unanimous abort*: In this definition, it is ensured that either all honest parties receive their outputs or all honest parties abort. This definition can be considered with different levels of fairness:

   (a) *Complete fairness:* Recall that when complete fairness is achieved, the honest parties are guaranteed to receive output if the adversary does. Thus, here one of two cases can occur. Either all parties receive output or all parties abort. Thus, the adversary

[1]We note that in private communication, Goldreich stated that the requirement in [44] of having all parties abort or all parties receive output was only made in order to simplify the definition.

can conduct a denial of service attack, but nothing else. (This definition can only be achieved in the case of $t < n/2$.)

   (b) *Partial fairness:* As in the case of complete fairness, the adversary may disrupt the computation and cause the honest parties to abort without receiving their prescribed output. However, unlike above, the adversary may receive the corrupted parties' outputs, even if the honest parties abort (and thus the abort is not always fair). In particular, the protocol *specifies* a single party such that the following holds. If this party is honest, then complete fairness is essentially achieved (i.e., either all parties abort or all parties receive correct output). On the other hand, if the specified party is corrupt, then fairness may be violated. That is, the adversary receives the corrupted parties' outputs first, and then decides whether or not the honest parties all receive their correct output or all receive abort (and thus the adversary may receive output while the honest parties do not).

   Although fairness is only guaranteed in the case that the specified party is not corrupted, there are applications where this feature may be of importance. For example, in a scenario where one of the parties may be "more trusted" than others (yet not too trusted), it may be of advantage to make this party the specified party. Another setting where this can be of advantage is one where all the participating parties are trusted. However, the problem that may arise is that of an external party "hacking" into the machine of one of the parties. In such a case, it may be possible to provide additional protection to the specified party.

   (c) *No fairness:* This is the same as in the case of partial fairness except that the adversary always receives the corrupted parties' outputs first (i.e., there is no specified party).

   We stress that in all the above three definitions, if one honest party aborts then so do all honest parties, and thus all are aware of the fact that the protocol did not successfully terminate. This feature of having all parties succeed or fail together may be an important one in some applications.

3. *Secure computation with abort:* The only difference between this definition and the one immediately preceding it, is that some honest parties may receive output while others abort. That is, the requirement of unanimity with respect to abort is removed. This yields two different definitions, depending on whether partial fairness or no fairness is taken. (Complete fairness is not considered here because it only makes sense in a setting where all the parties, including the corrupted parties, either all receive output or all abort. Therefore, it is not relevant in the setting of secure computation with non-unanimous abort.)

Using the above terminology, the definition proposed by Goldreich [44] for the case of any $t < n$ is that of secure computation with unanimous abort and partial fairness. Our new definition is that of secure computation with abort, and as we have mentioned, its key feature is a decoupling of the issues of secure computation and agreement (or unanimity).

Achieving secure computation with abort. Using the terminology introduced above, our results show that secure computation with abort and partial fairness can be achieved for any $t < n$, and without a broadcast channel or a trusted pre-processing phase. We achieve this result in the following way. First, we define a weak variant of the Byzantine Generals problem, called *broadcast with abort,* in which not all parties are guaranteed to receive the broadcasted value. In particular, there exists a single value x such that every party either outputs x or aborts. Furthermore, when the broadcasting party is honest, the value x equals its input, similarly to the validity condition of

Byzantine Generals. (Notice that in this variant, the parties do not necessarily agree on the output since some may output x while others abort.) We call this "broadcast with abort" because as with secure computation with abort, some parties may output x while other honest parties abort. We show how to achieve this type of broadcast with a simple deterministic protocol that runs in 2 rounds. Secure multi-party computation is then achieved by replacing the broadcast channel in known protocols with a broadcast with abort protocol. Despite the weak nature of agreement in this broadcast protocol, it is nevertheless enough for achieving secure multi-party computation with abort. Since our broadcast with abort protocol runs in only 2 rounds, we also obtain a very efficient transformation of protocols that work with a broadcast channel into protocols that require only a point-to-point network. In summary, we obtain the following theorem:

Theorem 3.1.1 (efficient transformation): *There exists an efficient protocol compiler that receives any protocol* Π *for the* broadcast model *and outputs a protocol* Π' *for the* point-to-point model *such that the following holds: If* Π *securely computes a functionality* f *with unanimous abort and with any level of fairness, then* Π' *securely computes* f *with abort and with no fairness. Furthermore, if* Π *tolerates up to* t *corruptions and runs for* R *rounds, then* Π' *tolerates up to* t *corruptions and runs for* $O(R)$ *rounds.*

Notice that in the transformation of Theorem 3.1.1, protocol Π' does not achieve complete fairness or partial fairness, even if Π did. Thus, fairness may be lost in the transformation. Nevertheless, meaningful secure computation is still obtained and at virtually no additional cost.

When obtaining some level of fairness is important, Theorem 3.1.1 does not provide a solution. We show that partial fairness *can* be obtained without a broadcast channel for the range of $t \geq n/2$ (recall that complete fairness cannot be obtained in this range, even with broadcast). That is, we prove the following theorem:

Theorem 3.1.2 (partial fairness): *For any probabilistic polynomial-time* n*-party functionality* f*, there exists a protocol in the point-to-point model for computing* f *that is secure with abort, partially fair and tolerates any* $t < n$ *corruptions.*

The theorem is proved by first showing that fairness can be boosted in the point-to-point model. That is, given a generic protocol for secure multi-party computation that achieves no fairness, one can construct a generic protocol for secure multi-party computation that achieves partial fairness. (Loosely speaking, a generic protocol is one that can be used to securely compute any efficient functionality.) Applying Theorem 3.1.1 to known protocols for the broadcast model, we obtain secure multi-party computation that achieves no fairness. Then, using the above "fairness boosting", we obtain Theorem 3.1.2. We note that the round complexity of the resulting protocol is of the same order of the "best" generic protocol that works in the broadcast model. In particular, based on the protocol of Beaver et al. [9], we obtain the first constant-round protocol in the point-to-point network for the range of $n/3 \leq t < n/2$.[2] That is:

Corollary 3.1.3 (constant round protocols without broadcast for $t < n/2$): *Assume that there exist public-key encryption schemes* (*or, alternatively, assume the existence of one-way functions and a model with private channels*). *Then, for every probabilistic polynomial-time functionality* f*, there exists a* constant round *protocol in the point-to-point network for computing* f *that is secure with abort, partially fair and tolerates* $t < n/2$ *corruptions.*

[2]For the range of $t < n/3$, the broadcast channel in the protocol of [9] can be replaced by the expected constant-round Byzantine agreement protocol of Feldman and Micali [37]. However, there is no known authenticated Byzantine agreement protocol with analogous round complexity.

Composition of secure multi-party protocols. An important corollary of our new definition is the ability to obtain secure multi-party protocols for $t > n/3$ that self-compose in parallel or concurrently, without a broadcast channel. Since our protocols do not use a broadcast channel (or authenticated Byzantine agreement), the lower bound of Chapter 2 does not apply. Specifically, if a protocol composes when using a broadcast channel, then the transformed protocol in the point-to-point network also composes. This is in contrast to all previous protocols that used authenticated Byzantine agreement in order to replace the broadcast channel.[3] Jumping ahead, we compare the composition achieved here to that obtained in Chapter 4. In Chapter 4, the strong notion of concurrent general composition is obtained. However, the protocol there relies on a common reference string. In contrast, in this chapter, we obtain only the weaker notion of self-composition. Nevertheless, our protocols are in the standard model and do not require any setup assumptions.

Discussion. We propose that the basic definition of secure computation should focus on the issues of privacy and correctness (and independence of inputs). In contrast, the property of agreement should be treated as an additional, and not central, feature. The benefit of taking such a position (irrespective of whether one is convinced conceptually) is that the feasibility of secure computation is completely decoupled from the feasibility of Byzantine agreement. Thus, the lower bounds relating to Byzantine agreement (and authenticated Byzantine agreement) do not imply anything regarding secure computation. Indeed, as we show, "broadcast with abort" is sufficient for secure computation. However, it lacks any flavor of agreement in the classical sense. This brings us to an important observation. Usually, proving a lower bound for a special case casts light on the difficulties in solving the general problem. However, in the case of secure computation this is not the case. Rather, the fact that the lower bounds of Byzantine agreement apply to secure computation is due to marginal issues relating to unanimity regarding the delivery of outputs, and not due to the main issues of security.

3.1.3 Related Work

We have recently learned of two independent and concurrent results [39, 40] studying a problem similar to ours, although apparently for different motivation.[4] In [39], Fitzi et al. study the question of multi-party computation in the case that the number of faults is $t < n/2$. They show that in this case, it is possible to achieve weak Byzantine agreement (where loosely speaking, either all honest parties abort or all honest parties agree on the broadcasted value). (We note that their protocol is probabilistic and "breaks" the $t < n/3$ lower-bound on deterministic weak Byzantine Agreement protocols of Lamport [63].) They further show how this can be used in order to obtain secure computation with unanimous abort and complete fairness for the case of $t < n/2$. Thus for the range of $n/2 \leq t < n/3$ their solution achieves complete fairness whereas ours achieves only partial fairness.

In subsequent work [40], Fitzi et al. studied the question of Byzantine agreement for any $t < n$ and whether its relaxation to weak Byzantine Agreement can be achieved without preprocessing. They show that it is indeed possible to achieve (randomized) weak Byzantine Agreement for *any*

[3]Unfortunately, full proofs regarding the composition of protocols using a broadcast channel do not appear in the literature. In this footnote we momentarily allow ourselves to rely on the belief/conjecture that in the broadcast model the protocol of [76] for $t < n/2$ self-composes concurrently, and the protocol of [52] for any $t < n$ self-composes in parallel. We can therefore conclude that in the point-to-point network, concurrent self-composition of secure multi-party computation is possible for $t < n/2$, and parallel self-composition is possible for any t.

[4]We were informed of this work while presenting our work at a seminar at MIT, February 14 2002.

$t < n$, in $O(t)$ rounds. They also show how their weak Byzantine Agreement protocol can be used to obtain secure computation with unanimous abort and partial fairness for any $t < n$.

In comparison, we achieve secure computation with (non-unanimous) abort and partial fairness for any $t < n$. However, our focus is different. In particular, our results emphasize the fact that the issue of agreement is not central to the task of secure computation. Furthermore, removing this requirement enables us to remove the broadcast channel with almost no cost. This also results in our obtaining a round-preserving transformation of secure protocols in the broadcast model to those in the point-to-point model. This is in contrast to [39, 40] who use their weak Byzantine agreement protocol in order to setup a public-key infrastructure for authenticated Byzantine agreement. They therefore incur the cost of setting up this infrastructure along with a cost of $t + 1$ rounds for simulating every broadcast in the original protocol. Our protocols are therefore significantly more round efficient.[5] Finally we note that we can use the weak Byzantine Agreement protocol of [40] to transform any generic r-round protocol for secure computation with abort into an $(r+t)$-round protocol with unanimous abort (and the same level of fairness). This is achieved by having the parties broadcast whether they received outputs or not after the protocol with abort concludes. It is enough to use weak Byzantine agreement for this broadcast. We therefore reduce the $O(tr)$ round complexity of [40] to $O(r+t)$, while achieving the same level of security.

Recall that Canetti in [14] introduced a communication model where the adversary has control over the delivery of messages. Essentially, this definition also decouples secure computation from agreement because parties are never guaranteed to get output. In particular, the adversary is allowed to deliver output to whomever it wishes, and only these parties will ever receive output. However, the motivation of [14] is different; it aims to decouple the issue of guaranteed output delivery from the main issues of secure computation. On the other hand, we focus on the question of agreement by the parties on whether or not output was delivered.

3.2 Definitions – Secure Computation

In this section we present definitions for secure multi-party computation. The basic description and definitions are based on [44], which in turn follows [53, 4, 69, 13]. We actually consider a number of definitions here. In particular, we present formal definitions for secure computation with *unanimous abort* and with *abort,* with *complete fairness, partial fairness,* and *no fairness.* In addition, we refer to *secure computation without abort.* This is the standard definition used when more than half the parties are honest. According to this definition, all parties receive the output and the adversary cannot disrupt the computation. However, we will not formally present this definition here.

Notation: We denote by U_k the uniform distribution over $\{0,1\}^k$; for a set S we denote $s \in_R S$ when s is chosen uniformly from S; finally, computational indistinguishability is denoted by $\overset{c}{\equiv}$ and statistical closeness by $\overset{s}{\equiv}$. The security parameter is denoted by k.

[5]We note one subtle, yet important caveat. Given a *generic* protocol for secure computation that uses a broadcast channel and runs for r rounds, we obtain an $O(r)$-round protocol that is secure with abort and partially fair (this is in contrast to the $O(tr)$ round complexity of [39, 40]). However, given a protocol that solves a specific secure computation problem, our transformation does not achieve partial fairness. In order to achieve partial fairness, we must revert to a generic protocol. On the other hand, the transformation of [39, 40] works for any protocol. Thus, given a very efficient protocol for a specific problem that achieves partial fairness, it may be "cheaper" to use [39, 40] rather than our results.

Multi-party computation. A multi-party protocol problem (for n parties $P_1, \ldots, P_n$) is cast by specifying a random process that maps vectors of inputs to vectors of outputs (one for each party). We refer to such a process as an n-ary functionality and denote it $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$, where $f = (f_1, \ldots, f_n)$. That is, for a vector of inputs $\overline{x} = (x_1, \ldots, x_n)$, the output-vector is a random variable $(f_1(\overline{x}), \ldots, f_n(\overline{x}))$ ranging over vectors of strings. The output for the i^{th} party (with input x_i) is defined to be $f_i(\overline{x})$.

Adversarial behavior. Loosely speaking, the aim of a secure multi-party protocol is to protect the honest parties against dishonest behavior from the corrupted parties. This "dishonest behavior" can manifest itself in a number of ways; in this paper we focus on *malicious* adversaries. Such an adversary may arbitrarily deviate from the specified protocol. When considering malicious adversaries, there are certain undesirable actions that cannot be prevented. Specifically, parties may refuse to participate in the protocol, may substitute their local input (and enter with a different input) and may cease participating in the protocol before it terminates.

Formally, the adversary is modeled by a non-uniform Turing machine: in the computational model this machine is polynomial-time whereas in the information-theoretic model it is unbounded. (We note that by standard arguments, we can assume that the adversary is deterministic.) For simplicity, in this work we consider a *static corruption* model. Therefore, at the beginning of the execution, the adversary is given a set I of corrupted parties which it controls. Then, throughout the computation, the adversary obtains the views of the corrupted parties, and provides them with the messages that they are to send.

Security of protocols (informal). The security of a protocol is analyzed by comparing what an adversary can do in the protocol to what it can do in an ideal scenario that is secure by definition. This is formalized by considering an *ideal* computation involving an incorruptible *trusted party* to whom the parties send their inputs. The trusted party computes the functionality on the inputs and returns to each party its respective output. Loosely speaking, a protocol is secure if any adversary interacting in the real protocol (where no trusted party exists) can do no more harm than if it was involved in the above-described ideal computation. We begin by formally defining this ideal computation.

3.2.1 Execution in the ideal model

The ideal model differs for each of the definitions. We therefore present each one separately (see Section 3.1.2 for an outline of the different definitions).

1. Secure computation with unanimous abort and complete fairness: This definition requires complete fairness. That is, either all parties (including the corrupted parties) receive output or all parties abort. Therefore, the abort is also unanimous. We note that this definition is only achievable when the number of corrupted parties is less than $n/2$ (i.e., $|I| < n/2$), even assuming a broadcast channel. Recall that a malicious party can always substitute its input or refuse to participate. Therefore, the ideal model takes these inherent adversarial behaviors into account; i.e., by giving the adversary the ability to do this also in the ideal model. An ideal execution proceeds as follows:

Inputs: Each party obtains its respective input from the input vector $\overline{x} = (x_1, \ldots, x_n)$.

Send inputs to trusted party: An honest party always sends its input x to the trusted party. The corrupted parties may, depending on their inputs $\{x_i\}_{i \in I}$, either abort or send modified values $x'_i \in \{0,1\}^{|x_i|}$ to the trusted party. Denote the sequence of inputs obtained by the trusted party by $\overline{x}' = (x'_1, \ldots, x'_n)$ (for honest parties, $x' = x$ always).

Trusted party answers the parties: In case $\overline{x}'$ is a valid input sequence, the trusted party computes $f(\overline{x}')$ and sends $f_i(\overline{x}')$ to party P_i for every i. Otherwise (i.e., in case a corrupted party aborted or sent a non-valid input), the trusted party replies to all parties with a special abort symbol $\perp$.

Outputs: An honest party always outputs the message that it received from the trusted party, whereas the corrupted parties output nothing (say, λ). On the other hand, the adversary outputs an arbitrary function of the initial inputs $\{x_i\}_{i \in I}$ and the messages the corrupted parties received from the trusted party.

Definition 3.2.1 (ideal-model computation with unanimous abort and complete fairness): *Let $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ be an n-ary functionality, where $f = (f_1, \ldots, f_n)$, and let $I \subset [n]$ be such that for every $i \in I$, the adversary $\mathcal{A}$ controls P_i (this is the set of corrupted parties). Then, the* joint execution of f under $(\mathcal{A}, I)$ in the ideal model *on input vector $\overline{x} = (x_1, \ldots, x_n)$, denoted $\text{IDEAL}^{(1)}_{f,(\mathcal{A},I)}(\overline{x})$, is defined as the output vector of $P_1, \ldots, P_n$ and $\mathcal{A}$ resulting from the above described ideal process.*

2. Secure computation with unanimous abort and partial fairness: As before, a malicious party can always substitute its input or refuse to participate. However, when there are a half or less honest parties, it is not possible to continue computing in the case that the adversary ceases prematurely (this definition is usually used when the number of corrupted parties is not limited in any way). Thus, we cannot prevent the "early abort" phenomenon in which the adversary receives its output, whereas the honest parties do not receive theirs. Nevertheless, party P_1 is specified so that if it is honest, then complete fairness is achieved. In contrast, if it corrupted, then the adversary receives the corrupted parties' outputs first and then can decide whether or not the honest parties receive output or abort. We note that the abort is unanimous and thus if one honest party aborts, then so do all honest parties. An ideal execution proceeds as follows:

Inputs: Each party obtains its respective input from the input vector $\overline{x} = (x_1, \ldots, x_n)$.

Send inputs to trusted party: An honest party always sends its input x to the trusted party. The corrupted parties may, depending on their inputs $\{x_i\}_{i \in I}$, either abort or send modified values $x'_i \in \{0,1\}^{|x_i|}$ to the trusted party. Denote the sequence of inputs obtained by the trusted party by $\overline{x}' = (x'_1, \ldots, x'_n)$ (for honest parties, $x' = x$ always).

Trusted party answers first party: In case $\overline{x}'$ is a valid input sequence, the trusted party computes $f(\overline{x}')$ and sends $f_1(\overline{x}')$ to party P_1. Otherwise (i.e., in case a corrupted party aborted or sent a non-valid input), the trusted party replies to all parties with a special symbol, $\perp$.

Trusted party answers remaining parties: If the first party is not corrupted (i.e., $1 \notin I$), then the trusted party sends $f_j(\overline{x}')$ to party P_j, for *every* j.

In case the first party is corrupted, then for every $i \in I$, the trusted party sends $f_i(\overline{x})$ to party P_i (i.e., the corrupted parties receive their outputs first). Then P_1, depending on the views of all the corrupted parties and controlled by the adversary, instructs the trusted party to either send $f_j(\overline{x}')$ to P_j for every $j \notin I$, or to send $\perp$ to P_j for every $j \notin I$.[6]

Outputs: An honest party always outputs the message that it received from the trusted party, whereas the corrupted parties output nothing (say, λ). On the other hand, the adversary outputs an arbitrary function of the initial inputs $\{x_i\}_{i\in I}$ and the messages the corrupted parties received from the trusted party.

Definition 3.2.2 (ideal-model computation with unanimous abort and partial fairness): *Let $f : (\{0,1\}^*)^n \rightarrow (\{0,1\}^*)^n$ be an n-ary functionality, where $f = (f_1, \ldots, f_n)$, and let $I \subset [n]$ be such that for every $i \in I$, the adversary $\mathcal{A}$ controls P_i (this is the set of corrupted parties). Then, the* joint execution of f under $(\mathcal{A}, I)$ in the ideal model *on input vector $\overline{x} = (x_1, \ldots, x_n)$, denoted $\text{IDEAL}^{(2)}_{f,(\mathcal{A},I)}(\overline{x})$, is defined as the output vector of $P_1, \ldots, P_n$ and $\mathcal{A}$ resulting from the above described ideal process.*

3. Secure computation with unanimous abort and no fairness: This definition is very similar to the previous one, except that there is no specified party. Rather, the adversary first receives the output of the corrupted parties. Then, it decides whether all the honest parties receive output or they all abort. Formally,

Inputs: Each party obtains its respective input from the input vector $\overline{x} = (x_1, \ldots, x_n)$.

Send inputs to trusted party: An honest party always sends its input x to the trusted party. The corrupted parties may, depending on their inputs $\{x_i\}_{i\in I}$, either abort or send modified values $x'_i \in \{0,1\}^{|x_i|}$ to the trusted party. Denote the sequence of inputs obtained by the trusted party by $\overline{x}' = (x'_1, \ldots, x'_n)$ (for honest parties, $x' = x$ always).

Trusted party answers adversary: In case $\overline{x}'$ is a valid input sequence, the trusted party computes $f(\overline{x}')$ and sends $f_i(\overline{x}')$ to party P_i for every $i \in I$. Otherwise (i.e., in case a corrupted party aborted or sent a non-valid input), the trusted party replies to all parties with a special symbol, $\perp$.

Trusted party answers remaining parties: The adversary, depending on the views of all the corrupted parties, instructs the trusted party to either send $f_j(\overline{x}')$ to P_j for every $j \notin I$, or to send $\perp$ to P_j for every $j \notin I$.

Outputs: An honest party always outputs the message that it received from the trusted party, whereas the corrupted parties output nothing (say, λ). On the other hand, the adversary outputs an arbitrary function of the initial inputs $\{x_i\}_{i\in I}$ and the messages the corrupted parties received from the trusted party.

Definition 3.2.3 (ideal-model computation with unanimous abort and no fairness): *Let $f : (\{0,1\}^*)^n \rightarrow (\{0,1\}^*)^n$ be an n-ary functionality, where $f = (f_1, \ldots, f_n)$, and let $I \subset [n]$ be such that for every $i \in I$, the adversary $\mathcal{A}$ controls P_i (this is the set of corrupted parties). Then, the* joint execution of f under $(\mathcal{A}, I)$ in the ideal model *on input vector $\overline{x} = (x_1, \ldots, x_n)$, denoted $\text{IDEAL}^{(3)}_{f,(\mathcal{A},I)}(\overline{x})$, is defined as the output vector of $P_1, \ldots, P_n$ and $\mathcal{A}$ resulting from the above described ideal process.*

[6]An equivalent definition to this one says that P_1 always instructs the trusted party to send the outputs or $\perp$. However, an honest P_1 always instructs the trusted party to provide all parties with outputs. In contrast, a corrupted party can decide whatever it wishes.

The above three definitions all relate to the case of secure computation with unanimous abort. We now present the analogous definitions for the case of secure computation with abort. The only difference between the definitions is regarding the "trusted party answers remaining parties" item. In the above definitions all honest parties either receive their output or they receive $\perp$. However, here some of these parties may receive their (correct) output and some may receive $\perp$. We only present definitions for partial and no fairness (complete fairness only makes sense if all parties, including the adversary, either receive their outputs or $\perp$).

4. Secure computation with abort and partial fairness: As we have mentioned, the only difference between this definition and the analogous definition with unanimous abort is that if party P_1 is corrupted, then it may designate who does and does not receive output. We repeat only the relevant item:

Trusted party answers remaining parties: If the first party is not corrupted (i.e., $1 \notin I$), then the trusted party sends $f_j(\overline{x}')$ to party P_j, for *every* j.

In case the first party is corrupted, then for every $i \in I$, the trusted party sends $f_i(\overline{x}')$ to P_i (i.e., the corrupted parties receive their output first). Then P_1, depending on the views of all the corrupted parties and controlled by the adversary, chooses a subset of the honest parties $J \subseteq [n] \setminus I$ and sends J to the trusted party. The trusted party then sends $f_j(\overline{x}')$ to P_j for every $j \in J$, and $\perp$ to all other honest parties.

Definition 3.2.4 (ideal-model computation with abort and partial fairness): *Let* $f : (\{0,1\}^*)^n \rightarrow (\{0,1\}^*)^n$ *be an* n*-ary functionality, where* $f = (f_1, \ldots, f_n)$*, and let* $I \subset [n]$ *be such that for every* $i \in I$*, the adversary* $\mathcal{A}$ *controls* P_i *(this is the set of corrupted parties). Then, the* joint execution of f under $(\mathcal{A}, I)$ in the ideal model *on input vector* $\overline{x} = (x_1, \ldots, x_n)$*, denoted* $\text{IDEAL}^{(4)}_{f,(\mathcal{A},I)}(\overline{x})$*, is defined as the output vector of* $P_1, \ldots, P_n$ *and* $\mathcal{A}$ *resulting from the above described ideal process.*

5. Secure computation with abort and no fairness: This definition is very similar to the previous one, except that the first party P_1 does not receive its output first. Rather, the adversary always receives the output of the corrupted parties first. Then, it designates which honest parties receive their output and which receive $\perp$. We repeat only the relevant item:

Trusted party answers remaining parties: The adversary, depending on the views of all the corrupted parties, chooses a subset of the honest parties $J \subseteq [n] \setminus I$ and sends J to the trusted party. The trusted party then sends $f_j(\overline{x}')$ to P_j for every $j \in J$, and $\perp$ to all other honest parties.

Definition 3.2.5 (ideal-model computation with abort and no fairness): *Let* $f : (\{0,1\}^*)^n \rightarrow (\{0,1\}^*)^n$ *be an* n*-ary functionality, where* $f = (f_1, \ldots, f_n)$*, and let* $I \subset [n]$ *be such that for every* $i \in I$*, the adversary* $\mathcal{A}$ *controls* P_i *(this is the set of corrupted parties). Then, the* joint execution of f under $(\mathcal{A}, I)$ in the ideal model *on input vector* $\overline{x} = (x_1, \ldots, x_n)$*, denoted* $\text{IDEAL}^{(5)}_{f,(\mathcal{A},I)}(\overline{x})$*, is defined as the output vector of* $P_1, \ldots, P_n$ *and* $\mathcal{A}$ *resulting from the above described ideal process.*

3.2.2 Execution in the real model

We now define a real model execution. In the real model, the parties execute the protocol in a *synchronous* network with *rushing.* That is, the execution proceeds in rounds: each round consists of a send phase (where parties send their message from this round) followed by a receive phase (where they receive messages from other parties). We stress that the messages sent by an honest party in a given round depend on the messages that it received in previous rounds only. On the other hand, the adversary can compute its messages in a given round based on the messages that it receives from the honest parties in the same round. The term rushing refers to this additional adversarial capability.

In this work, we consider a scenario where the parties are connected via a fully connected point-to-point network (and there is no broadcast channel). We refer to this model as the *point-to-point model* (in contrast to the *broadcast model* where the parties are given access to a physical broadcast channel in addition to the point-to-point network). The communication lines between parties are assumed to be ideally authenticated and private (and thus the adversary cannot modify or read messages sent between two honest parties).[7] We assume that any message sent by an honest party to another honest party is received immediately. Finally, we note that we do not assume a preprocessing setup phase.

Throughout the execution, the honest parties all follow the instructions of the prescribed protocol, whereas the corrupted parties receive their (arbitrary) instructions from the adversary. Likewise, at the conclusion of the execution, the honest parties output their prescribed output from the protocol, whereas the corrupted parties output nothing. On the other hand, the adversary outputs an arbitrary function of its view of the computation (which contains the views of all the corrupted parties). Without loss of generality, we assume that the adversary always outputs its view (and not some function of it). Formally,

Definition 3.2.6 (real-model execution): *Let f be an n-ary functionality and let Π be a multi-party protocol for computing f. Furthermore, let $I \subset [n]$ be such that for every $i \in I$, the adversary $\mathcal{A}$ controls P_i (this is the set of corrupted parties). Then, the* joint execution of Π under $(\mathcal{A}, I)$ in the real model *on input vector $\overline{x} = (x_1, \ldots, x_n)$, denoted $\textsc{real}_{\Pi,(\mathcal{A},I)}(\overline{x})$, is defined as the output vector of $P_1, \ldots, P_n$ and $\mathcal{A}$ resulting from the protocol interaction, where for every $i \in I$, party P_i computes its messages according to $\mathcal{A}$, and for every $j \notin I$, party P_j computes its messages according to Π.*

3.2.3 Security as emulation of a real execution in the ideal model

Having defined the ideal and real models, we can now define security of protocols. Loosely speaking, the definition asserts that the adversary can do no more harm in a real protocol execution that in the ideal model (where security trivially holds). This is formulated by saying that adversaries in the ideal model are able to simulate adversaries in an execution of a secure real-model protocol. The definition of security comes in two flavors. In the first, we consider polynomial-time bounded adversaries, and require that the simulation be such that the real-model and ideal-model output

[7]We note that when the parties are assumed to be computationally bounded, privacy can be achieved over authenticated channels by using public-key encryption. Therefore, in such a setting, the requirement that the channels be private is not essential. However, we include it for simplicity.

One can argue that achieving authenticated and private channels in practice essentially requires a trusted preprocessing phase for setting up a public-key infrastructure. Therefore, there is no reason not to utilize this preprocessing phase in the secure multi-party computation as well. In such a case, the preprocessing phase could be used in order to implement authenticated Byzantine Agreement (and thereby achieve secure broadcast for any number of corrupted parties). However, we claim that the issue of achieving "secure communication channels" should be separated from the issue of achieving "secure broadcast". An example of why this is important was demonstrated in Chapter 2 where we showed that authenticated Byzantine Agreement does not compose (in parallel or concurrently) when 2/3 or less of the parties are honest. On the other hand, secure channels can be achieved without any limitation on the protocol using them [18]; in particular, without restrictions on composability and the number of faults.

distributions are computationally indistinguishable. On the other hand, in the second, we consider unbounded adversaries and require that the simulation be such that the output distributions of the two models are statistically close.

Definition 3.2.7 (computational security): *Let f and Π be as above. We say that protocol Π is a protocol for* computational t-secure computation with unanimous abort (*resp.,* with abort) *and with* complete fairness (*resp., with* partial fairness *or with* no fairness)*, if for every non-uniform polynomial-time adversary $\mathcal{A}$ for the real model, there exists a non-uniform polynomial-time adversary $\mathcal{S}$ for the ideal model, such that for every $I \subset [n]$ with $|I| < t$,*

$$\{\text{IDEAL}^{(\alpha)}_{f,(\mathcal{S},I)}(\overline{x})\}_{k\in\mathsf{N},\overline{x}\in(\{0,1\}^k)^n} \overset{\text{c}}{\equiv} \{\text{REAL}_{\Pi,(\mathcal{A},I)}(\overline{x})\}_{k\in\mathsf{N},\overline{x}\in(\{0,1\}^k)^n}$$

where the value of $\alpha \in \{1,2,3,4,5\}$ depends on whether secure computation with unanimous abort or with abort is being considered, and whether complete fairness, partial fairness or no fairness is required.

Definition 3.2.8 (information-theoretic security): *Let f and Π be as above. We say that protocol Π is a protocol for* information-theoretic t-secure computation with unanimous abort (*resp.,* with abort) *and with* complete fairness (*resp., with* partial fairness *or with* no fairness)*, if for every non-uniform adversary $\mathcal{A}$ for the real model, there exists a non-uniform adversary $\mathcal{S}$ for the ideal model such that for every $I \subset [n]$ with $|I| < t$,*

$$\{\text{IDEAL}^{(\alpha)}_{f,(\mathcal{S},I)}(\overline{x})\}_{k\in\mathsf{N},\overline{x}\in(\{0,1\}^k)^n} \overset{\text{s}}{\equiv} \{\text{REAL}_{\Pi,(\mathcal{A},I)}(\overline{x})\}_{k\in\mathsf{N},\overline{x}\in(\{0,1\}^k)^n}$$

where the value of $\alpha \in \{1,2,3,4,5\}$ depends on whether secure computation with unanimous abort or with abort is being considered, and whether complete fairness, partial fairness or no fairness is required.

3.3 Broadcast with Abort

In this section, we present a weak variant of the Byzantine Generals problem, that we call *"broadcast with abort"*. The main idea is to weaken both the agreement and validity requirements so that some parties may output the broadcast value x while others output $\perp$. Formally,

Definition 3.3.1 (broadcast with abort): *Let $P_1, \ldots, P_n$, be n parties and let P_1 be the dealer with input x. In addition, let $\mathcal{A}$ be an adversary who controls up to t of the parties (which may include P_1). A protocol solves the* broadcast with abort *problem, tolerating t corruptions, if for any adversary $\mathcal{A}$ the following three properties hold:*

1. Agreement: *If an honest party outputs x', then all honest parties output either x' or $\perp$.*
2. Validity: *If P_1 is honest, then all honest parties output either x or $\perp$.*
3. Non-triviality: *If all parties are honest, then all parties output x.*

(The non-triviality requirement is needed to rule out a protocol in which all parties simply output $\perp$ and halt.) We now present a simple protocol that solves the broadcast with abort problem for *any* t. As we will see later, despite its simplicity, this protocol suffices for obtaining secure computation with abort. The protocol appears in Figure 3.1.

Protocol BWA

- **Input:** P_1 has a value x to broadcast.
- **The Protocol:**
  1. P_1 sends x to all parties.
  2. Denote by x^i the value received by P_i from P_1 in the previous round. Then, every party P_i (for $i > 1$) sends its value x^i to all other parties.
  3. Denote the value received by P_i from P_j in the previous round by x^i_j (recall that x^i denotes the value P_i received from P_1 in the first round). Then, P_i outputs x^i if this is the only value that it saw (i.e., if $x^i = x^i_2 = \cdots = x^i_n$). Otherwise, it outputs $\perp$.

     We note that if P_i did not receive any value in the first round, then it always outputs $\perp$.

Figure 3.1: The broadcast with abort protocol

Proposition 3.3.2 *Protocol* BWA *of Figure* 3.1 *solves the broadcast with abort problem, and tolerates* any $t < n$ *corruptions.*

Proof: The fact that the non-triviality condition is fulfilled is immediate. We now prove the other two conditions:

1. *Agreement:* Let P_i be an honest party, such that P_i outputs a value x'. Then, it must be that P_i received x' from P_1 in the first round (i.e., $x^i = x'$). Therefore, P_i sent this value to all other parties in the second round. Now, a party P_j will output x^j if this is the only value that it saw during the execution. However, as we have just seen, P_j definitely saw x' in the second round. Thus, P_j will only output x^j if $x^j = x'$. On the other hand, if P_j does not output x^j, then it outputs $\perp$.

2. *Validity:* If P_1 is honest, then all parties receive x in the first round. Therefore, they will only output x or $\perp$.

This completes the proof. ■

3.3.1 Strengthening Broadcast with Abort

A natural question to ask is whether or not we can strengthen Definition 3.3.1 in one of the following two ways (and still obtain a protocol for $t \geq n/3$):

1. *Strengthen the agreement requirement:* If an honest party outputs a value x', then all honest parties output x'. (On the other hand, the validity requirement remains unchanged.)

2. *Strengthen the validity requirement:* If P_1 is honest, then all honest parties output x. (On the other hand, the agreement requirement remains unchanged.)

It is easy to see that the above strengthening of the agreement requirement results in the definition of weak Byzantine Generals. (The validity and non-triviality requirements combined together are equivalent to the validity requirement of weak Byzantine Generals.) Therefore, there exists no *deterministic* protocol for the case of $t \geq n/3$. For what can be done if one utilizes probabilistic protocols, see the section on recent *related work* (Section 3.1.3). Regarding the strengthening of the validity requirement, the resulting definition implies a problem known as "Crusader Agreement". This was shown to be unachievable for any $t \geq n/3$ by Dolev in [31]. We therefore conclude that the "broadcast with abort" requirements cannot be strengthened in either of the above two ways (for deterministic protocols), without incurring a $t < n/3$ lower bound.

3.4 Secure Computation with Abort and No Fairness

In this section, we show that any protocol for secure computation (with unanimous abort and any level of fairness) that uses a broadcast channel can be "compiled" into a protocol for the point-to-point network that achieves secure computation with abort and no fairness. Furthermore, the fault tolerance of the compiled protocol is the same as the original one. Actually, we assume that the protocol is such that all parties terminate in the same round. We say that such a protocol has *simultaneous termination.* Without loss of generality, we also assume that all parties generate their output in the last round. The result of this section is formally stated in the following theorem:

Theorem 3.4.1 (Theorem 3.1.1 – restated): *There exists a (polynomial-time) protocol compiler that takes any protocol Π (with simultaneous termination) for the* broadcast model*, and outputs a protocol Π' for the* point-to-point model *such that the following holds: If Π is a protocol for information-theoretic (resp., computational) t-secure computation with unanimous abort and any level of fairness, then Π' is a protocol for information-theoretic (resp., computational) t-secure computation with abort and no fairness.*

Combining Theorem 3.4.1 with known protocols (specifically, [76] and [52][8]), we obtain the following corollaries:

Corollary 3.4.2 (information-theoretic security – compilation of [76]): *For any probabilistic polynomial-time n-ary functionality f, there exists a protocol in the point-to-point model, for the information-theoretic $n/2$-secure computation of f with abort and no fairness.*

We note that this result is optimal in the following sense. Ben-Or et al. [10] showed that there are functions for which there do not exist information-theoretically private protocols when $t \geq n/2$. The definition of a "private" (rather than "secure") protocol, is regarding the behavior of corrupted parties. In a private protocol, corrupted parties follow the protocol specification, but attempt to learn more information than intended (such adversarial behavior is known as passive or semi-honest). Therefore, security with abort implies privacy, and this means that for information-theoretic security, resilience of $t \geq n/2$ is not possible. On the other hand, the result is not optimal regarding fairness. (In particular, the recent [39] achieve an analogous result with complete fairness.)

Corollary 3.4.3 (computational security – compilation of [52]): *For any probabilistic polynomial-time n-ary functionality f, there exists a protocol in the point-to-point model, for the computational t-secure computation of f with abort and no fairness, for* any t.

We now proceed to prove Theorem 3.4.1.

[8]Both the [76] and [52] protocols have simultaneous termination

Proof of Theorem 3.4.1: Intuitively, we construct a protocol for the point-to-point model from a protocol for the broadcast model, by having the parties in the point-to-point network simulate the broadcast channel. When considering "pure" broadcast (i.e., Byzantine Generals), this is not possible for $t \geq n/3$. However, it suffices to simulate the broadcast channel using a protocol for "broadcast with abort". Recall that in such a protocol, either the correct value is delivered to all parties, or some parties output $\perp$. The idea is to halt the computation in the case that any honest party receives $\perp$ from a broadcast execution. The point at which the computation halts dictates which parties (if any) receive output. The key point is that if no honest party receives $\perp$, then the broadcast with abort protocol perfectly simulates a broadcast channel. Therefore, the result is that the original protocol (for the broadcast channel) is simulated perfectly until the point that it may prematurely halt.

Components of the compiler:

1. Broadcast with abort executions: Each broadcast of the original protocol (using the assumed broadcast channel) is replaced with an execution of the broadcast with abort protocol.

2. Blank rounds: Following each broadcast with abort execution, a blank round is added in which no protocol messages are sent. Rather, these blank rounds are used by parties to notify each other that they have received $\perp$. Specifically, if a party receives $\perp$ in a broadcast with abort execution, then it sends $\perp$ to all parties in the blank round that immediately follows. Likewise, if a party receives $\perp$ in a blank round, then it sends $\perp$ to all parties in the next blank round (it also does not participate in the next broadcast).

Thus each round of the original protocol is transformed into 3 rounds in the compiled protocol (2 rounds for broadcast with abort and an additional blank round). We now proceed to formally define the protocol compiler:

Construction 1 (protocol compiler): *Given a protocol Π, the compiler produces a protocol Π'. The specification of protocol Π' is as follows:*

- *The parties use* broadcast with abort *in order to emulate each broadcast message of protocol Π. Each round of Π is expanded into 3 rounds in Π':* broadcast with abort *is run in the first 2 rounds, and the third round is a* blank *round. Point-to-point messages of Π are sent unmodified in Π'. The parties emulate Π according to the following instructions:*

  1. Broadcasting messages: *Let P_i be a party who is supposed to broadcast a message m in the j^{th} round of Π. Then, in the j^{th} broadcast simulation of Π' (i.e., in rounds $3j$ and $3j+1$ of Π'), all parties run an execution of* broadcast with abort *in which P_i plays the dealer role and sends m.*

  2. Sending point-to-point messages: *Any message that P_i is supposed to send to P_j over the point-to-point network in the j^{th} round of Π is sent by P_i to P_j over the point-to-point network in round $3j$ of Π'.*

  3. Receiving messages: *For each message that party P_i is supposed to receive from a broadcast in Π, party P_i participates in an execution of* broadcast with abort *as a receiver. If its output from this execution is a message m, then it appends m to its view (to be used for determining its later steps according to Π).*

     If it receives $\perp$ from this execution, then it sends $\perp$ to all parties in the next round (i.e., in the blank round following the execution of broadcast with abort*), and halts immediately.*

4. Blank rounds: *If a party P_i receives $\perp$ in a blank round, then it sends $\perp$ to all parties in the next blank round and halts. In the 2 rounds preceding the next blank round, Party P_i does* not *send any point-to-point messages or messages belonging to a broadcast execution. (We note that if this blank round is the last round of the execution, then P_i simply halts.)*

5. Output: *If a party P_i received $\perp$ at any point in the execution (in an execution of* broadcast with abort *or in a blank round), then it outputs $\perp$. Otherwise, it outputs the value specified by Π.*

In order to prove that Π' is t-secure with abort and no fairness, we first define a different transformation of Π to $\tilde{\Pi}$ which is a hybrid protocol between Π and Π'. In particular, $\tilde{\Pi}$ is still run in the broadcast model. However, it provides the adversary with the additional ability of prematurely halting honest parties. We now define the hybrid protocol $\tilde{\Pi}$ and show that it is t-secure with abort and no fairness:

Lemma 3.4.4 *Let Π be a protocol in the broadcast model that is computational (resp., information-theoretic) t-secure with unanimous abort and any level of fairness. Then, define protocol $\tilde{\Pi}$ (also for the broadcast model) as follows:*

1. *Following each round of Π, add a blank round.*

2. *If in a blank round, P_i receives a $\perp$ message, then P_i sends $\perp$ to P_j for all $j \neq i$ in the next blank round and halts. P_i also does not broadcast any message or send any point-to-point messages in the next round of Π (before the blank round where it sends all the $\perp$ messages).*

3. *Apart from the above, the parties follow the instructions of Π.*

4. Output: *If a party P_i received $\perp$ in any blank round, then it outputs $\perp$. Otherwise, it outputs the value specified by Π.*

Then, $\tilde{\Pi}$ is computational (resp., information-theoretic) t-secure with abort and no fairness.

Proof: We prove this theorem for the case that Π is computationally t-secure with unanimous abort and partial fairness. The other cases (information theoretic security and security with complete fairness or no fairness) are proved in a similar way. Let $\tilde{\mathcal{A}}$ be a real-model adversary attacking $\tilde{\Pi}$. Our aim is to construct an ideal-model simulator $\tilde{\mathcal{S}}$ for $\tilde{\mathcal{A}}$. In order to do this, we must use the fact that for any adversary $\mathcal{A}$ attacking protocol Π, there exists an ideal-model simulator $\mathcal{S}$. Unfortunately we cannot apply $\mathcal{S}$ to $\tilde{\mathcal{A}}$ because $\mathcal{S}$ is a simulator for protocol Π and $\tilde{\mathcal{A}}$ participates in protocol $\tilde{\Pi}$. We therefore first construct an adversary $\mathcal{A}$ that attacks Π from the adversary $\tilde{\mathcal{A}}$ that attacks $\tilde{\Pi}$. The construction of $\mathcal{A}$ is such that the output distribution of an execution of Π with $\mathcal{A}$ is very similar to the output distribution of an execution of $\tilde{\Pi}$ with $\tilde{\mathcal{A}}$. Having constructed $\mathcal{A}$, it is then possible to apply the simulator $\mathcal{S}$ that we know is guaranteed to exist. We therefore obtain a simulator $\tilde{\mathcal{S}}$ for $\tilde{\mathcal{A}}$ by first "transforming" $\tilde{\mathcal{A}}$ into $\mathcal{A}$ and then applying $\mathcal{S}$. As we will show, the resulting simulator $\tilde{\mathcal{S}}$ is as required for $\tilde{\Pi}$. Details follow.

As we have mentioned, we first define the adversary $\mathcal{A}$ who attacks Π. Adversary $\mathcal{A}$ internally invokes $\tilde{\mathcal{A}}$ and therefore has internal communication with $\tilde{\mathcal{A}}$ and external communication with the honest parties executing Π.

Adversary $\mathcal{A}$ for Π:

- **Input:** $\mathcal{A}$ receives an input sequence $\{x_i\}_{i\in I}$ and a series of random-tapes $\{r_i\}_{i\in I}$. (Recall that $\tilde{\mathcal{A}}$ controls all parties in the set I. Thus, corrupted party P_i's input and random-tape equal x_i and r_i, respectively.)
- **Execution:**
  1. *Invoke $\tilde{\mathcal{A}}$:* $\mathcal{A}$ begins by invoking $\tilde{\mathcal{A}}$ upon input sequence $\{x_i\}_{i\in I}$ and random-tapes $\{r_i\}_{i\in I}$.
  2. *Emulation before $\tilde{\mathcal{A}}$ sends any $\perp$ messages:* $\mathcal{A}$ internally passes to $\tilde{\mathcal{A}}$ all the messages that it externally receives from the honest parties (through broadcast or point-to-point communication). Likewise, $\mathcal{A}$ externally broadcasts in Π any message that $\tilde{\mathcal{A}}$ broadcasts in $\tilde{\Pi}$, and $\mathcal{A}$ externally sends P_j in Π any message that $\tilde{\mathcal{A}}$ sends P_j in $\tilde{\Pi}$.
  3. *Emulation after $\tilde{\mathcal{A}}$ sends a $\perp$ message:* Once $\tilde{\mathcal{A}}$ sends a $\perp$ message in an execution of $\tilde{\Pi}$, the honest parties in Π and $\tilde{\Pi}$ may behave differently (in particular, a party receiving $\perp$ may continue to send messages in Π, whereas it would halt in $\tilde{\Pi}$). Therefore, $\mathcal{A}$ filters messages sent by honest parties in Π so that $\tilde{\mathcal{A}}$'s view equals what it would in an execution of $\tilde{\Pi}$. That is:

     In the round of Π following the first $\perp$ message sent by $\tilde{\mathcal{A}}$, adversary $\mathcal{A}$ forwards to $\tilde{\mathcal{A}}$ only the point-to-point and broadcast messages sent by a party P_j who did not receive $\perp$ from $\tilde{\mathcal{A}}$ in the previous (simulated) blank round of $\tilde{\Pi}$. Furthermore, $\mathcal{A}$ simulates for $\tilde{\mathcal{A}}$ the sending of all the $\perp$ messages that would be sent in the next blank round of $\tilde{\Pi}$ (if one exists), and halts.
  4. *Output:* $\mathcal{A}$ outputs whatever $\tilde{\mathcal{A}}$ does.

Before proceeding, we show that the only difference between an execution of Π with $\mathcal{A}$, and $\tilde{\Pi}$ with $\tilde{\mathcal{A}}$, is that some additional honest parties may output $\perp$ in the execution of $\tilde{\Pi}$. That is, we claim that the joint distribution of the outputs of all parties not outputting $\perp$ and the adversary, is identical in Π and $\tilde{\Pi}$. We begin with some notation:

- Let $\text{REAL}_{\Pi,(\mathcal{A},I)}(\overline{x},\overline{r})$ be the global output of an execution of Π with adversary $\mathcal{A}$, inputs $\overline{x}$, and random-tapes $\overline{r}$ (i.e., $\overline{r} = (r_1,\ldots,r_n)$ where P_i receives random-tape r_i). (Thus, $\text{REAL}_{\Pi,(\mathcal{A},I)}(\overline{x}) = \{\text{REAL}_{\Pi,(\mathcal{A},I)}(\overline{x},U_{|\overline{r}|})\}$.)
- For any subset $J \subseteq [n]$, denote by $\text{REAL}_{\Pi,(\mathcal{A},I)}(\overline{x},\overline{r})|_J$, the *restriction* of $\text{REAL}_{\Pi,(\mathcal{A},I)}(\overline{x},\overline{r})$ to the outputs of $\mathcal{A}$ and all parties P_j for $j \in J$. We stress that $\mathcal{A}$'s output is included in this restriction.
- In any execution of $\tilde{\Pi}$, it is possible to divide the parties into those who output $\perp$ and those who do not output $\perp$. We note that the set of parties outputting $\perp$ is chosen by the adversary $\tilde{\mathcal{A}}$ and is dependent on the parties' inputs $\overline{x}$ and random-tapes $\overline{r}$. We denote by $J = J_{\tilde{\mathcal{A}}}(\overline{x},\overline{r})$ the set of parties who do not output $\perp$ in an execution of $\tilde{\Pi}$ with adversary $\tilde{\mathcal{A}}$ (and where the inputs and random-tapes are $\overline{x}$ and $\overline{r}$). (Notice that $J_{\tilde{\mathcal{A}}}$ is a fixed function depending only on $\overline{x}$ and $\overline{r}$.) We also denote by $J_{\tilde{\mathcal{A}}}(\overline{x})$ a random variable taking values over $J_{\tilde{\mathcal{A}}}(\overline{x},\overline{r})$ for uniformly distributed $\overline{r}$.

We now consider the joint distribution of the outputs of the adversary and the parties in $J_{\tilde{\mathcal{A}}}(\overline{x},\overline{r})$ (i.e., those not outputting $\perp$). We claim that these distributions are identical in Π with $\mathcal{A}$ and in $\tilde{\Pi}$ with $\tilde{\mathcal{A}}$. Using the above notation, we claim that for every adversary $\tilde{\mathcal{A}}$ and set of corrupted parties I, and for all input and random-tape sequences $\overline{x}$ and $\overline{r}$,

$$\text{REAL}_{\Pi,(\mathcal{A},I)}(\overline{x},\overline{r})|_{J_{\tilde{\mathcal{A}}}(\overline{x},\overline{r})} = \text{REAL}_{\tilde{\Pi},(\tilde{\mathcal{A}},I)}(\overline{x},\overline{r})|_{J_{\tilde{\mathcal{A}}}(\overline{x},\overline{r})} \tag{3.1}$$

where $\mathcal{A}$ is as defined above. We now prove Eq. (3.1). First, it is clear that $\tilde{\mathcal{A}}$'s view in a real execution of $\tilde{\Pi}$ is identical to its view in the simulation by $\mathcal{A}$. Therefore, $\mathcal{A}$'s output in Π equals $\tilde{\mathcal{A}}$'s output in $\tilde{\Pi}$. Next, notice that if there exists an honest party that does not output $\perp$ in $\tilde{\Pi}$, then it must be that $\perp$ messages were sent in the last blank round only. However, this means that any honest party not receiving such a message has an identical view in Π and $\tilde{\Pi}$. Therefore, the outputs of all such parties are identical in Π and $\tilde{\Pi}$. Eq. (3.1) follows. We stress that the parties who output $\perp$ in $\tilde{\mathcal{A}}$ may have very different outputs in $\mathcal{A}$ (and in particular may output their prescribed outputs). Nevertheless, at this stage we are only interested in those parties not outputting $\perp$.

We now proceed to construct a simulator $\tilde{\mathcal{S}}$ for $\tilde{\mathcal{A}}$. Intuitively, $\tilde{\mathcal{S}}$ works by using the simulator $\mathcal{S}$ for $\mathcal{A}$, where $\mathcal{A}$ is derived from $\tilde{\mathcal{A}}$ as above. Recall that $\mathcal{A}$ is an adversary for the secure protocol Π, and thus a simulator $\mathcal{S}$ is guaranteed to exist for $\mathcal{A}$.

Simulator $\tilde{\mathcal{S}}$: First consider an adversary $\mathcal{A}$ for Π constructed from the adversary $\tilde{\mathcal{A}}$ as described above. By the security requirements of Π, for every adversary $\mathcal{A}$ there exists an ideal-model simulator $\mathcal{S}$ for $\mathcal{A}$. Simulator $\tilde{\mathcal{S}}$ for $\tilde{\mathcal{A}}$ works by emulating an ideal execution for $\mathcal{S}$. Recall that $\tilde{\mathcal{S}}$ works in an ideal model for secure computation with abort and no fairness, whereas $\mathcal{S}$ works in an ideal model for secure computation with unanimous abort and partial fairness. Further recall that where partial fairness holds, there are two cases depending on whether or not P_1 is corrupted. If P_1 is not corrupted, then essentially all parties receive output at the same time. If P_1 is corrupted, then the adversary receives the corrupted parties' outputs first and then decides whether the honest parties all receive output or all abort.

We now describe the simulator: $\tilde{\mathcal{S}}$ receives input series $\{x_i\}_{i \in I}$ and internally invokes $\mathcal{S}$ upon the same inputs $\{x_i\}_{i \in I}$. Then, $\tilde{\mathcal{S}}$ works as an intermediary between $\mathcal{S}$ and the trusted party. That is, $\tilde{\mathcal{S}}$ obtains the input values $\{x'_i\}_{i \in I}$ sent by $\mathcal{S}$ and externally sends these same values to the trusted party. Once $\tilde{\mathcal{S}}$ forwards these inputs to the trusted party, it receives back all the corrupted parties' outputs (recall that $\tilde{\mathcal{S}}$ interacts in an ideal model with no fairness). $\tilde{\mathcal{S}}$ then forwards these outputs to $\mathcal{S}$. We distinguish two cases:

1. P_1 *is not corrupted:* in this case, $\mathcal{S}$ concludes at this point, outputting some value.

2. P_1 *is corrupted:* in this case, $\mathcal{S}$ first instructs the trusted party to either send all the honest parties their outputs or send them all $\perp$. $\mathcal{S}$ then concludes, outputting some value.

$\tilde{\mathcal{S}}$ ignores the instruction sent to the trusted party in the second case, and sets its output to be whatever $\mathcal{S}$ output.

It remains to define the set J that $\tilde{\mathcal{S}}$ sends to the trusted party in the "trusted party answers remaining parties" stage of its ideal execution (recall that all honest parties in J receive their output and all others receive $\perp$). First notice that the string output by $\mathcal{S}$ is computationally indistinguishable to $\mathcal{A}$'s output from a real execution. However, by the definition of $\mathcal{A}$, this output contains $\tilde{\mathcal{A}}$'s view of an execution of $\tilde{\Pi}$. Furthermore, $\tilde{\mathcal{A}}$'s view fully defines which honest parties in an execution of $\tilde{\Pi}$ output $\perp$ and which receive their output. In particular, if $\tilde{\mathcal{A}}$ sent a $\perp$ message before the last blank round, then all honest parties abort (and $J = \phi$). Otherwise, all parties receive output except for those receiving $\perp$ messages in the last blank round. Therefore, $\tilde{\mathcal{S}}$ examines this view and defines the set J accordingly. Once J is defined, $\tilde{\mathcal{S}}$ sends it to the trusted party and halts. This completes the description of $\tilde{\mathcal{S}}$.

We now wish to show that the output of an ideal execution with abort and no fairness with adversary $\tilde{\mathcal{S}}$ is computationally indistinguishable to the output of a real execution of $\tilde{\Pi}$ with $\tilde{\mathcal{A}}$. We begin by showing an analog to Eq. (3.1) in the ideal model. That is, we show that the outputs of parties not outputting $\perp$ in an execution of $\tilde{\Pi}$ are the same in an ideal execution (by Def. 2) with $\mathcal{S}$ and in an ideal execution (by Def. 5) with $\tilde{\mathcal{S}}$. Formally, we claim the following: For every set I, every set of inputs $\overline{x}$ and every random-tape r (for $\mathcal{S}$ or $\tilde{\mathcal{S}}$),

$$\text{IDEAL}^{(5)}_{f,(\tilde{\mathcal{S}},I)}(\overline{x},r)|_{J_{\tilde{\mathcal{S}}}(\overline{x},r)} = \text{IDEAL}^{(2)}_{f,(\mathcal{S},I)}(\overline{x},r)|_{J_{\tilde{\mathcal{S}}}(\overline{x},r)} \tag{3.2}$$

where $\mathcal{S}$ and $\tilde{\mathcal{S}}$ are invoked with random-tape r, and where $J_{\tilde{\mathcal{S}}}(\overline{x},r)$ equals the set of parties in the ideal execution with $\tilde{\mathcal{S}}$ who do not output $\perp$. (I.e., $J_{\tilde{\mathcal{S}}}(\overline{x},r)$ equals the set J sent by $\tilde{\mathcal{S}}$ to the trusted party when the input vector equals $\overline{x}$ and its random-tape equals r.) In order to see that Eq. (3.2) holds, notice the following. First, $\tilde{\mathcal{S}}$ (upon input $\{x_i\}_{i\in I}$ and random-tape r) sends exactly the same inputs to the trusted party as $\mathcal{S}$ does (upon input $\{x_i\}_{i\in I}$ and random-tape r). Now, the outputs of all honest parties not outputting $\perp$ are fixed by $\overline{x}$ and the inputs sent to the trusted party by the simulators $\mathcal{S}$ or $\tilde{\mathcal{S}}$. Therefore, if $\mathcal{S}$ and $\tilde{\mathcal{S}}$ send the same inputs, it follows that all parties not outputting $\perp$ have exactly the same output. In addition, $\tilde{\mathcal{S}}$ outputs exactly the same string that $\mathcal{S}$ outputs. Eq. (3.2) therefore follows.

By assumption, Π is computationally t-secure with unanimous abort and partial fairness. It therefore holds that for every set $I \subset [n]$ such that $|I| < t$, and for *every set* $J \subseteq [n]$

$$\left\{\text{IDEAL}^{(2)}_{f,(\mathcal{S},I)}(\overline{x})|_J\right\} \stackrel{c}{\equiv} \left\{\text{REAL}_{\Pi,(\mathcal{A},I)}(\overline{x})|_J\right\}$$

Next, notice that the sets $J_{\tilde{\mathcal{S}}}$ and $J_{\tilde{\mathcal{A}}}$ are fully defined given $\tilde{\mathcal{A}}$ and $\tilde{\mathcal{S}}$'s outputs respectively. (Recall that $J_{\tilde{\mathcal{S}}}$ equals the set of parties not outputting $\perp$ in an ideal execution with $\tilde{\mathcal{S}}$, and $J_{\tilde{\mathcal{A}}}$ equals the set of parties not outputting $\perp$ in a real execution of $\tilde{\Pi}$ with $\tilde{\mathcal{A}}$.) Furthermore, by the definitions of $\mathcal{A}$ and $\mathcal{S}$, it follows that their outputs also fully define $J_{\tilde{\mathcal{S}}}$ and $J_{\tilde{\mathcal{A}}}$. Therefore, $J_{\tilde{\mathcal{S}}}$ (resp., $J_{\tilde{\mathcal{A}}}$) is part of the global output of IDEAL (resp., REAL). This implies that,

$$\left\{\text{IDEAL}^{(2)}_{f,(\mathcal{S},I)}(\overline{x})|_{J_{\tilde{\mathcal{S}}}(\overline{x})}\right\} \stackrel{c}{\equiv} \left\{\text{REAL}_{\Pi,(\mathcal{A},I)}(\overline{x})|_{J_{\tilde{\mathcal{A}}}(\overline{x})}\right\} \tag{3.3}$$

(Otherwise, we could distinguish $\text{IDEAL}^{(2)}_{f,(\mathcal{S},I)}(\overline{x})$ from $\text{REAL}_{\Pi,(\mathcal{A},I)}(\overline{x})$ by comparing the restriction to $J_{\tilde{\mathcal{S}}}$ or to $J_{\tilde{\mathcal{A}}}$, respectively.) Combining Eq. (3.3) with Equations (3.1) and (3.2), we have that

$$\left\{\text{IDEAL}^{(5)}_{f,(\tilde{\mathcal{S}},I)}(\overline{x})|_{J_{\tilde{\mathcal{S}}}(\overline{x})}\right\} \stackrel{c}{\equiv} \left\{\text{REAL}_{\tilde{\Pi},(\tilde{\mathcal{A}},I)}(\overline{x})|_{J_{\tilde{\mathcal{A}}}(\overline{x})}\right\}$$

It remains to show that the entire output distributions (including the honest parties *not* in J) are computationally indistinguishable. However, this is immediate, because for every party P_i for which $i \notin J$, it holds that P_i outputs $\perp$ (this is true for both the real and ideal executions). Therefore,

$$\left\{\text{IDEAL}^{(5)}_{f,(\tilde{\mathcal{S}},I)}(\overline{x})\right\} \stackrel{c}{\equiv} \left\{\text{REAL}_{\tilde{\Pi},(\tilde{\mathcal{A}},I)}(\overline{x})\right\}$$

completing the proof of Lemma 3.4.4. ■

Recall that our aim is to show the security of the compiled protocol Π' (and not $\tilde{\Pi}$). However, intuitively, there is no difference between $\tilde{\Pi}$ and Π'. The reason is as follows: in $\tilde{\Pi}$, the adversary can instruct any honest party P_i to halt by sending it $\perp$ in a blank round. On the other hand, in Π', the same effect can be achieved by having the "broadcast with abort" terminate with P_i receiving $\perp$. Therefore, whatever an adversary attacking Π' can achieve, an adversary attacking $\tilde{\Pi}$ can also achieve. Formally:

Lemma 3.4.5 *Let Π be a protocol in the broadcast model that is information-theoretic or computational t-secure with unanimous abort and with any level of fairness, and let $\tilde{\Pi}$ be the transformation of Π as described in Lemma 3.4.4. Then, for every real-model adversary $\mathcal{A}'$ for Π' of Construction 1, there exists a real-model adversary $\tilde{\mathcal{A}}$ for $\tilde{\Pi}$, such that for every $I \subset [n]$ with $|I| < t$,*

$$\left\{\text{REAL}_{\tilde{\Pi},(\tilde{\mathcal{A}},I)}(\overline{x})\right\} \equiv \left\{\text{REAL}_{\Pi',(\mathcal{A}',I)}(\overline{x})\right\}$$

Proof: We begin by describing the adversary $\tilde{\mathcal{A}}$. Intuitively, $\tilde{\mathcal{A}}$ works by simulating the executions of *broadcast with abort* for $\mathcal{A}'$. If a party receives $\perp$ in Π' (in a broadcast with abort execution or in a blank round), then $\tilde{\mathcal{A}}$ sends the appropriate $\perp$ messages in a blank round of $\tilde{\Pi}$. This strategy works because in Π', there is no difference if a party receives $\perp$ in a broadcast with abort execution or in the following blank round. Formally, adversary $\tilde{\mathcal{A}}$ invokes $\mathcal{A}'$ and in every round of the execution of $\tilde{\Pi}$ works as follows:

1. *Receiving messages in rounds $r, r+1$:* $\tilde{\mathcal{A}}$ receives the broadcast and point-to-point messages from the honest parties in $\tilde{\Pi}$. For every message broadcast by an honest party, $\tilde{\mathcal{A}}$ simulates a "broadcast with abort" execution, playing the honest parties' roles (where $\mathcal{A}'$ plays the corrupted parties' roles). In addition, $\tilde{\mathcal{A}}$ forwards any point-to-point messages unchanged to $\mathcal{A}'$.

2. *Sending messages in round $r, r+1$:* $\tilde{\mathcal{A}}$ plays the honest parties' roles in "broadcast with abort" executions, in which $\mathcal{A}'$ broadcasts messages to the honest parties. Consider a particular execution in which a corrupted party P plays the dealer. If all the honest parties receive $\perp$ in this execution, then $\tilde{\mathcal{A}}$ broadcasts nothing in $\tilde{\Pi}$. On the other hand, if at least one honest party outputs a message m, then $\tilde{\mathcal{A}}$ broadcasts m. As before, point-to-point messages are forwarded unchanged.

3. *Blank round following round $r+1$:* Let $\mathcal{P}$ denote the set of honest parties receiving $\perp$ in any of the above simulated "broadcast with abort" executions (i.e., for the broadcast of rounds r and $r+1$). Then, for every $P_i \in \mathcal{P}$, adversary $\tilde{\mathcal{A}}$ sends $\perp$ to P_i in this blank round.

At the conclusion of the execution, $\tilde{\mathcal{A}}$ outputs whatever $\mathcal{A}'$ does. This completes the description of $\tilde{\mathcal{A}}$. The fact that $\tilde{\mathcal{A}}$ perfectly simulates an execution of Π' with $\mathcal{A}'$ follows directly from the definition of Π'. That is, the only difference between $\tilde{\Pi}$ and Π' is that in Π' the broadcast channel is replaced by broadcast with abort. This means that some parties may receive $\perp$ instead of the broadcasted message. However, in this case, $\tilde{\mathcal{A}}$ knows exactly who these parties are and can send them $\perp$ in the following blank round. The key point is that in Π' it *makes no difference* if $\perp$ is received in a broadcast with abort execution or in the following blank round. We conclude that the outputs of all the honest parties and $\tilde{\mathcal{A}}$ in $\tilde{\Pi}$, are identically distributed to the outputs of the honest parties and $\mathcal{A}'$ in Π'. ■

Concluding the proof of Theorem 3.4.1: Let $\mathcal{A}'$ be an adversary attacking Π'. By Lemma 3.4.5, we have that there exists an adversary $\tilde{\mathcal{A}}$ attacking $\tilde{\Pi}$ such that the output distributions of Π' with $\mathcal{A}'$, and $\tilde{\Pi}$ with $\tilde{\mathcal{A}}$ are identical. Then, by Lemma 3.4.4, we have that for real-model adversary $\tilde{\mathcal{A}}$ for $\tilde{\Pi}$, there exists an ideal-model simulator $\tilde{\mathcal{S}}$ such that the output distributions of a real execution with $\tilde{\mathcal{A}}$ and an ideal execution (with abort and no fairness) with $\tilde{\mathcal{S}}$ are computationally indistinguishable (or statistically close). We conclude that the output distribution of a real execution of Π'

with adversary $\mathcal{A}'$ is computationally indistinguishable (or statistically close) to an ideal execution (with abort and no fairness) with $\tilde{\mathcal{S}}$. That is, Π' is t-secure with abort and no fairness, as required. ■

The complexity of protocol Π': We remark that the transformation of Π to Π' preserves the round complexity of Π. In particular, the number of rounds in Π' equals exactly 3 times the number of rounds in Π. On the other hand, the bandwidth of Π' is the same as that of Π except for the cost incurred in simulating the broadcast channel. Notice that in the "broadcast with abort" protocol, if a dealer sends a k-bit message, then the total bandwidth equals $n \cdot k$. (If the dealer cheats and sends different messages, then the bandwidth is upper-bound by the length of the longest message times n.) Therefore, the number of bits sent in an execution of Π' is only n times that sent in an execution of Π.

3.5 Secure Computation with Abort and Partial Fairness

In this section we show that for any functionality f, there exists a protocol for the *computational* t-secure computation of f with abort and partial fairness, for any t. (This construction assumes the existence of trapdoor permutations.) Furthermore, for any functionality f, there exists a protocol for *information-theoretic* $n/2$-secure computation of f with abort and partial fairness (and without any complexity assumptions).

Outline: We begin by motivating why the strategy used to obtain secure computation with abort and no fairness is not enough here. The problem lies in the fact that due to the use of a "broadcast with abort" protocol (and not a real broadcast channel), the adversary can disrupt communication between honest parties. That is, of course, unless this communication need not be broadcast. Now, in the definition of security with abort and partial fairness, once an honest P_1 receives its output, it must be able to give this output to all honest parties. That is, the adversary must not be allowed to disrupt the communication, following the time that an honest P_1 receives its output. This means that using a "broadcast with abort" protocol in the final stage where the remaining parties receive their outputs is problematic.

We solve this problem here by having the parties compute a different functionality. This functionality is such that when P_1 gets its output, it can supply all the other parties with their output directly and without broadcast. On the other hand, P_1 itself should learn nothing of the other parties' outputs. As a first attempt, consider what happens if instead of computing the original functionality f, the parties first compute the following:

First attempt:

Inputs: $\overline{x} = (x_1, \ldots, x_n)$

Outputs:

- Party P_1 receives its own output $f_1(\overline{x})$. In addition, for every $i > 1$, it receives $c_i = f_i(\overline{x}) \oplus r_i$ for a uniformly distributed r_i.
- For every $i > 1$, party P_i receives the string r_i.

That is, for each $i > 1$, party P_i receives a random pad r_i and P_1 receives an encryption of $f_i(\overline{x})$ with that random pad. Now, assume that the parties use a protocol that is secure with abort and *no* fairness in order to compute this new functionality. Then, there are two possible outcomes to such a protocol execution: either all parties receive their prescribed output, or at least one honest party receives $\perp$. In the case that at least one honest party receives $\perp$, this party can notify P_1 who can then immediately halt. The result is that no parties, including the corrupted ones, receive output (if P_1 does not send the c_i values, then the parties only obtain r_i which contains no information on $f_i(\overline{x})$). In contrast, if all parties received their prescribed output, then party P_1 can send each party P_i its encryption c_i, allowing it to reconstruct its output $f_i(\overline{x})$. The key point is that the adversary is unable to prevent P_1 from sending these c_i values and no broadcast is needed in this last step. Of course, if P_1 is corrupted, then it will learn all the corrupted parties' outputs first. However, under the definition of partial fairness, this is allowed.

The flaw in the above strategy arises in the case that P_1 is corrupted. Specifically, a corrupted P_1 can send the honest parties modified values, causing them to conclude with incorrect outputs. This is in contradiction to what is required of all secure protocols. Therefore, we modify the functionality that is computed so that a corrupted P_1 is unable to cheat. In particular, the aim is to prevent the adversary from modifying $c_i = f_i(\overline{x}) \oplus r_i$ without P_i detecting this modification. If the adversary can be restrained in this way, then it can choose not to deliver an output; however, any output delivered is guaranteed to be correct. The above-described aim can be achieved using standard (information-theoretic) authentication techniques, based on pairwise independent hash functions. That is, let $\mathcal{H}$ be a family of pairwise independent hash functions $h : \{0,1\}^k \rightarrow \{0,1\}^k$. Then, the functionality that the parties compute is as follows:

Functionality F:

Inputs: $\overline{x} = (x_1, \ldots, x_n)$

Outputs:

- Party P_1 receives its own output $f_1(\overline{x})$. In addition, for every $i > 1$, it receives $c_i = f_i(\overline{x}) \oplus r_i$ for a uniformly distributed r_i, and $a_i = h_i(c_i)$ for $h_i \in_R \mathcal{H}$.
- For every $i > 1$, party P_i receives the string r_i and the description of the hash function h_i.

Notice that as in the first attempt, P_1 learns nothing of the output of any honest P_i (since $f_i(\overline{x})$ is encrypted with a random pad). Furthermore, if P_1 attempts to modify c_i to c_i' in any way, then the probability that it will generate the correct authentication value $a_i' = h_i(c_i')$ is at most 2^{-k} (by the pairwise independent properties of h_i). Thus, the only thing a corrupt P_1 can do is refuse to deliver the output. The protocol for computing f, as motivated here, appears in Figure 3.2.

We now proceed to show that Protocol 3.2 achieves security with abort and *partial fairness.*

Theorem 3.5.1 (Theorem 3.1.2 – restated): *For any probabilistic polynomial-time n-ary functionality f, there exists a protocol in the point-to-point model for the computational t-secure computation of f* with abort and partial fairness, *for any t. Furthermore, there exists a protocol in the point-to-point model for the information-theoretic $n/2$-secure computation of f* with abort and partial fairness.

Proof: The protocol referred to in the theorem statement is that of Protocol 3.2, where the subprotocol used for Stage 1 is computationally t-secure (for the first part of the theorem) or

[9]By Corollaries 3.4.2 and 3.4.3 in Section 3.4, such protocols exist for any t assuming the existence of trapdoor permutations. Furthermore, for the case of $t > n/2$, no assumptions are required.

Protocol 3.2

The parties execute the following three steps:

1. *Stage* 1 – *computation:* The parties use any protocol for secure (computational or information-theoretic) computation with abort and *no fairness* in order to compute the functionality F defined above.[9]Thus, P_1 receives $f_1(\overline{x})$ and a pair (c_i, a_i) for every $i > 1$, and each P_i $(i > 1)$ receives (r_i, h_i) such that $c_i = f_i(\overline{x}) \oplus r_i$ and $a_i = h_i(c_i)$.

2. *Stage* 2 – *blank round:* After the above protocol concludes, a blank-round is added so that if any party receives $\perp$ for its output from Stage 1, then it sends $\perp$ to P_1 in this blank round.

3. *Stage* 3 – *outputs:* If P_1 received any $\perp$-messages in the blank round, then it sends $\perp$ to all parties and halts outputting $\perp$. Otherwise, for every i, it sends (c_i, a_i) to P_i and halts, outputting $f_1(\overline{x})$.

   Party P_i outputs $\perp$ if it received $\perp$ from P_1 (it ignores any $\perp$ it may receive from other parties). If it received (c_i, a_i) from P_1 (and not $\perp$), then it checks that $a_i = h_i(c_i)$. If yes, it outputs $f_i(\overline{x}) = c_i \oplus r_i$. Otherwise, it outputs $\perp$.

Figure 3.2: A protocol for secure computation with abort and partial fairness for any f

information-theoretically $n/2$-secure (for the second part of the theorem). Intuitively, the security of Protocol 3.2 is derived from the fact that Stage 1 is run using a protocol that is secure with abort (even though it has no fairness property). Consider the two cases regarding whether or not P_1 is corrupted:

1. P_1 *is corrupt:* in this case, $\mathcal{A}$ receives all the outputs of the corrupted parties first. Furthermore, $\mathcal{A}$ can decide exactly which parties to give output to and which not. However, this is allowed in the setting of secure computation with abort and partial fairness, and so is fine. We stress that $\mathcal{A}$ cannot cause an honest party to output any value apart from $\perp$ or its correct output. This is because the authentication properties of pairwise independent hash functions guarantee that $\mathcal{A}$ does not modify c_i, and the correctness of the protocol of Stage 1 guarantees that $c_i \oplus r_i$ equals the correct output $f_i(\overline{x})$.

2. P_1 *is honest:* there are two possible cases here; either some honest party received $\perp$ in the computation of Stage 1 or all honest parties received their correct outputs. If some honest party received $\perp$, then this party sends $\perp$ to P_1 in Stage 2 and thus no parties (including the corrupted parties) receive output. (Similarly, if $\mathcal{A}$ sends $\perp$ to P_1 in Stage 2 then no parties receive output.) On the other hand, if all honest parties received their outputs and $\mathcal{A}$ does not send $\perp$ to P_1 in Stage 2, then all parties receive outputs and the adversary cannot cause any honest party to abort. We therefore have that in this case complete fairness is achieved, as required.

In order to formally prove the security of the protocol, we use the sequential composition theorem of Canetti [13]. This theorem states that we can consider a hybrid model in which an ideal call is used for Stage 1 of the protocol, whereas the other stages are as described above. That is, the parties all interact with a trusted party for the computation of Stage 1 (where the ideal model for this computation is that of secure computation with abort and no fairness). Then, Stages 2 and 3 take place as in a real execution. The result is a protocol that is a hybrid of real and ideal executions. In order to prove the security of the (real) protocol, it suffices to construct an ideal-model simulator for the hybrid protocol. Thus, the description of the parties and adversary below relate to this hybrid model (the parties send messages to each other, as in a real execution, and to a trusted party, as in an ideal execution). The proof of [13] is stated for secure computation without abort (and complete fairness); however it holds also for secure computation with abort and no fairness.

Let $\mathcal{A}$ be an adversary attacking Protocol 3.2 in the above-described hybrid model. Notice that in the ideal call of Stage 1, $\mathcal{A}$ receives all the corrupted parties' outputs first and then decides which honest parties receive output (this is because there is no fairness in the computation of Stage 1). We now construct an ideal-model adversary $\mathcal{S}$ who works in an ideal model with abort and partial fairness. We stress that $\mathcal{A}$ works in a hybrid model in which the "ideal model" part has no fairness; whereas, $\mathcal{S}$ works in an ideal model with partial fairness. $\mathcal{S}$ has *external* communication with the trusted party of its ideal model and *internal*, simulated communication with the adversary $\mathcal{A}$. In our description of $\mathcal{S}$, we differentiate between the cases that P_1 is corrupt and P_1 is honest:

1. P_1 *is corrupt:* $\mathcal{S}$ invokes $\mathcal{A}$ and receives the inputs that $\mathcal{A}$ intends to send to the trusted party of the hybrid model. Then, $\mathcal{S}$ externally sends these inputs unmodified to the trusted party of its ideal model for computing f. If the inputs are not valid, then in the hybrid model all parties receive $\perp$ as output. Therefore, $\mathcal{S}$ internally hands $\perp$ to $\mathcal{A}$ as its output from Stage 1 and simulates all honest parties sending $\perp$ in Stage 2 (as would occur in a hybrid execution). $\mathcal{S}$ then halts, outputting whatever $\mathcal{A}$ does. Otherwise, if the inputs are valid, $\mathcal{S}$ receives all the corrupted parties outputs $\{f_i(\overline{x})\}_{i\in I}$ (this is the case because $\mathcal{S}$ controls P_1 and by partial fairness, when P_1 is corrupt the adversary receives the corrupted parties' outputs first). $\mathcal{S}$ then constructs the corrupted parties' outputs from Stage 1 that $\mathcal{A}$ expects to see in the hybrid execution. $\mathcal{S}$ defines P_1's output as follows: First, P_1's personal output is $f_1(\overline{x})$. Next, for every corrupted party P_i, party P_1's output contains the pair $(c_i = f_i(\overline{x}) \oplus r_i, h_i(c_i))$ for $r_i \in_R \{0,1\}^k$ and $h_i \in_R \mathcal{H}$. Finally, for every honest party P_j, party P_1's output contains a pair (c_j, a_j) where $c_j, a_j \in_R \{0,1\}^k$. This defines P_1's output. We now define how $\mathcal{S}$ constructs the other corrupted parties' outputs: for every corrupted P_i, simulator $\mathcal{S}$ defines P_i's output to equal (r_i, h_i) where these are the values used in preparing the corresponding pair $(c_i, h_i(c_i))$ in P_1's output. (We note that $\mathcal{S}$ can prepare these values because it knows $f_i(\overline{x})$ for every corrupted party P_i.) $\mathcal{S}$ then internally passes $\mathcal{A}$ all of these outputs. In the hybrid model, after receiving the outputs from Stage 1, $\mathcal{A}$ sends a set J' to the trusted party instructing it to give outputs to the parties specified in this set (all other parties receive $\perp$). $\mathcal{S}$ obtains this set J' from $\mathcal{A}$ and records it.

   $\mathcal{S}$ continues by simulating P_l sending $\perp$ to P_1 in the blank round, for every honest party P_l for which $l \notin J'$ (as would occur in a hybrid execution). Then, in the last stage, $\mathcal{A}$ (controlling P_1) sends to each honest party P_j a pair (c'_j, a'_j) or $\perp$. $\mathcal{S}$ receives these strings and defines the set of parties J to receive outputs to equal those parties in J' to whom $\mathcal{A}$ sends the same (c_j, a_j) that $\mathcal{S}$ gave $\mathcal{A}$ in Stage 1. (These are the parties who do not see $\perp$ in the execution and whose checks of Stage 3 succeed; they therefore do not abort.) $\mathcal{S}$ concludes by externally sending J to the ideal-model trusted party and outputting whatever $\mathcal{A}$ does.

2. P_1 *is honest:* In this case, $\mathcal{S}$ begins in the same way. That is, $\mathcal{S}$ invokes $\mathcal{A}$ and receives the inputs that $\mathcal{A}$ intends to send the trusted party of the hybrid model. However, unlike in the previous case, $\mathcal{S}$ does not forward these inputs to its trusted party; rather it just records them.[10] (If any of these inputs are invalid, then $\mathcal{S}$ internally sends $\perp$ to all corrupted parties,

[10] $\mathcal{S}$ cannot forward the inputs to the trusted party yet, because in the model of partial fairness as soon as it does this all parties receive output. However, in the execution of Protocol 3.2, $\mathcal{A}$ can cause the execution to abort at a later stage.

externally sends invalid inputs to the trusted party and halts. In the sequel, we assume that all inputs sent by $\mathcal{A}$ are valid.) Now, $\mathcal{A}$ expects to receive outputs from Stage 1 before it sends the trusted party the set J' of honest parties receive output from Stage 1. However, $\mathcal{S}$ does *not* have the corrupted parties' outputs yet. Fortunately, when P_1 is honest, $\mathcal{S}$ can perfectly simulate the corrupted parties outputs from Stage 1 by merely providing them with (r_i, h_i) where $r_i \in_R \{0,1\}^k$ and $h_i \in_R \mathcal{H}$. After internally passing $\mathcal{A}$ the simulated corrupted parties' outputs, $\mathcal{S}$ obtains a set J' from $\mathcal{A}$, instructing the trusted party of the hybrid model which parties should receive output.

$\mathcal{S}$ continues by simulating Stages 2 and 3 of the protocol. As above, $\mathcal{S}$ simulates every honest party P_l for which $l \notin J'$ sending $\perp$ to P_1 in Stage 2. Furthermore, $\mathcal{S}$ obtains any messages sent by $\mathcal{A}$ in this stage. If $\mathcal{A}$ sends $\perp$ to P_1 in Stage 2, then $\mathcal{S}$ simulates P_1 sending $\perp$ to all parties, sends invalid inputs to the trusted party and halts. Likewise, if J' does not contain *all* the honest parties, then $\mathcal{S}$ internally simulates P_1 sending $\perp$ to all the corrupted parties, and externally sends invalid inputs to the trusted party. (These cases correspond to the case that no parties receive their prescribed output.)

In contrast, if J' contains all the honest parties (i.e., no honest party received $\perp$ from Stage 1) and $\mathcal{A}$ did not send $\perp$ to P_1 in Stage 2 of the simulation, then $\mathcal{S}$ externally sends the trusted party the inputs that it recorded from $\mathcal{A}$ above, receiving back all of the corrupted parties outputs $\{f_i(\overline{x})\}_{i \in I}$. Then, for each corrupted party's output $f_i(\overline{x})$, simulator $\mathcal{S}$ generates the pair that corrupted P_i would see in a hybrid execution. In particular, previously in the simulation $\mathcal{S}$ provided P_i with a pair (r_i, h_i) where $r_i \in_R \{0,1\}^k$ and $h_i \in_R \mathcal{H}$. Now, $\mathcal{S}$ simulates P_1 sending corrupted party P_i the pair (c_i, a_i) where $c_i = f_i(\overline{x}) \oplus r_i$ and $a_i = h_i(c_i)$. ($\mathcal{S}$ can do this because it knows the random-pad r_i and the hash function h_i.) Finally, $\mathcal{S}$ outputs whatever $\mathcal{A}$ does and halts.

The fact that the global output in the hybrid execution with $\mathcal{S}$ is identically distributed to the global output in a real execution with $\mathcal{A}$ is derived from the following observations. First, $\mathcal{A}$'s outputs from Stage 1 can be perfectly simulated, both when P_1 is corrupt and when P_1 is honest. Second, the honest parties' messages in Stage 2 can be perfectly simulated given only the set J' sent by $\mathcal{A}$ to the hybrid-model trusted party in the ideal execution of Stage 1. Therefore, $\mathcal{A}$'s view in the hybrid-model execution is identical to its view in a real execution. It remains to show that the honest parties' outputs are also correctly distributed.

First, consider the case that P_1 is corrupt. In this case, with overwhelming probability, the set of honest parties receiving output in the real model are exactly those parties P_j for whom P_1 (controlled by $\mathcal{A}$) sends the exact pair (c_j, a_j) that it received as output from Stage 1 (and who did not see $\perp$ at any time in the execution). This is due to the authentication properties of pairwise independent hash functions. Likewise, in the ideal-model simulation, $\mathcal{S}$ designates these same parties to be the ones receiving output. Therefore, except with negligible probability, the set J sent by $\mathcal{S}$ to the trusted party contains exactly those parties who would receive output in a real execution.

Next, consider the case that P_1 is honest. In this case, all parties receive output unless P_1 sees $\perp$ in Stage 2. This can happen if $\mathcal{A}$ sends P_1 such a value, or if any honest party received $\perp$ from Stage 1. Both of these cases are detected by $\mathcal{S}$ in the hybrid-model simulation, and therefore the case that all parties abort in the hybrid model corresponds to this case in the real model (and likewise for the case that all parties receive output). This completes the proof of Theorem 3.5.1 ■

Chapter 4

Universally Composable Multi-Party Computation

In this chapter we prove a fundamental result stating that secure multi-party computation that remains secure under *concurrent general-composition* can be achieved, for any number of corrupted parties. That is, we consider an asynchronous multi-party network and an adversary that can adaptively corrupt as many parties as it wishes. We present protocols that allow any subset of parties in this setting to securely realize any desired functionality of their inputs, and be guaranteed that security is preserved *regardless of the activity in the rest of the network.* Our protocols are in the common reference string model and rely on standard intractability assumptions.

4.1 Introduction

As we have seen, the model of "stand-alone computation" does not fully capture the security requirements from cryptographic protocols in a modern computer network. In such networks, a protocol execution may run concurrently with an unknown number of other protocols. These arbitrary protocols may be executed by the same parties or other parties, they may have potentially related inputs and the scheduling of message delivery may be adversarially coordinated. Furthermore, the local outputs of a protocol execution may be used by other protocols in an unpredictable way. These concerns, or "attacks" on a protocol are not captured by the stand-alone model.

One way to guarantee that protocols withstand some specific security threats in multi-execution environments is to explicitly incorporate these threats into the security model and analysis. Such an approach was taken, for instance, in the case of non-malleability of protocols [32], and regarding the concurrent composition of zero-knowledge [33, 77] and oblivious transfer [43]. However, this approach is inherently limited since it needs to explicitly address each new concern, whereas in a realistic network setting, the threats may be unpredictable. Furthermore, it inevitably results in definitions with ever-growing complexity.

In contrast, we take the approach where a protocol is designed and analyzed as "stand alone", and security in a multi-execution environment is guaranteed via a *secure composition theorem.* In particular, we use the recently proposed framework of universally composable security [14]. Here a generic definition is given for what it means for a protocol to "securely realize a given ideal functionality", where an "ideal functionality" is a natural algorithmic way of capturing the desired functionality of the protocol problem at hand. In addition, it is shown that security of protocols is preserved under a general composition operation called universal composition. This essentially means that any protocol that securely realizes an ideal functionality when considered as stand-alone, continues to securely realize the same functionality even when composed with any other set of protocols that may be running concurrently in the same system. (In other words, the protocol remains secure under concurrent general-composition.) A protocol that is secure within the [14] framework is called universally composable (UC).

It has been shown that *any* ideal functionality can be securely realized in a universally composable way using known constructions, as long as a majority of the participants remain uncorrupted [14] (building upon [10, 76, 15]). However, this result does not hold when half or more of the parties may be corrupted. In particular, it does not hold for the important case of *two-party* protocols, where each party wishes to maintain its security even if the other party is corrupted. In fact, it was shown in [16, 14] that in the standard model, a number of basic two-party functionalities (such as commitment, zero-knowledge, and common coin-tossing) cannot be securely realized in this framework by two-party protocols. Nonetheless, protocols that securely realize the commitment and zero-knowledge functionalities in the common reference string (CRS) model were shown in [16, 26]. (In the CRS model all parties are given a common, public reference string that is ideally chosen from a given distribution. This model was originally proposed in the context of non-interactive zero-knowledge proofs [12] and since then has proved useful in other cases as well.)

Our results. Loosely speaking, we show that any functionality can be realized in a universally composable way, in the CRS model, regardless of the number of corrupted parties. More specifically, consider an *asynchronous* multi-party network where the communication is open and delivery of messages is not guaranteed. (For simplicity, we assume that delivered messages are authenticated. This can be achieved using standard methods.) The network contains an unspecified number of parties, and any number of these parties can be adaptively corrupted throughout the computation. In this setting, we show how arbitrary subsets of parties can securely realize any functionality of their inputs in a universally composable way. The functionality may be reactive, namely it may receive inputs and generate outputs multiple times throughout the computation.

In addition to our general constructions for two-party and multi-party computation, we also present a new adaptively secure UC commitment scheme in the CRS model, assuming only the existence of trapdoor permutations. (UC commitment schemes are protocols that securely realize the ideal commitment functionality [16]. Existing constructions of UC commitments [16, 25] rely on specific cryptographic assumptions.) Since UC zero-knowledge can be obtained given a UC commitment scheme [16], we can plug our new scheme into the UC zero-knowledge protocol of [16] and thereby obtain an adaptively secure UC zero-knowledge protocol in the CRS model, for any NP relation, and based on any trapdoor permutation. Beyond being interesting in its own right, we use this commitment scheme in order to base our constructions on general cryptographic assumptions.

Adaptive security. Our protocol is the first general construction that guarantees security against adaptive adversaries in the two-party case and in the case of multi-party protocols with honest minority. (We note that no adaptively secure general construction was known in these cases even in the traditional stand-alone model; all previous adaptively secure protocols for general multi-party computation assumed an honest majority.) We remark that, in contrast to the case of stand-alone protocols, in our setting adaptive security is a relevant concern even for protocols with only two participants. Furthermore, it is important to protect even against adversaries that eventually break into *all* the participants in an interaction. This is because we consider multiple interactions that take place between different sets of parties in the system. Therefore, all the participants in one interaction may constitute a proper subset of the participants in another interaction. Our results hold even in a model where no data can ever be erased.

Cryptographic assumptions. Our protocols are based on the following cryptographic assumptions. For the non-adaptive case (both semi-honest and malicious) we assume the existence of trapdoor permutations only. For the adaptive case we additionally assume the existence of *augmented* non-committing encryption protocols [15]. The augmentation includes oblivious key generation and invertible samplability [24]. Loosely speaking, oblivious key generation states that public keys can be generated without knowing the corresponding private keys, and invertible samplability states that given a public/private key-pair it is possible to obtain the random coin tosses of the key generator when outputting this key-pair (the oblivious key generator should also be invertible). Such encryption schemes are known to exist under the RSA and DDH assumptions.

As we have mentioned, our protocols are in the CRS model. The above assumptions suffice if we use a common reference string that is not uniformly distributed (but is rather taken from some different distribution). If a uniformly distributed common reference string is to be used, then we additionally assume the existence of dense cryptosystems [27].

Related work. In a concurrent and independent work [25], Damgard and Nielsen consider a functionality that has great resemblance to our commit-and-prove functionality, and construct universally composable protocols that realize this functionality under specific number-theoretic assumptions. Our commit-and-prove protocol is based on more general assumptions, whereas their protocol is considerably more efficient.

Organization. In Section 4.2 we provide an overview of the model of [14] and an outline of our construction of UC two-party and multi-party protocols. Section 4.3 contains a number of important preliminaries: First, in Section 4.3.1, a more detailed description of the [14] framework and of the composition theorem itself is presented. Then, in Section 4.3.2, the issue of universal composition and joint state is discussed (this is important where a common reference string is used, as is the case in our constructions). Finally, in Section 4.6, the ideal zero-knowledge functionality is described. Our new UC commitment scheme is presented in Section 4.5.

We then begin our presentation of constructions for UC two-party and multi-party computation. That is, in Section 4.4, we show how to obtain UC two-party secure computation in the presence of semi-honest adversaries. Next we proceed to the case of malicious adversaries. That is, in Section 4.7 we define and realize the two-party commit-and-prove functionality. This is then used in Section 4.8 to construct a two-party protocol compiler that is used to transform the protocol of Section 4.4 into a protocol that is secure against malicious adversaries. Finally, in Section 4.9, we extend our two-party constructions to the multi-party case. We present the two-party case separately because most of the cryptographic ideas already arise in this setting.

4.2 Overview

This section provides a high-level overview of the model and our constructions. Section 4.2.1 contains an overview of the general framework of universal composability, the definition of security and the composition theorem. Then, in Section 4.2.2 we provide a brief outline of our constructions for two-party and multi-party computation. The aim of this outline is to provide the reader with the "big picture", before delving into details.

4.2.1 The model

We begin by outlining the framework for universal composability; for more details see Section 4.3.1 and [14]. The framework provides a rigorous method for defining the security of cryptographic tasks, while ensuring that security is maintained under a general composition operation in which a secure protocol for the task in question is run in a system concurrently with an unbounded number of other arbitrary protocols. This composition operation is called universal composition, and tasks that fulfill the definitions of security in this framework are called universally composable (UC).

As in other general definitions (e.g., [53, 69, 5, 73, 13]), the security requirements of a given task (i.e., the functionality expected from a protocol that carries out the task) are captured via a set of instructions for a "trusted party" that obtains the inputs of the participants and provides them with the desired outputs (in one or more iterations). We call the algorithm run by the trusted party an ideal functionality. Informally, a protocol securely carries out a given task if any adversary can gain nothing more from an attack on a real execution of the protocol, than from an attack on an ideal process where the parties merely hand their inputs to a trusted party with the appropriate functionality and obtain their outputs from it, without any other interaction. In other words, we require that a real execution can be *emulated* in the above ideal process (where the meaning of *emulation* is described below). We stress that in a real execution of the protocol, no trusted party exists and the parties interact amongst themselves only.

In order to prove the universal composition theorem, the notion of emulation in this framework is considerably stronger than in previous ones. Traditionally, the model of computation includes the parties running the protocol, plus an adversary $\mathcal{A}$ that controls the communication channels and potentially corrupts parties. Emulation means that for any adversary $\mathcal{A}$ attacking a real protocol execution, there should exist an "ideal process adversary" or simulator $\mathcal{S}$, that causes the outputs of the parties in the ideal process to be essentially the same as the outputs of the parties in a real execution. In the universally composable framework, an additional adversarial entity called the environment $\mathcal{Z}$ is introduced. This environment generates the inputs to all parties, reads all outputs, and in addition interacts with the adversary in an arbitrary way throughout the computation. (As is hinted by its name, $\mathcal{Z}$ represents the external environment that consists of arbitrary protocol executions that may be running concurrently with the given protocol.) A protocol is said to securely realize a given ideal functionality $\mathcal{F}$ if for any "real-life" adversary $\mathcal{A}$ that interacts with the protocol there exists an "ideal-process adversary" $\mathcal{S}$, such that *no environment* $\mathcal{Z}$ can tell whether it is interacting with $\mathcal{A}$ and parties running the protocol, or with $\mathcal{S}$ and parties that interact with $\mathcal{F}$ in the ideal process. (In a sense, here $\mathcal{Z}$ serves as an "interactive distinguisher" between a run of the protocol and the ideal process with access to $\mathcal{F}$. See [14] for more motivating discussion on the role of the environment.) Note that the definition requires the "ideal-process adversary" (or simulator) $\mathcal{S}$ to interact with $\mathcal{Z}$ throughout the computation. Furthermore, $\mathcal{Z}$ cannot be "rewound".

The following *universal composition theorem* is proven in [14]: Consider a protocol π that operates in a *hybrid* model of computation where parties can communicate as usual, and in addition have ideal access to an unbounded number of copies of some ideal functionality $\mathcal{F}$. (This model is called the $\mathcal{F}$-hybrid model.) Furthermore, let ρ be a protocol that securely realizes $\mathcal{F}$ as sketched above, and let π^ρ be the "composed protocol". That is, π^ρ is identical to π with the exception that each interaction with the ideal functionality $\mathcal{F}$ is replaced with a call to (or an activation of) an appropriate instance of the protocol ρ. Similarly, ρ-outputs are treated as values provided by the functionality $\mathcal{F}$. The theorem states that in such a case, π and π^ρ have essentially the same input/output behavior. Thus, ρ behaves just like the ideal functionality $\mathcal{F}$, even when composed with an arbitrary protocol π. A special case of this theorem states that if π securely realizes some ideal functionality $\mathcal{G}$ in the $\mathcal{F}$-hybrid model, then π^ρ securely realizes $\mathcal{G}$ from scratch.

We consider a network where the adversary sees all the messages sent, and delivers or blocks these messages at will. (The fact that message delivery is not guaranteed frees us from the need to explicitly deal with the "early stopping" problem of protocols run between two parties or amongst many parties where only a minority may be honest. This is because even the ideal process allows the adversary to abort the execution at any time.) We note that although the adversary may block messages, it cannot modify messages sent by honest parties (i.e., the communication lines are ideally authenticated). Our protocols are cast in a completely asynchronous point-to-point network (and thus the adversary has full control over when messages are delivered, if at all). Also, as usual, the adversary is allowed to corrupt parties. In the case of static adversaries the set of corrupted parties is fixed at the onset of the computation. In the adaptive case the adversary corrupts parties at will throughout the computation. We also distinguish between malicious and semi-honest adversaries: If the adversary is malicious then corrupted parties follow the arbitrary instructions of the adversary. In the semi-honest case, even corrupted parties follow the prescribed protocol and the adversary essentially only gets read access to the states of corrupted parties.

4.2.2 An outline of the results and techniques

In this section we provide a high-level description of our protocols for two-party and multi-party computation, and the techniques used in obtaining them. Our construction is conceptually very similar to the construction of Goldreich, Micali and Wigderson [52, 44]. This construction (which we call the GMW construction) is comprised of two stages. First, they present a protocol for securely realizing any functionality in the semi-honest adversarial model. Next, they construct a *protocol compiler* that takes any semi-honest protocol and transforms it into a protocol that has the same functionality in the malicious adversarial model. (However, as discussed above, they consider a model where only a single protocol execution takes place in the system. In contrast, we construct protocols for universally composable secure computation.) We begin by considering the two-party case.

Two-party computation in the case of semi-honest adversaries

Recall that in the case of semi-honest adversaries, even the corrupted parties follow the protocol specification. However, the adversary may attempt to learn more information than intended by examining the transcript of messages that it received during the protocol execution. Despite the seemingly weak nature of the adversarial model, obtaining protocols secure against semi-honest adversaries is a non-trivial task.

We begin by briefly recalling the [52, 44] construction for secure two-party computation in the semi-honest adversarial model. Let f be the two-party functionality that is to be securely computed. Then, the parties are given an arithmetic circuit over $GF(2)$ that computes the function f. The protocol starts with the parties sharing their inputs with each other using bitwise-xor secret sharing, and thus following this stage, they both hold shares of the input lines of the circuit. That is, for each input line l, party A holds a value a_l and party B holds a value b_l, such that both a_l and b_l are random under the constraint that $a_l + b_l$ equals the value of the input into this line. Next, the parties evaluate the circuit gate-by-gate, computing random shares of the output line of the gate from the random shares of the input lines to the gate. There are two types of gates in the circuit: addition gates and multiplication gates. Addition gates are evaluated by each party locally adding its shares of the input values. Multiplication gates are evaluated using 1-out-of-4 oblivious transfer (the oblivious transfer protocol used is basically that of [35]). In the above way, the parties jointly compute the circuit and obtain shares of the output gates. The protocol concludes with each party revealing the prescribed shares to the other party (i.e, if a certain output gate provides a bit of A's input, then B will reveal its share of this output line to A).

Our general construction is exactly that of GMW, except that the oblivious transfer protocol used is universally composable. That is, we first define an ideal oblivious transfer functionality, $\mathcal{F}_{\text{OT}}$, and show that in the $\mathcal{F}_{\text{OT}}$-hybrid model, the GMW protocol securely realizes any two-party functionality in the presence of semi-honest, *adaptive* adversaries. This holds unconditionally and even if the adversary and environment are computationally unbounded. Of course, computational assumptions are used for securely realizing $\mathcal{F}_{\text{OT}}$ itself. (Our construction is actually somewhat more general than that of GMW in that it deals with reactive functionalities that have multiple stages which are separately activated. This is achieved by having the parties hold shares of the state of the ideal functionality between activations.)

Next we present protocols that securely realize $\mathcal{F}_{\text{OT}}$ in the semi-honest case. In the non-adaptive case, the protocol of [35, 44] suffices. In the adaptive case, our protocol uses an augmented version of non-committing encryption [15]. The augmentation consists of two additional properties. First, the encryption scheme should have an alternative key generation algorithm that generates only public encryption keys without the corresponding decryption key. Second, the standard and additional key generation algorithms should be invertible in the sense that given the output key or keys, it is possible to find the random coin tosses used in generating these keys. (Following [24], we call these properties *oblivious key generation* and *invertible samplability.*) All known non-committing encryption schemes have this properties. In particular, such schemes exist under either the RSA assumption or the DDH assumption.) In all, we show:

Proposition 4.2.1 (semi-honest computation – informal): *Assume that trapdoor permutations exist. Then, for any two-party ideal functionality $\mathcal{F}$, there exists a protocol Π that securely realizes $\mathcal{F}$ in the presence of* semi-honest, static *adversaries. Furthermore, if augmented two-party non-committing encryption protocols exist, then there exists a protocol Π that securely realizes $\mathcal{F}$ in the presence of* semi-honest, adaptive *adversaries.*

Proposition 4.2.1 as stated above is not precise. This is due to two technicalities regarding the model of computation as defined in [14]. We therefore define a class of functionalities for which these technical problems do not arise and then construct secure protocols for any functionality in this class. See Section 4.3.3 for more discussion and an exact definition.

Another point where our results formally differ from Proposition 4.2.1 is due to the fact that, according to the definitions used here, protocols which do not generate any output are technically secure (for any functionality). Thus, Proposition 4.2.1 as stated, can be easily (but un-interestingly) achieved. In contrast, we prove the existence of protocols which *do* generate output and securely realize any functionality (we call such a protocol non-trivial; for more details, see the discussion after Definition 4.3.2 in Section 4.3.1). Proposition 4.2.1 is formally restated in Section 4.4.2.

Obtaining two-party computation secure against malicious adversaries

Having constructed a protocol that is universally composable when the adversary is limited to semi-honest behavior, we construct a protocol compiler that transforms this protocol into one that is secure even against malicious adversaries. From here on, we refer to the protocol that is secure against semi-honest adversaries as the "basic protocol". Recall that the basic protocol is only secure in the case that even the corrupted parties follow the protocol specification exactly, using a uniformly chosen random tape. Thus, in order to obtain a protocol secure against malicious adversaries, we need to enforce potentially malicious corrupted parties to behave in a semi-honest manner. First and foremost, this involves forcing the parties to follow the prescribed protocol. However, this only makes sense relative to a *given* input and random tape. Furthermore, a malicious party must be forced into using a *uniformly chosen* random tape. This is because the security of the basic protocol may depend on the fact that the party has no freedom in setting its own randomness. We begin with a description of the GMW compiler.

An informal description of the GMW compiler. The GMW compiler begins by having each party commit to its input. Next, the parties run a coin-tossing protocol in order to fix their random tapes. A simple coin-tossing protocol in which both parties receive the same uniformly distributed string is not sufficient here. This is because the parties' random tapes must remain secret. Instead, an augmented coin-tossing protocol is used, where one party receives a uniformly distributed string (to be used as its random tape) and the other party receives a commitment to that string. Now, following these two steps, each party holds its own input and uniformly distributed random tape, and a commitment to the other party's input and random tape.

Next, the commitments to the random tape and to the inputs are used to "enforce" semi-honest behavior. Observe that a protocol specification is a deterministic function of a party's view consisting of its input, random tape and messages received so far, and recall that each party holds a commitment to the input and random tape of the other party. Observe also that the messages sent so far are public. Therefore, the assertion that a new message is computed according to the protocol is an NP statement (and the party sending the message knows an adequate NP-witness to it). This means that the parties can use zero-knowledge proofs to show that their steps are indeed according to the protocol specification. Therefore, in the protocol emulation phase, the parties send messages according to the instructions of the basic protocol, while proving at each step that the messages sent are correct. The key point is that, due to the soundness of the proofs, even a malicious adversary cannot deviate from the protocol specification without being detected. Therefore, the adversary is limited to semi-honest behavior. Furthermore, since the proofs are zero-knowledge, nothing "more" is revealed in the compiled protocol than in the basic protocol. We conclude that the security of the compiled protocol (against malicious adversaries) is directly derived from the security of the basic protocol (against semi-honest adversaries).

In summary, the GMW compiler has three components: input commitment, coin-tossing and protocol emulation (where the parties prove that their steps are according to the protocol specification).

Universally composable protocol compilation. A natural way of adapting the GMW compiler to the setting of universally composable secure computation would be to take the same compiler, but rather use *universally composable* commitments, coin-tossing and zero-knowledge as sub-protocols. However, such a strategy fails because the receiver of a universally composable commitment receives *no information* about the value committed to. (Instead, the recipient receives only a formal "receipt" assuring it that a value was committed to. See Section 4.5 for more details.) Thus, there is no NP-statement that a party can prove relative to its input commitment. This is in contrast to the GMW protocol where standard (perfectly binding) commitments are used and thus each party holds a string that uniquely determines the other party's input and random tape.

A different strategy is therefore required for constructing a universally composable compiler. Before describing our strategy, observe that in GMW the use of the commitment scheme is not standard. Specifically, although both parties commit to their inputs etc., they never decommit. Rather, they *prove* NP-statements relative to their committed values. Thus, a natural primitive to use would be a "commit-and-prove" functionality, which is comprised of two phases. In the first phase, a party "commits" (or is bound) to a specific value. In the second phase, this party proves NP-statements (in zero-knowledge) relative to the committed value. We formulate this notion in a universally composable commit-and-prove functionality, denoted $\mathcal{F}_{\text{CP}}$, and then use this functionality to implement all three phases of the compiler. More specifically, our protocol compiler uses the "commit" phase of the $\mathcal{F}_{\text{CP}}$ functionality in order to execute the input and coin-tossing phases of the compiler. The "prove" phase of the $\mathcal{F}_{\text{CP}}$ functionality is then used to force the adversary to send messages according to the protocol specification and consistent with the committed input and the random tape resulting from the coin-tossing. The result is a universally composable analog to the GMW compiler. We remark that in the $\mathcal{F}_{\text{CP}}$-hybrid model the compiler is unconditionally secure against *adaptive* adversaries, even if the adversary and the environment are computationally unbounded.

We show how to securely realize $\mathcal{F}_{\text{CP}}$ in the $\mathcal{F}_{\text{ZK}}$-hybrid model, i.e. in a hybrid model with ideal access to an ideal zero-knowledge functionality, $\mathcal{F}_{\text{ZK}}$. (Functionality $\mathcal{F}_{\text{ZK}}$ expects to receive a statement x and a witness w from the prover. It then forwards x to the verifier, together with an assertion whether $R(x,w)$ holds, where R is a predetermined relation.) Essentially, in the commit phase of the commit-and-prove protocol, the committer commits to its input value w using some commitment scheme C, and in addition it proves to the receiver, using $\mathcal{F}_{\text{ZK}}$ with an appropriate relation, that it "knows" the committed value. In the prove phase, where the committer wishes to assert that the committed value w stands in relation R with some public value x, the committer presents x and w to $\mathcal{F}_{\text{ZK}}$ again — but this time the relation used by $\mathcal{F}_{\text{ZK}}$ asserts two properties: first that $R(x,w)$ holds, and second that w is the same value that was previously committed to.

To guarantee security against static adversaries, the commitment scheme of Naor [70] is sufficient as an instantiation of the scheme C. We thus obtain a protocol for securely realizing $\mathcal{F}_{\text{CP}}$ in the $\mathcal{F}_{\text{ZK}}$-hybrid model, based on any one-way function. To guarantee security against *adaptive* adversaries we need "adaptively secure" commitment schemes, namely commitment schemes where a simulator can generate "dummy commitments" which can be later opened in multiple ways. (In fact, a slightly stronger property is needed here, see details within.) Such commitments exist assuming the existence of trapdoor permutations, as is demonstrated by our construction of universally composable commitments in Section 4.5. In all we obtain:

Theorem 4.2.2 (two-party computation in the malicious model – informal): *Assume that trapdoor permutations exist. Then, for any two-party ideal functionality $\mathcal{F}$, there exists a protocol Π that securely realizes $\mathcal{F}$ in the $\mathcal{F}_{\text{ZK}}$-hybrid model in the presence of* malicious, static *adversaries. Furthermore, if augmented two-party non-committing encryption protocols exist, then there exists a protocol Π that securely realizes $\mathcal{F}$ in the $\mathcal{F}_{\text{ZK}}$-hybrid model in the presence of* malicious, adaptive *adversaries.*

Let $\mathcal{F}_{\text{CRS}}$ denote the common random string functionality (that is, $\mathcal{F}_{\text{CRS}}$ provides all parties with a common, public string drawn from a predefined distribution). Then, as we show in Section 4.5, universally composable commitments can be securely realized in the $\mathcal{F}_{\text{CRS}}$-hybrid model, assuming the existence of trapdoor permutations. Furthermore, [16] showed that the $\mathcal{F}_{\text{ZK}}$ functionality can be securely realized given universally composable commitments. Combining these results together, we have that $\mathcal{F}_{\text{ZK}}$ can be securely realized in the $\mathcal{F}_{\text{CRS}}$-hybrid model, assuming the existence of trapdoor permutations. Using the composition theorem we obtain a similar result to Theorem 4.2.2, with the exception that $\mathcal{F}$ is realized in the $\mathcal{F}_{\text{CRS}}$-hybrid model (rather than in the $\mathcal{F}_{\text{ZK}}$-hybrid model). As with Proposition 4.2.1, Theorem 4.2.2 is not stated exactly. It is formally restated in Section 4.8.2.

On the distribution of the reference string. In obtaining the above corollary, the common reference string is used only in the construction of the universally composable commitment scheme (which is used for obtaining $\mathcal{F}_{\text{ZK}}$). As we have mentioned, in the $\mathcal{F}_{\text{CRS}}$-hybrid model, universally composable commitments can be obtained assuming the existence of trapdoor permutations only. However, in this case, the common reference string is *not* uniformly distributed. Nevertheless, a uniformly distributed string can be used, under the additional assumption of the existence of *dense cryptosystems* [27]. We therefore conclude that universally composable two-party computation can be obtained with a *uniformly distributed* reference string, under the assumption that the following primitives exist: trapdoor permutations, dense cryptosystems and augmented two-party non-committing encryption protocols.

Extensions to multi-party computation

We now describe how the two-party construction of Theorem 4.2.2 is extended to the setting of multi-party computation, where any number of parties may be corrupt. Recall that in this setting, each set of interacting parties is assumed to have access to an *authenticated* broadcast channel.

The outline of our construction is as follows. Similarly to the two-party case, we first construct a multi-party protocol that is secure against semi-honest adversaries (as above, this protocol is essentially that of GMW). Then, we construct a protocol compiler (again, like that of GMW), that transforms semi-honest protocols into ones that are secure even against malicious adversaries. This protocol compiler is constructed using a one-to-many extension of the commit-and-prove functionality, denoted $\mathcal{F}_{\text{CP}}^{1:\text{M}}$. The extension of the protocol that realizes two-party $\mathcal{F}_{\text{CP}}$ to a protocol that realizes one-to-many $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ constitutes the main difference between the two-party and multi-party constructions. Therefore, in this outline, we focus exclusively on how this extension is achieved.

The first step in realizing $\mathcal{F}_{\text{CP}}^{1:\text{M}}$, is to construct one-to-many extensions of universal commitments and zero-knowledge. In a one-to-many commitment scheme, all parties receive the commitment (and the committer is bound to the same value for all parties). Likewise, in one-to-many zero-knowledge, all parties verify the proof (and they either all accept or all reject the proof). Now, any *non-interactive* commitment scheme can be transformed into a one-to-many equivalent by simply having the committer broadcast its message to all parties. Thus, this functionality is immediately obtained from our commitment scheme in Section 4.5 or from the scheme of [16] (both of these constructions are non-interactive). However, obtaining one-to-many zero-knowledge is more involved, since we do not know how to construct non-interactive adaptively-secure universally composable zero-knowledge.[1] Nevertheless, using the methodology of [44], a one-to-many zero-knowledge protocol can be constructed as follows. The construction is based on the universally-composable zero-knowledge protocol of [16]. Specifically, they show that parallel executions of the 3-round zero-knowledge protocol of Hamiltonicity is universally composable, when a universally composable commitment scheme is used for the prover's commitments. Thus, the prover runs a copy of the above zero-knowledge protocol with each receiver over the broadcast channel, using the one-to-many commitment scheme for its commitments. Furthermore, each verifying party checks that the proofs of all the other parties are accepting (this is possible because the proof of Hamiltonicity is publicly verifiable and because all parties view all the communication). Thus, at the end of the protocol, all honest parties agree (without any additional communication) on whether the proof was successful or not. (Note also that the adversary cannot cause an honest prover's proof to be rejected.)

[1] In the case of static adversaries, the non-interactive zero-knowledge protocol of [26] suffices. Thus, here too, the prover message can simply be broadcast and one-to-many zero-knowledge is obtained.

It remains to describe how to realize $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$ in the $\mathcal{F}_{\mathrm{ZK}}^{1:\mathrm{M}}$-hybrid model. The basic idea is to generalize the $\mathcal{F}_{\mathrm{CP}}$ protocol. As with zero-knowledge, this is not straightforward because in the protocol for adaptive adversaries, the $\mathcal{F}_{\mathrm{CP}}$ commit-phase is interactive. Nevertheless, this problem is solved by having the committer commit to its input value w by separately running the protocol for the commit-phase of (two-party) $\mathcal{F}_{\mathrm{CP}}$ with every party over the broadcast channel. Following this, the committer uses one-to-many zero-knowledge to prove that it committed to the same value in all of these commitments. (Since each party views the communication from all the commitments, every party can verify this zero-knowledge proof.) The prove phase is similar to the two-party case, except that the one-to-many extension of zero-knowledge is used (instead of two-party zero-knowledge).

Finally, we note that, as in the two-party case, a multi-party protocol compiler can be constructed in the $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$-hybrid model, with no further assumptions. Denoting the ideal broadcast functionality used by the parties by $\mathcal{F}_{\mathrm{BC}}$, we have the following theorem:

Theorem 4.2.3 (multi-party computation in the malicious model – informal): *Assume that trapdoor permutations exist. Then, for any multi-party ideal functionality $\mathcal{F}$, there exists a protocol Π that securely realizes $\mathcal{F}$ in the $(\mathcal{F}_{\mathrm{BC}}, \mathcal{F}_{\mathrm{CRS}})$-hybrid model in the presence of* malicious, static *adversaries, and for any number of corruptions. Furthermore, if augmented two-party non-committing encryption protocols exist, then there exists a protocol Π that securely realizes $\mathcal{F}$ in the $(\mathcal{F}_{\mathrm{BC}}, \mathcal{F}_{\mathrm{CRS}})$-hybrid model in the presence of* malicious, adaptive *adversaries, and for any number of corruptions.*

As with Proposition 4.2.1, Theorem 4.2.3 is not stated exactly. It is formally restated in Section 4.9.4.

4.3 Preliminaries

Section 4.3.1 reviews the framework of [14] and the universal composition theorem. In Section 4.3.2 we discuss issues that arise regarding universal composition when some amount of joint state between protocols is desired. Finally, Section 4.3.3 presents the class of functionalities which we will show how to securely realize. Before proceeding, we recall the definition of computational indistinguishability. A distribution ensemble $X = \{X(k,a)\}_{k\in\mathbf{N},a\in\{0,1\}^*}$ is an infinite set of probability distributions, where a distribution $X(k,a)$ is associated with each $k \in \mathbf{N}$ and $a \in \{0,1\}^*$. The ensembles considered in this work describe outputs where the parameter a represents input, and the parameter k is taken to be the security parameter. A distribution ensemble is called binary if it consists only of distributions over $\{0,1\}$. Then,

Definition 4.3.1 *Two* binary *distribution ensembles X and Y are* indistinguishable (*written* $X \overset{\mathrm{c}}{\equiv} Y$) *if for any $c \in \mathbf{N}$ there exists $k_0 \in \mathbf{N}$ such that for all $k > k_0$ and for all a we have*

$$|\Pr(X(k,a)=1) - \Pr(Y(k,a)=1)| < k^{-c}.$$

4.3.1 Universally Composable Security: The general framework

We start by reviewing the syntax of message-driven protocols in asynchronous networks. We then present the real-life model of computation, the ideal process, and the general definition of securely realizing an ideal functionality. Next we present the hybrid model and the composition theorem. The text is somewhat informal for clarity and brevity, and is mostly taken from the Overview section of [14]. For full details see there.

Protocol syntax. Following [55, 45], a protocol is represented as a system of probabilistic interactive Turing machines (ITMs), where each ITM represents the program to be run within a different party. Specifically, the input and output tapes model inputs and outputs that are received from and given to other programs running on the same machine, and the communication tapes model messages sent to and received from the network. Adversarial entities are also modeled as ITMs. We concentrate on a model where the adversaries have an arbitrary additional input, or an "advice" string. From a complexity-theoretic point of view, this essentially implies that adversaries are non-uniform ITMs.

In order to simplify the exposition, we introduce the following convention. We assume that all protocols are such that the parties read their input tapes only at the onset of a protocol execution. This can easily be achieved by having the parties copy their input tape onto an internal work tape. This convention prevents problems that may occur when parties' input tapes are modified in the middle of a protocol execution (as is allowed in the model).

The basic framework

As sketched in Section 4.2, protocols that securely carry out a given task (or, protocol problem) are defined in three steps, as follows. First, the process of executing a protocol in the presence of an adversary and in a given computational environment is formalized. Next, an "ideal process" for carrying out the task at hand is formalized. In the ideal process the parties do not communicate with each other. Instead they have access to an "ideal functionality", which is essentially an incorruptible "trusted party" that is programmed to capture the desired functionality of the given task. A protocol is said to securely realize an ideal functionality if the process of running the protocol amounts to "emulating" the ideal process for that ideal functionality. We overview the model for protocol execution (called the *real-life model*), the ideal process, and the notion of protocol emulation.

We concentrate on the following model of computation, aimed at representing current realistic communication networks (such as the Internet). The communication takes place in an asynchronous, public network, without guaranteed delivery of messages. We assume that the communication is *authenticated* and thus the adversary cannot modify messages sent by honest parties.[2] Furthermore, the adversary may only deliver messages that were previously sent by parties, and may deliver each message sent only once. The fact that the network is asynchronous means that the messages are not necessarily delivered in the order which they are sent. Parties may be broken into (i.e., become corrupted) throughout the computation, and once corrupted their behavior is arbitrary (or, *malicious*). (Thus, our main consideration is that of malicious, adaptive adversaries. However, below we present the modifications necessary for modeling static and semi-honest adversaries.) We do not trust data erasures; rather, we postulate that past states are available to the adversary upon corruption. Finally, all the involved entities are restricted to probabilistic polynomial time (or "feasible") computation.

Protocol execution in the real-life model. We sketch the process of executing a given protocol π (run by parties $P_1, ..., P_n$) with some adversary $\mathcal{A}$ and an environment machine $\mathcal{Z}$ with input z. All parties have a security parameter $k \in \mathbf{N}$ and are polynomial in k. The execution consists

[2]We remark that the basic model in [14] postulates *unauthenticated* communication, i.e. the adversary may delete, modify, and generate messages at wish. Here we concentrate on authenticated networks for sake of simplicity. Authentication can be added in standard ways. Formally, the model here corresponds to the $\mathcal{F}_{\text{AUTH}}$-hybrid model in [14].

of a sequence of *activations,* where in each activation a single participant (either $\mathcal{Z}$, $\mathcal{A}$, or some P_i) is activated. The environment is activated first. In each activation it may read the contents of the output tapes of all the *uncorrupted* parties[3] and the adversary, and may write information on the input tape of one of the parties or of the adversary. Once the activation of the environment is complete (i,e, once the environment enters a special waiting state), the entity whose input tape was written on is activated next.

Once the adversary is activated, it may read its own tapes and the outgoing communication tapes of all parties. It may either deliver a message to some party by writing this message on the party's incoming communication tape or corrupt a party. Only messages that were sent in the past by some party can be delivered, and each message can be delivered at most once. Upon corrupting a party, the adversary gains access to all the tapes of that party and controls all the party's future actions. (We assume that the adversary also learns all the past internal states of the corrupted party. This means that the model does not assume effective cryptographic erasure of data.) In addition, whenever a party is corrupted the environment is notified (say, via a message that is added to the output tape of the adversary). If the adversary delivered a message to some uncorrupted party in its activation then this party is activated once the activation of the adversary is complete. Otherwise the environment is activated next.

Once a party is activated (either due to an input given by the environment or due to a message delivered by the adversary), it follows its code and possibly writes local outputs on its output tape and outgoing messages on its outgoing communication tape. Once the activation of the party is complete the environment is activated. The protocol execution ends when the environment completes an activation without writing on the input tape of any entity. The output of the protocol execution is the output of the environment. We assume that this output consists of only a single bit.

Let $\text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(k, z, \overline{r})$ denote the output of environment $\mathcal{Z}$ when interacting with adversary $\mathcal{A}$ and parties running protocol π on security parameter k, input z and random tapes $\overline{r} = r_{\mathcal{Z}}, r_{\mathcal{A}}, r_1, \ldots, r_n$ as described above (z and $r_{\mathcal{Z}}$ for $\mathcal{Z}$, $r_{\mathcal{A}}$ for $\mathcal{A}$; r_i for party P_i). Let $\text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(k, z)$ denote the random variable describing $\text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(k, z, \overline{r})$ when $\overline{r}$ is uniformly chosen. Let $\text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}$ denote the ensemble $\{\text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(k, z)\}_{k\in\mathbf{N}, z\in\{0,1\}^*}$.

The ideal process. Security of protocols is defined via comparing the protocol execution in the real-life model to an *ideal process* for carrying out (a single instance of) the task at hand. A key ingredient in the ideal process is the ideal functionality that captures the desired functionality, or the specification, of that task. The ideal functionality is modeled as another ITM that interacts with the environment and the adversary via a process described below. More specifically, the ideal process involves an ideal functionality $\mathcal{F}$, an ideal process adversary $\mathcal{S}$, an environment $\mathcal{Z}$ with input z, and a set of dummy parties $\tilde{P}_1, ..., \tilde{P}_n$.

As in the process of protocol execution in the real-life model, the environment is activated first. As there, in each activation it may read the contents of the output tapes of all (dummy) parties and the adversary, and may write information on the input tape of either one of the (dummy) parties or of the adversary. Once the activation of the environment is complete the entity whose input tape was written on is activated next.

The dummy parties are fixed and simple ITMs: Whenever a dummy party is activated with input x, it forwards x to the ideal functionality $\mathcal{F}$, say by writing x on the incoming communication

[3]The adversary is not given read access to the corrupted parties' output tapes because once a party is corrupted, it is no longer activated. Rather, the adversary sends messages in its name. Therefore, the output tapes of corrupted parties are not relevant.

tape of $\mathcal{F}$. In this case $\mathcal{F}$ is activated next, and a note that the party sent a message to $\mathcal{F}$ is written on the incoming communication tape of $\mathcal{S}$. Whenever a dummy party is activated due to delivery of some message (from $\mathcal{F}$), it copies this message to its output. In this case $\mathcal{Z}$ is activated next.

Once $\mathcal{F}$ is activated, it reads the contents of its incoming communication tape, and potentially sends messages to the parties and to the adversary by writing these messages on its outgoing communication tape. Once the activation of $\mathcal{F}$ is complete, the entity that was last activated before $\mathcal{F}$ is activated again. In the case this entity was one of the dummy parties, it immediately relinquishes control to $\mathcal{Z}$.

Once the adversary $\mathcal{S}$ is activated, it may read its own input tape and in addition it can read the *destinations* of the messages on the outgoing communication tape of $\mathcal{F}$. That is, $\mathcal{S}$ can see the identity of the recipient of each message sent by $\mathcal{F}$, but it cannot see the *contents* of this message (unless the recipient of the message is $\mathcal{S}$ or a corrupted party[4]). $\mathcal{S}$ may either deliver a message from $\mathcal{F}$ to some party by having this message copied to the party's incoming communication tape, write a message from itself on $\mathcal{F}$'s incoming communication tape[5], or corrupt a party. Upon corrupting a party, both $\mathcal{Z}$ and $\mathcal{F}$ learn the identity of the corrupted party (say, a special message is written on their respective incoming communication tapes).[6] In addition, the adversary learns all the past inputs and outputs of the party. Finally, the adversary controls the party's actions from the time that the corruption takes place.

If the adversary delivered a message to some uncorrupted (dummy) party in an activation then this party is activated once the activation of the adversary is complete. Otherwise the environment is activated next.

As in the real-life model, the protocol execution ends when the environment completes an activation without writing on the input tape of any entity. The output of the protocol execution is the (one bit) output of $\mathcal{Z}$.

Let $\textsc{ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k,z,\overline{r})$ denote the output of environment $\mathcal{Z}$ after interacting in the ideal process with adversary $\mathcal{S}$ and ideal functionality $\mathcal{F}$, on security parameter k, input z, and random input $\overline{r} = r_{\mathcal{Z}}, r_{\mathcal{S}}, r_{\mathcal{F}}$ as described above (z and $r_{\mathcal{Z}}$ for $\mathcal{Z}$, $r_{\mathcal{S}}$ for $\mathcal{S}$; $r_{\mathcal{F}}$ for $\mathcal{F}$). Let $\textsc{ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k,z)$ denote the random variable describing $\textsc{ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k,z,\overline{r})$ when $\overline{r}$ is uniformly chosen. Let $\textsc{ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ denote the ensemble $\{\textsc{ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k,z)\}_{k\in\mathbf{N},z\in\{0,1\}^*}$.

Securely realizing an ideal functionality. We say that a protocol ρ securely realizes an ideal functionality $\mathcal{F}$ if for any real-life adversary $\mathcal{A}$ there exists an ideal-process adversary $\mathcal{S}$ such that no environment $\mathcal{Z}$, on any input, can tell with non-negligible probability whether it is interacting with $\mathcal{A}$ and parties running ρ in the real-life process, or with $\mathcal{S}$ and $\mathcal{F}$ in the ideal process. This means that, from the point of view of the environment, running protocol ρ is 'just as good' as interacting with an ideal process for $\mathcal{F}$. (In a way, $\mathcal{Z}$ serves as an "interactive distinguisher" between the two

[4]Note that the ideal process allows $\mathcal{S}$ to obtain the output values sent by $\mathcal{F}$ to the corrupted parties as soon as they are generated. Furthermore, if at the time that $\mathcal{S}$ corrupts some party P_i there are messages sent from $\mathcal{F}$ to P_i, then $\mathcal{S}$ immediately obtains the contents of these messages.

[5]Many natural ideal functionalities indeed send messages to the adversary $\mathcal{S}$ (see the zero-knowledge and commitments functionalities of Sections 4.6 and 4.5 for examples). On the other hand, having the adversary send messages to $\mathcal{F}$ is less common. Nevertheless, this option can be useful in order to relax the requirements on protocols that realize the functionality. For example, it may be easier to obtain coin-tossing if the adversary is allowed to bias some of the bits of the result. If this is acceptable for the application in mind, we can allow the adversary this capability by having it send its desired bias to $\mathcal{F}$.

[6]Allowing $\mathcal{F}$ to know which parties are corrupted gives it considerable power. This power provides greater freedom in formulating ideal functionalities for capturing the requirements of given tasks. On the other hand, it also inherently limits the scope of general realizability theorems. See more discussion in Section 4.3.3.

processes. Here it is important that $\mathcal{Z}$ can provide the process in question with *adaptively chosen* inputs throughout the computation.) We have:

Definition 4.3.2 *Let $n \in \mathbf{N}$. Let $\mathcal{F}$ be an ideal functionality and let π be an n-party protocol. We say that π* securely realizes *$\mathcal{F}$ if for any adversary $\mathcal{A}$ there exists an ideal-process adversary $\mathcal{S}$ such that for any environment $\mathcal{Z}$,*

$$\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} \overset{c}{\equiv} \text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}. \tag{4.1}$$

Non-trivial protocols and the requirement to generate output. Recall that the ideal process does not require the ideal-process adversary to deliver messages that are sent by the ideal functionality to the dummy parties. Consequently, the definition provides no guarantee that a protocol will ever generate output or "return" to the calling protocol. Indeed, in our setting where message delivery is not guaranteed, it is impossible to ensure that a protocol "terminates" or generates output. Rather, the definition concentrates on the security requirements *in the case that the protocol generates output.*

A corollary of the above fact is that a protocol that "hangs", never sends any messages and never generates output, securely realizes any ideal functionality. Thus, in order to obtain a meaningful feasibility result, we introduce the notion of a non-trivial protocol. Such a protocol has the property that if the real-life adversary delivers all messages and does not corrupt any parties, then the ideal-process adversary also delivers all messages (and does not corrupt any parties). Note that in a non-trivial protocol, a party may not necessarily receive output. However, this only happens if either the functionality does not specify output for this party, or if the real-life adversary actively interferes in the execution (by either corrupting parties or refusing to deliver some messages). Our main result is to show the existence of *non-trivial* protocols for securely realizing any ideal functionality. All our protocols are in fact clearly non-trivial; therefore, we ignore this issue from here on.

Relaxations of Definition 4.3.2. We recall two standard relaxations of the definition:

- *Static (non-adaptive) adversaries.* Definition 4.3.2 allows the adversary to corrupt parties throughout the computation. A simpler (and somewhat weaker) variant forces the real-life adversary to corrupt parties only at the onset of the computation, before any uncorrupted party is activated. We call such adversaries static.

- *Passive (semi-honest) adversaries.* Definition 4.3.2 gives the adversary complete control over corrupted parties (such an adversary is called malicious). Specifically, the model states that from the time of corruption the corrupted party is no longer activated, and instead the adversary sends messages in the name of that party. In contrast, when a semi-honest adversary corrupts a party, the party continues to follow the prescribed protocol. Nevertheless, the adversary is given read access to the internal state of the party at all times, and is also able to modify the values that the environment writes on the corrupted parties' input tapes.[7] Formally, if in a given activation, the environment wishes to write information on the input tape of a *corrupted* party, then the environment first passes the adversary the value x that it wishes to write (along with the identity of the party whose input tape it wishes to write to). The adversary then passes

[7]Allowing a semi-honest adversary to modify a corrupted party's input is somewhat non-standard. However, this simplifies the presentation of this work (and in particular the protocol compiler). All the protocols presented for the semi-honest model in this paper are secure both when the adversary can modify a corrupted party's input tape and when it cannot.

a (possibly different) value x' back to the environment. Finally, the environment writes x' on the input tape of the corrupted party, following which the corrupted party is activated. We stress that when the environment writes on the input tape of an honest party, the adversary learns nothing of the value and cannot modify it. Everything else remains the same as in the above-described malicious model. We say that protocol π securely realizes functionality $\mathcal{F}$ *for semi-honest adversaries,* if for any semi-honest real-life adversary $\mathcal{A}$ there exists an ideal-process semi-honest adversary $\mathcal{S}$ such that Eq. (4.1) holds for any environment $\mathcal{Z}$.

The composition theorem

The hybrid model. In order to state the composition theorem, and in particular in order to formalize the notion of a real-life protocol with access to multiple copies of an ideal functionality, the hybrid model of computation with access to an ideal functionality $\mathcal{F}$ (or, in short, the $\mathcal{F}$-hybrid model) is formulated. This model is identical to the real-life model, with the following additions. On top of sending messages to each other, the parties may send messages to and receive messages from an unbounded number of copies of $\mathcal{F}$. Each copy of $\mathcal{F}$ is identified via a unique session identifier (SID); all messages addressed to this copy and all message sent by this copy carry the corresponding SID. (Sometimes a copy of $\mathcal{F}$ will interact only with a subset of the parties. The identities of these parties is determined by the protocol in the $\mathcal{F}$-hybrid model.)

The communication between the parties and each one of the copies of $\mathcal{F}$ mimics the ideal process. That is, once a party sends a message m to a copy of $\mathcal{F}$ with a particular SID, that copy is immediately activated to receive this message. (If no such copy of $\mathcal{F}$ exists then a new copy of $\mathcal{F}$ is created and immediately activated to receive m.) Furthermore, although the adversary in the hybrid model is responsible for delivering the messages from the copies of $\mathcal{F}$ to the parties, it does not have access to the contents of these messages.

The hybrid model does not specify how the SIDs are generated, nor does it specify how parties "agree" on the SID of a certain protocol copy that is to be run by them. These tasks are left to the protocol in the hybrid model. This convention simplifies formulating ideal functionalities, and designing protocols that securely realize them, by freeing the functionality from the need to choose the SIDs and guarantee their uniqueness. In addition, it seems to reflect common practice of protocol design in existing networks. See more discussion following Theorem 4.3.3 below.

Let $\text{HYBRID}^{\mathcal{F}}_{\pi,\mathcal{A},\mathcal{Z}}(k,z)$ denote the random variable describing the output of environment machine $\mathcal{Z}$ on input z, after interacting in the $\mathcal{F}$-hybrid model with protocol π, adversary $\mathcal{A}$, analogously to the definition of $\text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(k,z)$. (We stress that here π is a hybrid of a real-life protocol with ideal evaluation calls to $\mathcal{F}$.) Let $\text{HYBRID}^{\mathcal{F}}_{\pi,\mathcal{A},Z}$ denote the distribution ensemble $\{\text{HYBRID}^{\mathcal{F}}_{\pi,\mathcal{A},Z}\}_{k\in\mathbf{N},z\in\{0,1\}^*}$.

Replacing a call to $\mathcal{F}$ with a protocol invocation. Let π be a protocol in the $\mathcal{F}$-hybrid model, and let ρ be a protocol that securely realizes $\mathcal{F}$ (with respect to some class of adversaries). The composed protocol π^ρ is constructed by modifying the code of each ITM in π so that the first message sent to each copy of $\mathcal{F}$ is replaced with an invocation of a new copy of ρ with fresh random input, with the same SID, and with the contents of that message as input. Each subsequent message to that copy of $\mathcal{F}$ is replaced with an activation of the corresponding copy of ρ, with the contents of that message given to ρ as new input. Each output value generated by a copy of ρ is treated as a message received from the corresponding copy of $\mathcal{F}$. (See [14] for more details on the operation of "composed protocols", where a party, i.e. an ITM, runs multiple protocol-instances concurrently.)

If protocol ρ is a protocol in the real-life model then so is π^ρ. If ρ is a protocol in some $\mathcal{G}$-hybrid model (i.e., ρ uses ideal evaluation calls to some functionality $\mathcal{G}$) then so is π^ρ.

Theorem statement. In its general form, the composition theorem basically says that if ρ securely realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model for some functionality $\mathcal{G}$, then an execution of the composed protocol π^ρ, running in the $\mathcal{G}$-hybrid model, "emulates" an execution of protocol π in the $\mathcal{F}$-hybrid model. That is, for any adversary $\mathcal{A}$ in the $\mathcal{G}$-hybrid model there exists an adversary $\mathcal{S}$ in the $\mathcal{F}$-hybrid model such that no environment machine $\mathcal{Z}$ can tell with non-negligible probability whether it is interacting with $\mathcal{A}$ and π^ρ in the $\mathcal{G}$-hybrid model or it is interacting with $\mathcal{S}$ and π in the $\mathcal{F}$-hybrid model.

A corollary of the general theorem states that if π securely realizes some functionality $\mathcal{I}$ in the $\mathcal{F}$-hybrid model, and ρ securely realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model, then π^ρ securely realizes $\mathcal{I}$ in the $\mathcal{G}$-hybrid model. (Here one has to define what it means to securely realize functionality $\mathcal{I}$ in the $\mathcal{F}$-hybrid model. This is done in the natural way.) That is:

Theorem 4.3.3 ([14]) *Let $\mathcal{F},\mathcal{G},\mathcal{I}$ be ideal functionalities. Let π be an n-party protocol in the $\mathcal{F}$-hybrid model, and let ρ be an n-party protocol that securely realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model. Then for any adversary $\mathcal{A}$ in the $\mathcal{G}$-hybrid model there exists an adversary $\mathcal{S}$ in the $\mathcal{F}$-hybrid model such that for any environment machine $\mathcal{Z}$ we have:*

$$\textsc{hybrid}^{\mathcal{G}}_{\pi^\rho,\mathcal{A},\mathcal{Z}} \overset{c}{\equiv} \textsc{hybrid}^{\mathcal{F}}_{\pi,\mathcal{S},\mathcal{Z}}. \tag{4.2}$$

In particular, if π securely realizes functionality $\mathcal{I}$ in the $\mathcal{F}$-hybrid model then π^ρ securely realizes $\mathcal{I}$ in the $\mathcal{G}$-hybrid model.

On the uniqueness of the session IDs. The session IDs play a central role in the hybrid model and the composition operation, in that they enable the parties to distinguish different instances of a protocol. Indeed, differentiating protocol instances via session IDs is a natural and common mechanism in protocol design.

Yet, the current formulation of the hybrid model provides a somewhat over-idealized treatment of session IDs. Specifically, it is assumed that the session IDs are *globally unique* and *common to all parties.* That is, it is assumed that no two copies of an ideal functionality with the same session ID exist, even if the two copies have different (and even disjoint) sets of participants. Furthermore, all parties are assumed to hold the same SID (and they must somehow have agreed upon it). This treatment greatly simplifies the exposition of the model and the definition of ideal functionalities and protocols that realize them. Nonetheless, it is somewhat restrictive in that it requires the protocol in the hybrid model to guarantee global uniqueness of common session IDs. This may be hard (or even impossible) to achieve in the case that the protocol in the hybrid model is truly distributed and does not involve global coordination. See [66] for more discussion on this point. More elaborate ways of defining session IDs so as not to require global uniqueness exist. We leave this issue for future work.

4.3.2 Universal Composition with Joint State

Traditionally, composition operations among protocols assume that the composed protocol instances have disjoint states, and in particular independent local randomness. The universal composition operation is no exception: if protocol ρ securely realizes some ideal functionality $\mathcal{F}$, and protocol π in the $\mathcal{F}$-hybrid model uses m copies of $\mathcal{F}$, then the composed protocol π^ρ uses m independent copies of ρ, and no two copies of ρ share any amount of state.

This property of universal composition (and of protocol composition in general) is bothersome in our context, where we wish to construct and analyze protocols in the common reference string (CRS) model. Let us elaborate. Assume that we follow the natural formalization of the CRS model as the $\mathcal{F}_{\text{CRS}}$-hybrid model, where $\mathcal{F}_{\text{CRS}}$ is the functionality that chooses a string from the specified distribution and hands it to all parties. Now, assume that we construct a protocol ρ that realizes some ideal functionality $\mathcal{F}$ in the $\mathcal{F}_{\text{CRS}}$-hybrid model (say, let $\mathcal{F}$ be the commitment functionality, $\mathcal{F}_{\text{COM}}$). Assume further that some higher level protocol π (in the $\mathcal{F}$-hybrid model) uses multiple copies of $\mathcal{F}$, and that we use the universal composition operation to replace each copy of $\mathcal{F}$ with an instance of ρ. We now obtain a protocol π^ρ that runs in the $\mathcal{F}_{\text{CRS}}$-hybrid model and emulates π. However, this protocol is highly wasteful of the reference string. Specifically, each instance of ρ in π^ρ has its own separate copy of $\mathcal{F}_{\text{CRS}}$, or in other words each instance of ρ requires its own independent copy of the reference string. This stands in sharp contrast with our common view of the CRS model, where an unbounded number of protocol instances should be able to use the *same copy* of the reference string.

One way to get around this limitation of universal composition (and composition theorems in general) is to treat the entire, multi-session interaction as a single instance of a more complex protocol, and then to explicitly require that all sessions use the same copy of the reference string. More specifically, proceed as follows. First, given a functionality $\mathcal{F}$ as described above, define a functionality, $\hat{\mathcal{F}}$, called the "multi-session extension of $\mathcal{F}$". Functionality $\hat{\mathcal{F}}$ will run multiple copies of $\mathcal{F}$, where each copy will be identified by a special "sub-session identifier", *ssid*. Upon receiving a message for the copy associated with *ssid*, $\hat{\mathcal{F}}$ activates the appropriate copy of $\mathcal{F}$ (running within $\hat{\mathcal{F}}$), and forwards the incoming message to that copy. If no such copy of $\mathcal{F}$ exists then a new copy is invoked and is given that *ssid*. Outputs generated by the copies of $\mathcal{F}$ are copied to $\hat{\mathcal{F}}$'s output. The next step after having defined $\hat{\mathcal{F}}$ is to construct protocols that directly realize $\hat{\mathcal{F}}$ in the $\mathcal{F}_{\text{CRS}}$-hybrid model, while making sure that the constructed protocols use only a single copy of $\mathcal{F}_{\text{CRS}}$.

This approach works, in the sense that it allows constructing and analyzing universally composable protocols that are efficient in their use of the reference string. However, it results in a cumbersome and non-modular formalization of ideal functionalities and protocols in the CRS model. Specifically, if we want to make sure that multiple sessions of some protocol (or set of protocols) use the same copy of the reference string, then we must treat all these sessions (that may take place among different sets of parties) as a single instance of some more complex protocol. For example, assume that we want to construct commitments in the CRS model, then use these commitments to construct UC zero-knowledge protocols, and then use these protocols in yet higher-level protocols. Then, in any level of protocol design, we must design functionalities and protocols that explicitly deal with multiple sessions. Furthermore, we must prove the security of these protocols within this multi-session setting. This complexity obviates much of the advantages of universal composition (and protocol composition in general).

In contrast, we would like to be able to formulate a functionality that captures only a single instance of some interaction, realize this functionality by some protocol π in the CRS model, and then securely compose multiple copies of π in spite of the fact that all copies use the same copy of the reference string. This approach is, in general, dangerous, since it can lead to insecure protocols. However, there are conditions under which such "composition with joint state" maintains security. This section describes a general tool that enables the composition of protocols even when they have some amount of joint state, under some conditions. Using this tool (suggested in [21] and called universal composition with joint state (JUC)), we are able to state and realize most of the functionalities in this work as functionalities for a single session, while still ending up with protocols where an unbounded number of instances use the same copy of the common reference string. This greatly simplifies the presentation while not detracting from the composability and efficiency of the presented protocols.

In a nutshell, universal composition with joint state is a new composition operation that can be sketched as follows. Let $\mathcal{F}$ be an ideal functionality, and let π be a protocol in the $\mathcal{F}$-hybrid model. Let $\hat{\mathcal{F}}$ denote the "multi-session extension of $\mathcal{F}$" sketched above, and let ρ be a protocol that securely realizes $\hat{\mathcal{F}}$. Then construct the composed protocol $\pi^{[\rho]}$ by replacing *all copies* of $\mathcal{F}$ in π by a *single copy* of ρ. (We stress that π assumes that it has access to multiple independent copies of $\mathcal{F}$. Still, we replace all copies of $\mathcal{F}$ with a single copy of some protocol.) The JUC theorem states that protocol $\pi^{[\rho]}$, running in the real-life model, "emulates" π in the usual sense. A more detailed presentation follows.

The multi-session extension of an ideal functionality. We formalize the notion of a multi-session extension of an ideal functionality, sketched above. Let $\mathcal{F}$ be an ideal functionality. Recall that $\mathcal{F}$ expects each incoming message to contain a special field consisting of its session ID (SID). All messages received by $\mathcal{F}$ are expected to have the same SID. (Messages that have different SIDs than that of the first message are ignored.) Similarly, all outgoing messages generated by $\mathcal{F}$ carry the same SID.

The multi-session extension of $\mathcal{F}$, denoted $\hat{\mathcal{F}}$, is defined as follows. $\hat{\mathcal{F}}$ expects each incoming message to contain *two* special fields. The first is the usual SID field as in any ideal functionality. The second field is called the sub-session ID (SSID) field. Upon receiving a message $(sid, ssid, v)$ (where sid is the SID, $ssid$ is the SSID, and v is an arbitrary value or list of values), $\hat{\mathcal{F}}$ first verifies that sid is the same as that of the first message, otherwise the message is ignored. Next, $\hat{\mathcal{F}}$ checks if there is a running copy of $\mathcal{F}$ whose session ID is $ssid$. If so, then $\hat{\mathcal{F}}$ activates that copy of $\mathcal{F}$ with incoming message $(ssid, v)$, and follows the instructions of this copy. Otherwise, a new copy of $\mathcal{F}$ is invoked (within $\hat{\mathcal{F}}$) and immediately activated with input $(ssid, v)$. From now on, this copy is associated with sub-session ID $ssid$. Whenever a copy of $\mathcal{F}$ sends a message $(ssid, v')$ to some party P_i, $\hat{\mathcal{F}}$ sends $(sid, ssid, v')$ to P_i, and sends $ssid$ to the adversary. (Sending $ssid$ to the adversary implies that $\hat{\mathcal{F}}$ does not hide which copy of $\mathcal{F}$ is being activated within $\hat{\mathcal{F}}$.)

The composition operation. Let $\mathcal{F}$ be an ideal functionality. The composition operation, called universal composition with joint state (JUC), takes two protocols as arguments: a protocol π in the $\mathcal{F}$-hybrid model and a protocol ρ that securely realizes $\hat{\mathcal{F}}$. The result is a composed protocol denoted $\pi^{[\rho]}$ and described as follows.

Recall that the $\mathcal{F}$-hybrid model is identical to the real-life model of computation, with the exception that the parties have access to multiple copies of $\mathcal{F}$. The different copies of $\mathcal{F}$ are identified via their SIDs as described above. Let $\mathcal{F}_{(sid)}$ denote the copy of functionality $\mathcal{F}$ with SID sid. Protocol $\pi^{[\rho]}$ behaves like π with the following exceptions:[8]

1. When activated for the first time within party P_i, $\pi^{[\rho]}$ invokes a copy of protocol ρ with SID sid_0. That is, a copy of the ith Interactive Turing Machine in ρ is invoked as a subroutine within P_i, and is (locally) given identity sid_0. No activation of ρ occurs yet. (sid_0 is some fixed, predefined value. For instance, set $sid_0 = 0$.)

[8]For simplicity, we assume that ρ securely realizes $\hat{\mathcal{F}}$ in the real-life model of computation. The composition operation and theorem can be extended in a natural way to account for protocols ρ that securely realize $\hat{\mathcal{F}}$ in the $\mathcal{G}$-hybrid model for some ideal functionality $\mathcal{G}$.

2. Whenever π instructs party P_i to send a message (sid, v) to $\mathcal{F}_{(sid)}$, protocol π^ρ instructs P_i to activate ρ with input value (sid_0, sid, v).

3. Whenever protocol ρ wishes to send a message m to some party $P_{i'}$, P_i writes the message $(\rho, sid, P_{i'}, m)$ on the outgoing communication tape.

4. Whenever activated due to delivery of a message (ρ, sid, m) from $P_{i'}$, P_i activates ρ with incoming message $(sid, P_{i'}, m)$.

5. Whenever (the single copy of) ρ generates an output value (sid, v), proceed just as π proceeds with incoming message v from $\mathcal{F}_{(sid)}$.

Theorem statement. The JUC theorem asserts that if $\hat{\rho}$ securely realizes $\hat{\mathcal{F}}$, then protocol $\pi^{[\hat{\rho}]}$ behaves essentially like π with ideal access to multiple independent copies of $\mathcal{F}$. More precisely,

Theorem 4.3.4 (Universal composition with joint state [21]): *Let $\mathcal{F}, \mathcal{G}$ be ideal functionalities. Let π be a protocol in the $\mathcal{F}$-hybrid model, and let $\hat{\rho}$ be a protocol that securely realizes $\hat{\mathcal{F}}$, the multi-session extension of $\mathcal{F}$, in the $\mathcal{G}$-hybrid model. Then the composed protocol $\pi^{[\hat{\rho}]}$ in the $\mathcal{G}$-hybrid model emulates protocol π in the $\mathcal{F}$-hybrid model. That is, for any adversary $\mathcal{A}$ there exists an adversary $\mathcal{S}$ such that for any environment $\mathcal{Z}$ we have*

$$\text{HYBRID}^{\mathcal{F}}_{\pi,\mathcal{S},\mathcal{Z}} \overset{c}{\equiv} \text{HYBRID}^{\mathcal{G}}_{\pi^{[\rho]},\mathcal{A},\mathcal{Z}}.$$

In particular, if π securely realizes some functionality $\mathcal{I}$ in the $\mathcal{F}$-hybrid model then $\pi^{[\rho]}$ securely realizes $\mathcal{I}$ in the $\mathcal{G}$-hybrid model.

Discussion. Jumping ahead, we sketch our use of the JUC theorem. Recall the commitment functionality, $\mathcal{F}_{\text{COM}}$, formalized in [16]. This functionality captures the process of commitment and decommitment to a single value, performed by two parties. In addition, [16] show how to realize $\hat{\mathcal{F}}_{\text{COM}}$ in the CRS model, using a single copy of the CRS for all commitments. (In [16] functionality $\hat{\mathcal{F}}_{\text{COM}}$ is called $\mathcal{F}_{\text{MCOM}}$.) An alternative protocol that realizes $\hat{\mathcal{F}}_{\text{COM}}$ is also presented here.

In this work we construct protocols that use these commitment protocols. However, to preserve modularity of exposition, we present our protocols in the $\mathcal{F}_{\text{COM}}$-hybrid model, while allowing the protocols to use multiple copies of $\mathcal{F}_{\text{COM}}$ and thus enjoy full modularity. We then use universal composition with joint state to compose any protocol π in the $\mathcal{F}_{\text{COM}}$-hybrid model with any protocol ρ that securely realized $\hat{\mathcal{F}}_{\text{COM}}$ using a single copy of the reference string, to obtain a protocol $\pi^{[\rho]}$ that emulates π and uses only a single copy of the reference string for all the commitments. (We remark that the same technique is applied also to protocols that use the ideal zero-knowledge functionality, $\mathcal{F}_{\text{ZK}}$. See more details in Section 4.6.)

4.3.3 Well-Formed Functionalities

We would like to be able to state a theorem saying that *any* ideal functionality can be securely realized. However, for technical reasons, such a claim cannot be made in our model. The first problem that arises is as follows. Since the ideal functionality is informed of the identities of the corrupted parties, it can do things that cannot be realized by any protocol. For example, consider the ideal functionality that lets all parties know which parties are corrupted. Then this functionality cannot be realized in the face of an adversary that corrupts a single random party but instructs that party to continue following the prescribed protocol.

In order to bypass this problem, we define a special class of functionalities that do not utilize their direct knowledge of the identities of the corrupted parties. For the lack of a better name, we call these functionalities well-formed. A well-formed functionality consists of a main procedure (called the shell) and a subroutine (called the core.) The core is an arbitrary probabilistic polynomial-time algorithm, while the shell is a simple procedure described as follows. The shell forwards any incoming message to the core, with the exception that notifications of corruptions of parties are *not* forwarded. Outgoing messages generated by the core are copied by the shell to the outgoing communication tape. The above definition guarantees that the code of a well-formed ideal functionality "does not depend" on its direct knowledge regarding who is corrupted.

In subsequent sections, we show how to realize any well-formed functionality in the face of *static* adversaries. However, another technicality arises when considering *adaptive* adversaries. Consider for instance a two-party ideal functionality $\mathcal{F}$ that works as follows: Upon activation, it chooses two large random primes p and q and sends $n = pq$ to both parties. The value n is the only message output by the functionality; in particular, it never reveals the values p and q. The important property of this functionality that we wish to focus on is the fact that it has *private randomness* that is never revealed. Such a functionality can be securely realized in the static corruption model. However, consider what happens in a real execution if an adaptive adversary corrupts *both* parties after they output n. In this case, all prior randomness is revealed (recall that we assume no erasures). Therefore, if this randomness explicitly defines the primes p and q (as is the case in all known protocols for such a problem), these values will necessarily be revealed to the adversary. On the other hand, in the ideal process, even if both parties are corrupted, p and q are never revealed.[9] (We stress that this is not a weakness of the model because in the case that all participating parties are corrupted, there are no security requirements on a protocol. In particular, there are no honest parties to "protect"). In light of the above discussion, we define adaptively well-formed functionalities that do not keep private randomness when all parties are corrupted. Formally, such functionalities have a shell and a core, as described above. However, in addition to forwarding messages to and from the core, the shell keeps track of the parties with whom the functionality interacts. If at some activation all these parties are corrupted, then the shell sends the random tape of the core to the adversary. Thus, when all the parties participating in some activation are corrupted, all the randomness is revealed (even in the ideal process). We show how any adaptively well-formed functionality can be securely realized in the face of adaptive adversaries.

In order to make sure that the multi-session extension of an adaptively well-formed functionality remains adaptively well-formed, we slightly modify the definition of the multi-session extension of an ideal functionality (see Section 4.3.2) as follows. If the given ideal functionality $\mathcal{F}$ is adaptively well-formed, then $\hat{\mathcal{F}}$, the multi-session extension of $\mathcal{F}$, is an adaptively well-formed functionality defined as follows. The core of $\hat{\mathcal{F}}$ consists of the multi-session extension (in the usual sense) of the core of $\mathcal{F}$. The shell of $\hat{\mathcal{F}}$ is as defined above except that it separately keeps track of the participating parties of each session. Then, if all the participating parties of some session are corrupted in some activation, the shell sends the random tape of the core for that session to the adversary. (Recall that each session of the multi-session functionality uses an independent random tape.) We note that the JUC theorem (Theorem 4.3.4) holds even with respect to the modified definition of multi-session extensions.

[9]We do not claim that it is impossible to realize this specific functionality. Indeed, it may be possible to sample the domain $\{n \mid n = pq\}$ (or a domain that is computationally indistinguishable from it) without knowing p or q. Nevertheless, the example clearly demonstrates the problem that arises.

4.4 Two-party Secure Computation for Semi-Honest Adversaries

This section presents general constructions for securely realizing any two-party ideal functionality in the presence of semi-honest adversaries. In the case of static adversaries, the construction is basically that of Goldreich, Micali and Wigderson [52]. In the case of adaptive adversaries the construction of oblivious transfer is somewhat different (however, the rest of the construction remains unchanged). The main difference between our presentation and that of [44], is that we consider the reactive case where parties may receive new inputs during the run of the protocol. Each new input may depend on the current adversarial view of the system. In particular, it may depend on previous outputs of this execution and on the activity in other executions. We note that although the basic construction of our protocol is very similar to [52], our proof is significantly different. This is due to the fact that we show universal composability, and in addition consider also the adaptive corruption case (and not only the static case like in [52]).

We begin by presenting the oblivious-transfer ideal functionality $\mathcal{F}_{\text{OT}}$, and demonstrate how to realize this functionality in the presence of semi-honest adversaries (both static and adaptive). Following this we present our protocol for securely realizing any two-party functionality, in the $\mathcal{F}_{\text{OT}}$-hybrid model.

4.4.1 Universally Composable Oblivious Transfer

Oblivious transfer [75, 35] is a two-party functionality, involving a sender with input $x_1, ..., x_\ell$, and a receiver with input $i \in \{1, \ldots, \ell\}$. The receiver should learn x_i (and nothing else) and the sender should learn nothing. An exact definition of the ideal oblivious transfer functionality, denoted $\mathcal{F}_{\text{OT}}^\ell$, appears in Figure 4.1. (Using standard terminology, $\mathcal{F}_{\text{OT}}^\ell$ captures 1-out-of-ℓ OT.)

Functionality $\mathcal{F}_{\text{OT}}^\ell$

$\mathcal{F}_{\text{OT}}^\ell$ proceeds as follows, parameterized with an integer ℓ and running with an oblivious transfer sender T, a receiver R and an adversary $\mathcal{S}$.

1. Upon receiving a message $(\mathsf{sender}, sid, x_1, ..., x_\ell)$ from T, where each $x_j \in \{0,1\}^m$, record the tuple $(x_1, ..., x_\ell)$. (The length of the strings m is fixed and known to all parties.)
2. Upon receiving a message $(\mathsf{receiver}, sid, i)$ from R, where $i \in \{1, \ldots, \ell\}$, send (sid, x_i) to R and (sid) to $\mathcal{S}$, and halt. (If no $(\mathsf{sender}, \ldots)$ message was previously sent, then send nothing to R.)

Figure 4.1: The oblivious transfer functionality, $\mathcal{F}_{\text{OT}}^\ell$

Section 4.4.1 presents a protocol that securely realizes $\mathcal{F}_{\text{OT}}$ for static adversaries. Section 4.4.1 presents our protocol for securely realizing $\mathcal{F}_{\text{OT}}$ for adaptive adversaries.

Static UC Oblivious Transfer

The oblivious transfer protocol of [52, 44], denoted SOT (for Static Oblivious Transfer) is presented in Figure 4.2. For simplicity we present the protocol for the case where each of the ℓ input values is a single bit. (In the semi-honest case, oblivious transfer for strings can be constructed from this one via the composition theorem.)

Protocol SOT

Proceed as follows, on security parameter k.

1. Given input $(\mathsf{sender}, sid, x_1, ..., x_\ell)$, party T chooses a trapdoor permutation f over $\{0,1\}^k$, together with its inverse f^{-1}, and sends (sid, f) to the receiver R. (The permutation f is chosen uniformly from a given family of trapdoor permutations.)
2. Given input $(\mathsf{receiver}, sid, i)$, and having received (sid, f) from T, receiver R chooses $y_1, ...y_{i-1}, r, y_{i+1}, ..., y_\ell \in_R \{0,1\}^k$, computes $y_i = f(r)$, and sends $(sid, y_1, \ldots, y_\ell)$ to T.
3. Having received $(sid, y_1, \ldots, y_\ell)$ from R, the sender T sends $(sid, x_1 \oplus B(f^{-1}(y_1)), \ldots, x_\ell \oplus B(f^{-1}(y_\ell)))$ to R, where $B(\cdot)$ is a hard-core predicate for f.
4. *Output:* Having received $(sid, b_1, \ldots, b_\ell)$ from T, the receiver R outputs $(sid, b_i \oplus B(r))$.

Figure 4.2: The static, semi-honest oblivious transfer protocol

Claim 4.4.1 *Assuming that f is a trapdoor permutation, Protocol* SOT *securely realizes $\mathcal{F}_{\text{OT}}^\ell$ in the presence of semi-honest, static adversaries.*

Proof: Let $\mathcal{A}$ be a semi-honest, static adversary that interacts with parties running the above protocol. We construct an adversary $\mathcal{S}$ for the ideal process for $\mathcal{F}_{\text{OT}}^\ell$ such that no environment $\mathcal{Z}$ can tell with non-negligible probability whether it is interacting with $\mathcal{A}$ and the above protocol or with $\mathcal{S}$ in the ideal process for $\mathcal{F}_{\text{OT}}^\ell$. Recall that $\mathcal{S}$ interacts with the ideal functionality $\mathcal{F}_{\text{OT}}^\ell$ and with the environment $\mathcal{Z}$. Simulator $\mathcal{S}$ starts by invoking a copy of $\mathcal{A}$ and running a simulated interaction of $\mathcal{A}$ with $\mathcal{Z}$ and parties running the protocol. (We refer to the interaction of $\mathcal{S}$ in the ideal process as *external interaction.* The interaction of $\mathcal{A}$ with the simulated $\mathcal{A}$ is called *internal interaction.*) $\mathcal{S}$ proceeds as follows:

Simulating the communication with $\mathcal{Z}$: Every input value that $\mathcal{S}$ receives from $\mathcal{Z}$ is written on $\mathcal{A}$'s input tape (as if coming from $\mathcal{A}$'s environment). Likewise, every output value written by $\mathcal{A}$ on its output tape is copied to $\mathcal{S}$'s own output tape (to be read by $\mathcal{S}$'s environment $\mathcal{Z}$).

Simulating the case where the sender T only is corrupted: $\mathcal{S}$ simulates a real execution in which T is corrupted. $\mathcal{S}$ begins by activating $\mathcal{A}$ and receiving the message (sid, f) that $\mathcal{A}$ (controlling T) would send R in a real execution. Then, $\mathcal{S}$ chooses $y_1, \ldots, y_\ell \in_R \{0,1\}^k$ and simulates R sending T the message $(sid, y_1, \ldots, y_\ell)$ in the internal interaction. Finally, when $\mathcal{A}$ sends the message $(sid, b_1, \ldots, b_\ell)$ from T to R in the internal interaction, $\mathcal{S}$ externally sends T's input $x_1, \ldots, x_\ell$ to $\mathcal{F}_{\text{OT}}^\ell$ and delivers the output from $\mathcal{F}_{\text{OT}}^\ell$ to R. (Recall that in the semi-honest model as defined here, $\mathcal{A}$ is able to modify the input tape of T. Therefore, the value $x_1, \ldots, x_\ell$ sent by $\mathcal{S}$ to $\mathcal{F}_{\text{OT}}^\ell$ is the (possibly) modified value. Formally this causes no problem because actually it is the environment who writes the modified value, after "consultation" with $\mathcal{A}$. Since all communication is forwarded unmodified between $\mathcal{A}$ and $\mathcal{Z}$, the value that $\mathcal{Z}$ writes on T's input tape is the already-modified value. We ignore this formality in the subsequent proofs in this section.)

Simulating the case where the receiver R only is corrupted: $\mathcal{S}$ begins by activating $\mathcal{A}$ and internally sending it the message that $\mathcal{A}$ (controlling R) receives from T in a real execution. That is, $\mathcal{S}$ chooses a random trapdoor permutation f and its inverse f^{-1}, and sends it to

$\mathcal{A}$. Next, it internally receives a message of the form $(sid, y_1, \ldots, y_\ell)$ from $\mathcal{A}$. Simulator $\mathcal{S}$ then externally sends R's input i to $\mathcal{F}^{\ell}_{\text{OT}}$ and receives back the output x_i. $\mathcal{S}$ concludes the simulation by choosing $b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_\ell$ uniformly, setting $b_i = x_i \oplus B(f^{-1}(y_i))$, and internally sending $\mathcal{A}$ the message $(sid, b_1, \ldots, b_\ell)$. (Recall that x_i is the output as obtained by $\mathcal{S}$ from the ideal functionality $\mathcal{F}^{\ell}_{\text{OT}}$.)

Simulating the case that neither party is corrupted: In this case, $\mathcal{S}$ receives a message (sid) signalling it that T and R concluded an ideal execution with $\mathcal{F}_{\text{OT}}$. $\mathcal{S}$ then generates a simulated transcript of messages between the real model parties. That is, $\mathcal{S}$ generates T's first message (sid, f) as the real T would, sets R's reply to be $(sid, y_1, \ldots, y_\ell)$ where $y_j \in_R \{0,1\}^k$ for each j, and finally sets T's second message to $(sid, b_1, \ldots, b_\ell)$ where $b_j \in_R \{0,1\}$ for every j.

Simulating the case that both parties are corrupted: In this case, $\mathcal{S}$ knows both parties' inputs and can therefore simulate a protocol execution by generating the actual messages that the parties would send in a real execution.

We demonstrate that $\text{IDEAL}_{\mathcal{F}_{\text{OT}},\mathcal{S},\mathcal{Z}} \overset{c}{\equiv} \text{REAL}_{\text{SOT},\mathcal{A},\mathcal{Z}}$. This is done by showing that the joint view of $\mathcal{Z}$ and $\mathcal{A}$ in the execution of SOT is indistinguishable from the joint view of $\mathcal{Z}$ and the simulated $\mathcal{A}$ within $\mathcal{S}$ in the ideal process. First, notice that the simulation for the case where T is corrupted is perfect. This is because in both the ideal simulation and a real execution, the message received by T consists of ℓ uniformly distributed k-bit strings, and the output of R is the same in both executions. (Notice that since f is a permutation, choosing r uniformly and computing $y_i = f(r)$, as occurs in a real execution, results in a uniformly distributed y_i. Furthermore, the output of R is $b_i \oplus B(f^{-1}(y_i))$ where b_i is the ith value sent by T.) Second, we claim that the simulation for the case where R is corrupted is indistinguishable from in a real execution. The only difference between the two is in the message $b_1, \ldots, b_\ell$ received by R. The bit b_i is identically distributed in both cases (in particular, in both the simulation and a real execution it equals $x_i \oplus B(f^{-1}(y_i))$). However, all the bits b_j for $j \neq i$ are uniformly chosen and are not distributed according to $x_j \oplus B(f^{-1}(y_j))$. Nevertheless, due to the hard-core properties of B, the bit $B(f^{-1}(y_j))$ for a random y_j is indistinguishable from a random-bit $b_j \in_R \{0,1\}$. The same is also true for $x_j \oplus B(f^{-1}(y_j))$ when x_j is fixed before y_j is chosen. (More precisely, given an environment that distinguishes with non-negligible probability between the real-life and the ideal interactions, we can construct an adversary that contradicts the hard-core property of B.) Thus the views are indistinguishable. By the same argument, we also have that the simulation for the case that neither party is corrupted results in a view that is indistinguishable from a real execution. This completes the proof. ■

Our proof of security of the above protocol fails in the case of adaptive adversaries. Intuitively the reason is that when a party gets corrupted, $\mathcal{S}$ cannot present the simulated $\mathcal{A}$ with a valid internal state of the corrupted party. (This internal state should be consistent with the past messages sent by the party and with the local input and output of that party.) In particular, the messages (sid, f), $(sid, y_1, \ldots, y_\ell)$ and $(sid, b_1, \ldots, b_\ell)$ fully define the input bits $x_1, \ldots, x_\ell$. However, in the case that T is not initially corrupted, $\mathcal{S}$ does not know $x_1, \ldots, x_\ell$ and therefore with high probability, the messages define a different set of input bits. Thus, if $\mathcal{A}$ corrupts T after the execution has concluded, $\mathcal{S}$ cannot provide $\mathcal{A}$ with an internal state of T that is consistent both with $x_1, \ldots, x_\ell$ and the simulated transcript that it had previously generated.

Adaptive UC Oblivious Transfer

Due to the above-described problem, we use a different protocol for securely realizing $\mathcal{F}_{\mathrm{OT}}^{\ell}$ for the case of adaptive, semi-honest adversaries. A main ingredient in this protocol are non-committing encryptions as defined in [15] and constructed in [15, 6, 24]. In addition to standard semantic security, such encryption schemes have the property that ciphertexts that can be opened to both 0 and 1 can be generated. That is, a non-committing (bit) encryption scheme consists of a tuple (G, E, D, S), where G is a key generation algorithm, E and D are encryption and decryption algorithms, and S is a simulation algorithm (for generating non-committing ciphertexts). The triple (G, E, D) satisfies the usual properties of semantically secure encryption. That is, $G(1^k) = (e, d)$ where e and d are the respective encryption and decryption keys, and $D_d(E_e(m)) = m$ except with negligible probability. Furthermore, $\{E_e(1)\}$ is indistinguishable from $\{E_e(0)\}$. In addition, the simulator algorithm S is able to generate "dummy ciphertexts" that can be later "opened" as encryptions of either 0 or 1. More specifically, it is required that $S(1^k) = (e, c, r_0, r_1, d_0, d_1)$ with the following properties:

- The tuple (e, c, r_0, d_0) looks like a normal encryption and decryption process for the bit 0. That is, (e, c, r_0, d_0) is indistinguishable from a tuple (e', c', r', d') where (e', d') is a randomly chosen pair of encryption and decryption keys, r' is randomly chosen, and $c' = E_e(0; r')$. (In particular, it should hold that $D_{d_0}(c) = 0$.)
- The tuple (e, c, r_1, d_1) looks like a normal encryption and decryption process for the bit 1. That is, (e, c, r_1, d_1) is indistinguishable from a tuple (e', c', r', d') where (e', d') is a randomly chosen pair of encryption and decryption keys, r' is randomly chosen, and $c' = E_e(1; r')$. (In particular, it should hold that $D_{d_1}(c) = 1$.)

Thus, given a pair (e, c), it is possible to explain c both as an encryption of 0 (by providing d_0 and r_0) and as an encryption of 1 (by providing d_1 and r_1). Here, we actually use *augmented* non-committing encryption protocols that have the following two additional properties:

1. Oblivious key generation: It should be possible to choose a public encryption key e "without knowing" the decryption key d. That is, there should exist a different key generation algorithm $\hat{G}$ such that $\hat{G}(1^k) = \hat{e}$ where $\hat{e}$ is indistinguishable from the encryption key e chosen by G, and in addition $\{E_{\hat{e}}(0)\}$ remains indistinguishable from $\{E_{\hat{e}}(1)\}$ even when the entire random input of $\hat{G}$ is known.

2. Invertible samplability: this property states that the key generation and oblivious key generation algorithms G and $\hat{G}$ should be invertible. That is, we require the existence of an inverting algorithm who receives any e output by the simulator algorithm S and outputs r such that $\hat{G}(r) = e$. (This algorithm may receive the coins used by S in computing e in order to find r.) We also require an algorithm that receives any pair (e, d_i) for $i \in \{0, 1\}$ from the output of S, and outputs r such that $G(r) = (e, d_i)$. (As before, this algorithm may receive the coins used by S.) The idea here is that in order to "explain" the simulator-generated keys as being generated in a legal way, it must be possible to find legal random coin tosses for them.[10]

Augmented two-party non-committing encryption protocols exist under either one of the RSA or the DDH assumptions. The requirements are also fulfilled in the case that public keys are uniformly

[10]In its most general form, one can define an invertible sampling algorithm for G that receives *any* pair (e, d) in the range of G and outputs r such that $G(r) = (e, d)$. However, we actually only need the inverting algorithm to work for keys output by the simulator S.

distributed over some public domain and secret keys are defined by the random coins input to G. See more details in [15, 24].

The protocol for securely realizing $\mathcal{F}_{\mathrm{OT}}^{\ell}$, denoted AOT (for Adaptive Oblivious Transfer) is presented in Figure 4.3. As in the static case, the protocol is defined for the case where each of the ℓ input values is a single bit.

Protocol AOT

Proceed as follows, on security parameter k and using an augmented non-committing encryption scheme $(G, \hat{G}, E, D, S)$.

1. Given input $(\mathsf{receiver}, sid, i)$, receiver R runs $G(1^k)$ to obtain (e, d), and runs $\hat{G}(1^k)$ $\ell-1$ times to obtain $\hat{e}_1, ..., \hat{e}_{i-1}, \hat{e}_{i+1}, ..., \hat{e}_\ell$. Then, R sends $(sid, \hat{e}_1, ..., \hat{e}_{i-1}, e, \hat{e}_{i+1}, ..., \hat{e}_\ell)$ to T.
2. Given input $(\mathsf{sender}, sid, x_1, ..., x_\ell)$, and having received $(sid, e_1, ..., e_\ell)$ from R, sender T computes $c_j = E_{e_j}(x_j)$ for every $1 \leq j \leq \ell$, and sends $(sid, c_1, ..., c_\ell)$ to R.
3. Having received $(sid, c_1, \ldots, c_\ell)$ from T, receiver R computes $x_i = D_d(c_i)$ and outputs (sid, x_i).

Figure 4.3: The adaptive, semi-honest oblivious transfer protocol

Claim 4.4.2 *Assume that* $(G, \hat{G}, E, D, S)$ *is an augmented non-committing encryption scheme. Then, Protocol* AOT *securely realizes* $\mathcal{F}_{\mathrm{OT}}^{\ell}$ *in the presence of semi-honest, adaptive adversaries.*

Proof: The main difference between this proof and the proof of Claim 4.4.1 is due to the fact that the real-model adversary $\mathcal{A}$ can corrupt parties during (or after) the simulation. When $\mathcal{S}$ receives such a "corrupt" command, it corrupts the ideal-model party and receives its input (and possibly its output). Given this information, $\mathcal{S}$ must produce random coins for this party such that the simulated transcript generated so far is consistent with the revealed input (and output).

Let $\mathcal{A}$ be a semi-honest, *adaptive* adversary that interacts with parties running the above protocol. We construct an adversary $\mathcal{S}$ in the ideal process for $\mathcal{F}_{\mathrm{OT}}^{\ell}$ such that no environment $\mathcal{Z}$ can tell with non-negligible probability whether it is interacting with $\mathcal{A}$ and the above protocol or with $\mathcal{S}$ in the ideal process for $\mathcal{F}_{\mathrm{OT}}^{\ell}$. In describing the simulation here, it is helpful to explicitly consider the ITMs representing the real model parties in the internal interaction. We denote these parties by $\tilde{T}$ and $\tilde{R}$. $\mathcal{S}$ works as follows:

Simulating the communication with $\mathcal{Z}$: Every input value that $\mathcal{S}$ receives from $\mathcal{Z}$ is written on $\mathcal{A}$'s input tape (as if coming from $\mathcal{A}$'s environment). Likewise, every output value written by $\mathcal{A}$ on its output tape is copied to $\mathcal{S}$'s own output tape (to be read by $\mathcal{S}$'s environment $\mathcal{Z}$).

Simulating the receiver: We separately describe the simulation for a corrupted and uncorrupted receiver.

1. *Simulation when the receiver is not corrupted:* In this case, $\mathcal{S}$ needs to simulate the receiver message. This is done as follows: $\mathcal{S}$ runs the encryption simulation algorithm $S(1^k)$ independently ℓ times. For each j, $\mathcal{S}$ obtains a tuple $(e_j, c_j, r_0^j, r_1^j, d_0^j, d_1^j)$; see the explanation above for the meaning of each element in this tuple. Then, $\mathcal{S}$ generates $\tilde{R}$'s message to be $(sid, e_1, \ldots, e_\ell)$ and simulates $\tilde{R}$ sending it to $\tilde{T}$.

2. *Simulation when the receiver is corrupted:* In this case, $\mathcal{S}$ holds the input $(\mathsf{receiver}, sid, i)$ of the ideal receiver R and constructs a virtual real-model receiver $\tilde{R}$ as follows. The contents of the input tape of $\tilde{R}$ is set to $(\mathsf{receiver}, sid, i)$. In order to set the contents of $\tilde{R}$'s random tape, $\mathcal{S}$ first runs the encryption simulation algorithm $S(1^k)$ independently ℓ times, obtaining tuples $(e_j, c_j, r_0^j, r_1^j, d_0^j, d_1^j)$. Next, for every j, $\mathcal{S}$ uses the invertible sampling algorithm in order to find r_j so that $\hat{G}(r_j) = e_j$, where $\hat{G}$ is the oblivious key generator. Then, $\mathcal{S}$ sets the contents of $\tilde{R}$'s random tape to equal $r_1, \ldots, r_\ell$.

   $\mathcal{S}$ passes the internal state of $\tilde{R}$ (including the contents of its input and random tapes) to $\mathcal{A}$ and waits for $\mathcal{A}$ to activate $\tilde{R}$ in the simulation. When this occurs, $\mathcal{S}$ internally obtains the message $(sid, e_1, \ldots, e_\ell)$ that $\tilde{R}$ writes on its outgoing message tape, and externally sends $(\mathsf{receiver}, sid, i)$ to $\mathcal{F}_{\mathrm{OT}}^\ell$. (We note that for every $j \neq i$, it holds that $e_j = \hat{G}(r_j)$ is the same e_j as generated by $S(1^k)$. On the other hand, $e_i = G(r_i)$, where G is the standard encryption key generator.)

Simulating the sender: Once again, we separately consider the case that the sender is corrupted or not corrupted.

1. *Simulation when the sender is not corrupted:* $\mathcal{S}$ simulates $\tilde{T}$ sending $(sid, c_1, \ldots, c_\ell)$ where the c_i's were generated from $S(1^k)$ when simulating the receiver message above. This is the same whether or not R was corrupted.
2. *Simulation when the sender is corrupted:* $\mathcal{S}$ holds the ideal T's input $(\mathsf{sender}, sid, x_1, \ldots, x_\ell)$ and constructs a virtual real-model sender $\tilde{T}$ by writing $(\mathsf{sender}, sid, x_1, \ldots, x_\ell)$ on its input tape and a *uniformly* distributed string on its random tape. Then, as above, $\mathcal{S}$ passes $\mathcal{A}$ the internal state of $\tilde{T}$ (consisting of the contents of its input and random tapes). When $\mathcal{A}$ activates $\tilde{T}$, simulator $\mathcal{S}$ externally sends the message $(\mathsf{sender}, sid, x_1, \ldots, x_\ell)$ to $\mathcal{F}_{\mathrm{OT}}^\ell$.

Dealing with "corrupt" commands: We assume that any corruption of a party occurs after the party has sent its protocol message in the simulation. (Otherwise, the corruption essentially occurs before the protocol begins and the instructions above suffice.) Now, if $\mathcal{S}$ receives a command from $\mathcal{A}$ to corrupt the real-model $\tilde{R}$, then it corrupts the ideal model R and obtains its input i and its output x_i. Given i and x_i, simulator $\mathcal{S}$ passes $\mathcal{A}$ the decryption-key $d_{x_i}^i$ (and thus the ciphertext c_i given to R in the simulated sender-message is "decrypted" to x_i). Furthermore, for every $j \neq i$, $\mathcal{S}$ runs the invertible sampling algorithm on e_j in order to find r_j such that $\hat{G}(r_j) = e_j$. Finally, $\mathcal{S}$ runs the invertible sampling algorithm on e_i in order to find r_i such that $G(r_i) = (e_i, d_{x_i}^i)$. Notice that these two invertible sampling algorithms are different. $\mathcal{S}$ supplies $\mathcal{A}$ with the random tape $r_1, \ldots, r_\ell$ for $\tilde{R}$.

If $\mathcal{S}$ receives a command from $\mathcal{A}$ to corrupt real-model $\tilde{T}$, then it first corrupts the ideal-model T and obtains $x_1, \ldots, x_\ell$. Next, it prepares appropriate randomness to make it appear that for every j, it holds that $c_j = E_{e_j}(x_j)$ (where the (c_j, e_j) pairs are taken from the simulated receiver and sender messages). Since the encryption keys are non-committing and were generated by running $S(1^k)$, we have that for every $1 \leq j \leq \ell$ simulator $\mathcal{S}$ has a string r_{x_j} such that $c_j = E_{e_j}(x_j; r_{x_j})$. Therefore, $\mathcal{S}$ writes $r_{x_1}, \ldots, r_{x_\ell}$ as $\tilde{T}$'s random tape.

Output and output delivery: $\mathcal{S}$ delivers the output from $\mathcal{F}_{\mathrm{OT}}^\ell$ to (an uncorrupted) R when $\mathcal{A}$ delivers the message from $\tilde{T}$ to $\tilde{R}$ in the simulation.

As argued in the proof of Claim 4.4.1, it suffices to show that $\mathcal{A}$'s view in the simulation is indistinguishable from its view in a real execution. (Note that in the adaptive case there is a positive correctness error. That is, there is non-zero probability that the outputs of the uncorrupted parties in the real-life interaction will differ from their outputs in the ideal process. This error probability is due to the fact that encryption schemes can "fail" with negligible probability. Since the probability of such an event is negligible, we ignore it here.) The indistinguishability of the views is demonstrated using the properties of the augmented non-committing encryption scheme. In particular, the non-committing encryption keys, ciphertexts and decommitment strings are all indistinguishable from those appearing in a real execution. Furthermore, by the oblivious key-generation algorithm, $\mathcal{S}$ supplies only a single decryption key (for the i^{th} encryption key) and this is what a real receiver would also have. (More precisely, given an environment that distinguishes between the real-life and ideal interactions we construct an adversary that breaks either the security of the non-committing encryption or the oblivious key generation property. We omit further details.) ■

4.4.2 The General Construction

We are now ready to show how to securely realize any (adaptively) well-formed two-party functionality in the $\mathcal{F}_{\text{OT}}$-hybrid model, in the semi-honest case. (Adaptively well-formed functionalities are defined in Section 4.3.3. Two-party functionalities are functionalities that interact with the adversary, plus at most two parties.) The construction is essentially that of [52, 44], although as we have mentioned, our protocol is actually more general in that it also deals with reactive functionalities. We begin by formally restating Proposition 4.2.1:

Proposition 4.4.3 (Proposition 4.2.1 – formally restated): *Assume that trapdoor permutations exist. Then, for any two-party well-formed ideal functionality $\mathcal{F}$, there exists a non-trivial protocol that securely realizes $\mathcal{F}$ in the presence of* semi-honest, static *adversaries. Furthermore, if two-party augmented non-committing encryption protocols exist, then for any two-party adaptively well-formed ideal functionality $\mathcal{F}$, there exists a non-trivial protocol that securely realizes $\mathcal{F}$ in the presence of* semi-honest, adaptive *adversaries.*

Recall that a protocol is non-trivial if the ideal-process adversary delivers all messages from the functionality to the parties whenever the real-life adversary delivers all messages and doesn't corrupt any parties. This restriction is included to rule out meaningless protocols such as the protocol that never generates output. (See Section 4.3.1 for more discussion.)

The construction. Let $\mathcal{F}$ be an ideal two-party functionality and let P_1 and P_2 be the participating parties. The first step in constructing a protocol that securely realizes $\mathcal{F}$ is to represent (the shell of) $\mathcal{F}$ via a family $C_{\mathcal{F}}$ of boolean circuits. That is, the m^{th} circuit in the family describes an activation of $\mathcal{F}$ when the security parameter is set to m. Following [52, 44], we use arithmetic circuits over GF(2) where the operations are addition and multiplication modulo 2.

For simplicity we assume that the input and output of each party in each activation has at most m bits, the number of random bits used by $\mathcal{F}$ in all activations is at most m, and at the end of each activation the local state of $\mathcal{F}$ can be described in at most m bits. Consequently the circuit has $3m$ input lines and $3m$ output lines, with the following interpretation. In each activation, only one party has input. Therefore, m of the input lines are allocated for this input. The other $2m$ input lines describe $\mathcal{F}$'s m random coins and the length-m internal state of $\mathcal{F}$ at the onset of an activation. The $3m$ output lines are allocated as follows: m output lines for the output of each of the two parties and m output lines to describe the state of $\mathcal{F}$ following an activation. The circuit is constructed so that each input from the adversary is set to 0, and outputs to the adversary are ignored.[11] We note that if the input or output of a party in some activation is less than m bits then this is encoded in a standard way. Also, each party initially sets its shares of the state of $\mathcal{F}$ to 0 (this denotes the empty state).

Protocol $\Pi_{\mathcal{F}}$ (for securely realizing $\mathcal{F}$): We state the protocol for an activation in which P_1 sends a message to $\mathcal{F}$; the case where P_2 sends the message is easily derived (essentially by exchanging the roles of P_1 and P_2 throughout the protocol). It is assumed that both parties hold the same session identifier sid as auxiliary input. When activated with input (sid, v) within P_1, the protocol first sends a message to the partner P_2, asking it to participate in a joint evaluation of the m^{th} circuit in $C_{\mathcal{F}}$. Next, P_1 and P_2 engage in a gate-by-gate evaluation of $C_{\mathcal{F}}$, on inputs that represent the incoming message v from P_1, the current internal state of $\mathcal{F}$, and a random string. This is done as follows.

1. **Input Preparation Stage:**

   - *Input value:* Recall that v is P_1's input for this activation. P_1 first pads v to be of length exactly m (using some standard encoding). Next P_1 "shares" its input. That is, P_1 chooses a random string $v_1 \in_R \{0,1\}^m$ and defines $v_2 = v_1 \oplus v$. Then, P_1 sends (sid, v_2) to P_2 and stores v_1.

   - *Internal state:* At the onset of each activation, the parties hold shares of the current internal state of $\mathcal{F}$. That is, let c denote the current internal state of $\mathcal{F}$, where $|c| = m$. Then, P_1 and P_2 hold $c_1, c_2 \in \{0,1\}^m$, respectively, such that c_1 and c_2 are random under the restriction that $c = c_1 \oplus c_2$. (In the first activation of $\mathcal{F}$, the internal state is empty and so the parties' shares both equal 0.)

   - *Random coins:* Upon the first activation of $\mathcal{F}$ only, parties P_1 and P_2 choose random strings $r_1 \in_R \{0,1\}^m$ and $r_2 \in_R \{0,1\}^m$, respectively. These constitute shares of the random coins $r = r_1 \oplus r_2$ to be used by $C_{\mathcal{F}}$. We stress that r_1 and r_2 are chosen upon the first activation only. The same r_1 and r_2 are then used for each subsequent activation of $\mathcal{F}$ (r_1 are r_2 are kept the same because the random tape of $\mathcal{F}$ does not change from activation to activation).

   At this point, P_1 and P_2 hold (random) shares of the input message to $\mathcal{F}$, the internal state of $\mathcal{F}$ and the random tape of $\mathcal{F}$. That is, they hold shares of every input line into $C_{\mathcal{F}}$. Note that the only message sent in the above stage is the input share v_2 sent from P_1 to P_2.

2. **Circuit Evaluation:** P_1 and P_2 proceed to evaluate the circuit $C_{\mathcal{F}}$ in a gate-by-gate manner. Let α and β denote the values of the two input lines to a given gate. Then P_1 holds bits α_1, β_1 and P_2 holds bits α_2, β_2 such that $\alpha = \alpha_1 + \alpha_2$ and $\beta = \beta_1 + \beta_2$. The gates are computed as follows:

   - *Addition gates:* If the gate is an addition gate, then P_1 locally sets its share of the output line of the gate to be $\gamma_1 = \alpha_1 + \beta_1$. Similarly, P_2 locally sets its share of the output line of the gate to be $\gamma_2 = \alpha_2 + \beta_2$. (Thus $\gamma_1 + \gamma_2 = \alpha + \beta$.)

   - *Multiplication gates:* If the gate is a multiplication gate, then the parties use $\mathcal{F}_{\text{OT}}^4$ in order to compute their shares of the output line of the gate. That is, the parties wish to compute random shares γ_1 and γ_2 such that $\gamma_1 + \gamma_2 = \alpha \cdot \beta = (\alpha_1 + \alpha_2)(\beta_1 + \beta_2)$. For this purpose,

[11] Thus, we effectively prevent the ideal-model adversary from utilizing its capability of sending and receiving messages. This simplifies the construction, and only strengthens the result.

P_1 chooses a random bit $\gamma_1 \in_R \{0,1\}$, sets its share of the output line of the gate to γ_1, and defines the following table:

Value of (α_2, β_2)	Receiver input i	Receiver output γ_2
(0,0)	1	$o_1 = \gamma_1 + \alpha_1 \cdot \beta_1$
(0,1)	2	$o_2 = \gamma_1 + \alpha_1 \cdot (\beta_1 + 1)$
(1,0)	3	$o_3 = \gamma_1 + (\alpha_1 + 1) \cdot \beta_1$
(1,1)	4	$o_4 = \gamma_1 + (\alpha_1 + 1) \cdot (\beta_1 + 1)$

Having prepared this table, P_1 sends the oblivious transfer functionality $\mathcal{F}_{\mathrm{OT}}^4$ the message $(\mathsf{sender}, sid \circ j, o_1, o_2, o_3, o_4)$, where this is the j^{th} multiplication gate in the circuit and $\circ$ denotes concatenation (the index j is included in order to ensure that the inputs of the parties match to the same gate). P_2 sets its input value i for $\mathcal{F}_{\mathrm{OT}}^4$ according to the above table (e.g., for $\alpha_2 = 1$ and $\beta_2 = 0$, P_2 sets $i = 3$). Then, P_2 sends $(\mathsf{receiver}, sid \circ j, i)$ to $\mathcal{F}_{\mathrm{OT}}^4$ and waits to receive $(sid \circ j, \gamma_2)$ from $\mathcal{F}_{\mathrm{OT}}^4$. Upon receiving this output, P_2 sets γ_2 to be its share of the output line of the gate. Thus, we have that $\gamma_1 + \gamma_2 = (\alpha_1 + \beta_1)(\alpha_2 + \beta_2)$ and the parties hold random shares of the output line of the gate.

3. **Output Stage:** Following the above stage, the parties P_1 and P_2 hold shares of all the output lines of the circuit $C_{\mathcal{F}}$. Each output line of $C_{\mathcal{F}}$ is either an output addressed to one of the parties P_1 and P_2, or belongs to the internal state of $C_{\mathcal{F}}$ after the activation. The activation concludes as follows:

   - *P_1's output:* P_2 sends P_1 all of its shares in P_1's output lines. P_1 reconstructs every bit of its output value by adding the appropriate shares. (If the actual output generated by $\mathcal{F}$ has less than the full m bits then this will be encoded in the output in a standard way.)
   - *P_2's output:* Likewise, P_1 sends P_2 all of its shares in P_2's output lines; P_2 reconstructs the value and writes it on its output tape.
   - *$\mathcal{S}$'s output:* Recall that the outputs of $\mathcal{F}$ to $\mathcal{S}$ are ignored by $C_{\mathcal{F}}$. Indeed, the protocol does not provide the real-life adversary with any information on these values. (This only strengthens the security provided by the protocol.)
   - *Internal state:* Finally, P_1 and P_2 both store the shares that they hold for the internal state lines of $C_{\mathcal{F}}$. (These shares are to be used in the next activation.)

Recall that there is no guarantee on the order of message delivery, so messages may be delivered "out of order". However, to maintain correctness, the protocol must not start some evaluation of $C_{\mathcal{F}}$ before the previous evaluation of $\mathcal{C}_{\mathcal{F}}$ has completed. Furthermore, evaluating some gate can take place only after the shares of the input lines of this gate are known. Thus, in order to guarantee that messages are processed in the correct order, a tagging method is used. Essentially, the aim of the method is to assign a *unique* tag to every message sent during all activations of $\mathcal{F}$. Thus, the adversary can gain nothing by sending messages in different orders. This is achieved in the following way. Recall that both parties hold the same session-identifier sid. Then, in activation i, the parties use the session-identifier $sid \circ i$. They also attach a tag identifying the stage which the message sent belongs to. Thus, for example, the message v_2 sent by P_1 in the input stage of the ℓ^{th} activation is tagged with $\langle sid \circ \ell \circ \mathsf{input} \rangle$. Likewise, the j^{th} call to $\mathcal{F}_{\mathrm{OT}}$ in the i^{th} activation is referenced with the session identifier $sid \circ \ell \circ j$ (and not just $sid \circ j$ as described above). Now, given the above tagging method, the ordering guarantees can be dealt with in standard ways by keeping messages that arrive too early in appropriate buffers until they become relevant (where the time that a message becomes relevant is self-evident from the labelling). By the above, it makes no difference whether or not the adversary delivers the messages according to the prescribed order. From here on we therefore assume that all messages *are* delivered in order. We also drop explicit reference to the additional tagging described here. This completes the description of $\Pi_{\mathcal{F}}$.

We now prove that the above construction securely realizes any adaptively well-formed functionality. (We stress that for the case of static adversaries, $\Pi_{\mathcal{F}}$ securely realizes any well-formed functionality, and not just those that are *adaptively* well-formed. Nevertheless, here we prove the claim only for adaptively well-formed functionalities and adaptive adversaries. The static case with security for any well-formed functionality is easily derived.)

Notice that each activation of $\Pi_{\mathcal{F}}$ is due to an input sent by one of the participating parties. This implicitly assumes that the only messages that the functionality receives are from the parties themselves. This is indeed the case for well-formed functionalities (or, more accurately, the shells of such functionalities). However, recall that in general, functionalities receive notification of the parties that are corrupted. The protocol does not (and cannot) deal with such messages and therefore does not securely realize functionalities that are not well-formed.

Claim 4.4.4 *Let $\mathcal{F}$ be a two-party adaptively well-formed functionality. Then, protocol $\Pi_{\mathcal{F}}$ securely realizes $\mathcal{F}$ in the $\mathcal{F}_{\mathrm{OT}}$-hybrid model, in the presence of semi-honest, adaptive adversaries.*

Note that the claim holds unconditionally. In fact, it holds even if the environment and the adversary are computationally unbounded. (Of course, computational assumptions are required for securely realizing the oblivious transfer functionality.) The proof below deals with the security of reactive functionalities, in the presence of adaptive adversaries. This proof is significantly more involved than an analogous claim regarding non-reactive functionalities and static adversaries. For a warm-up, we refer the reader unfamiliar with this more simple case to [44, Sec. 2.2.4].

Proof: First note that protocol $\Pi_{\mathcal{F}}$ "correctly" computes $\mathcal{F}$. That is, in each activation, if the inputs of both parties in the real-life model are identical to their inputs in the ideal process, then the outputs of the uncorrupted parties are distributed identically as their outputs in the ideal process. This fact is easily verified and follows inductively from the property that the parties always hold correct shares of the lines above the gates computed so far. (The base case of the induction relates to the fact that the parties hold correct shares of the input and internal state lines. In addition, the lines corresponding to $\mathcal{F}$'s random tape contain uniformly distributed values.)

We now proceed to show that $\Pi_{\mathcal{F}}$ securely realizes $\mathcal{F}$. Intuitively, the security of protocol $\Pi_{\mathcal{F}}$ is based on the fact that all the intermediate values seen by the parties are uniformly distributed. In particular, the shares that each party receives of the other party's input are random. Furthermore, every output that a party receives from an oblivious transfer is masked by a random bit chosen by the sender. Throughout the proof, we denote by x and y the outputs of P_1 and P_2, respectively.

Let $\mathcal{A}$ be a semi-honest, adaptive adversary that interacts with parties running Protocol $\Pi_{\mathcal{F}}$ in the $\mathcal{F}_{\mathrm{OT}}$-*hybrid model.* We construct an adversary $\mathcal{S}$ for the ideal process for $\mathcal{F}$ such that no environment $\mathcal{Z}$ can tell whether it interacts with $\mathcal{A}$ and $\Pi_{\mathcal{F}}$ in the $\mathcal{F}_{\mathrm{OT}}$-hybrid model, or with $\mathcal{S}$ in the ideal process for $\mathcal{F}$. $\mathcal{S}$ internally runs a simulated copy of $\mathcal{A}$, and proceeds as follows:

Simulating the communication with $\mathcal{Z}$: Every input value that $\mathcal{S}$ receives from $\mathcal{Z}$ is written on $\mathcal{A}$'s input tape (as if coming from $\mathcal{A}$'s environment). Likewise, every output value written by $\mathcal{A}$ on its output tape is copied to $\mathcal{S}$'s own output tape (to be read by $\mathcal{S}$'s environment $\mathcal{Z}$).

Simulation of the input stage: We first describe the simulation in the case that P_1 is corrupted (before the protocol begins). In this case, $\mathcal{S}$ holds the contents of P_1's input tape (sid, v) and therefore externally sends the value to the ideal functionality $\mathcal{F}$. Now, the input stage of $\Pi_{\mathcal{F}}$ consists only of P_1 sending a random string v_2 to P_2. In the case that P_1 is corrupted, this string is already determined and thus no further simulation is required. In the case that P_1 is not corrupted, $\mathcal{S}$ chooses a uniformly distributed string v_2 and simulates P_1 sending this string to P_2.

Simulation of the circuit evaluation stage: The computation of addition gates consists only of local computation and therefore requires no simulation. In contrast, each multiplication gate is computed using an ideal call to $\mathcal{F}_{\rm OT}$, where P_1 plays the sender and P_2 plays the receiver. We describe the simulation of these calls to $\mathcal{F}_{\rm OT}$ separately for each corruption case:

1. *Simulation when both P_1 and P_2 are not corrupted:* In this case, the only message seen by $\mathcal{A}$ in the evaluation of the $j^{\rm th}$ gate is the $(sid \circ j)$ message from the corresponding copy of $\mathcal{F}_{\rm OT}$. Thus, $\mathcal{S}$ simulates this stage by merely simulating for $\mathcal{A}$ an $(sid \circ j)$ message sent from $\mathcal{F}_{\rm OT}$ to the recipient P_2.
2. *Simulation when P_1 is corrupted and P_2 is not corrupted:* The simulation in this case is the same as in the previous (P_1 obtains no output from $\mathcal{F}_{\rm OT}$ and therefore $\mathcal{A}$ receives $(sid \circ j)$ only).
3. *Simulation when P_1 is not corrupted and P_2 is corrupted:* The receiver P_2 obtains a uniformly distributed bit γ_2 as output from each call to $\mathcal{F}_{\rm OT}$. Therefore, $\mathcal{S}$ merely chooses $\gamma_2 \in_R \{0,1\}$ and simulates P_2 receiving γ_2 from $\mathcal{F}_{\rm OT}$.
4. *Simulation when both P_1 and P_2 are corrupted:* Since all input and random tapes are already defined when both parties are corrupted, simulation is straightforward.

Simulation of the output stage: $\mathcal{S}$ simulates P_1 and P_2 sending each other their shares of the output lines. As above, we separately describe the simulation for each corruption case:

1. *Simulation when both P_1 and P_2 are not corrupted:* In this case, all $\mathcal{A}$ sees is P_1 and P_2 sending each other random m-bit strings. Therefore, $\mathcal{S}$ chooses $y_1, x_2 \in_R \{0,1\}^m$ and simulates P_1 sending y_1 to P_2 and P_2 sending x_2 to P_1 (y_1 is P_1's share in P_2's output y and vice versa for x_2).
2. *Simulation when P_1 is corrupted and P_2 is not corrupted:* First, notice that the output shares of a corrupted party are already defined (because $\mathcal{A}$ holds the view of any corrupted party and this view defines the shares in all output lines). Thus, in this case, the string sent by P_1 in the output stage is predetermined. In contrast, P_2's string is determined as follows: P_1 is corrupted and therefore $\mathcal{S}$ has P_1's output x. Furthermore, P_1's shares x_1 in its own output lines are fixed (because P_1 is corrupted). $\mathcal{S}$ therefore simulates P_2 sends $x_2 = x \oplus x_1$ to P_1 (and so P_1 reconstructs its output to x, as required).
3. *Simulation when P_1 is not corrupted and P_2 is corrupted:* The simulation here is the same as in the previous case (while reversing the roles of P_1 and P_2).
4. *Simulation when both P_1 and P_2 are corrupted:* The shares of all output lines of both parties are already determined and so simulation is straightforward.

Simulation of the first corruption: We now show how $\mathcal{S}$ simulates the first corruption of a party. Notice that this can occur at any stage after the simulation described above begins. (If the party is corrupted before the execution begins, then the simulation is according to above.) We describe the corruption as if it occurs at the end of the simulation; if it occurs earlier, then the simulator follows the instructions only until the appropriate point. We differentiate between the corruptions of P_1 and P_2:

1. *P_1 is the first party corrupted:* Upon corrupting P_1, simulator $\mathcal{S}$ receives P_1's input value v. $\mathcal{S}$ proceeds by generating the view of P_1 in the input preparation stage. Let v_2 be the message that P_1 sent P_2 in the simulation of the input stage by $\mathcal{S}$. Then, $\mathcal{S}$ sets P_1's shares of its input to v_1, where $v_1 \oplus v_2 = v$. Furthermore, $\mathcal{S}$ sets P_1's m-bit input r_1 to the lines corresponding to $C_{\mathcal{F}}$'s random tape to be a uniformly distributed string, and P_1's shares of the internal state of $\mathcal{F}$ to be a random string $c_1 \in_R \{0,1\}^m$. (Actually, if this is the first activation of $C_{\mathcal{F}}$, then c_1 is set to 0 to denote the empty state.) In addition, $\mathcal{S}$ sets P_1's random tape to be a uniformly distributed string of the appropriate length for running $\Pi_{\mathcal{F}}$. (Notice that this random tape defines the bits γ_1 that P_1 chooses when defining the oblivious transfer tables for the multiplication gates; recall that these bits then constitute P_1's shares of the output lines from these gates.) In the case that P_1 is corrupted before the output stage, this actually completes the simulation of P_1's view of the evaluation until the corruption took place. This is due to the fact that P_1 receives *no* messages during the protocol execution until the output stage (P_1 is always the oblivious transfer sender).

   We now consider the case that P_1 is corrupted after the output stage is completed. In this case the output messages x_2 and y_1 of both parties have already been sent. Thus, we must show that $\mathcal{S}$ can efficiently compute a random tape for P_1 that is consistent with these messages. For simplicity of exposition, we assume that only multiplication gates, and no addition gates, lead to output lines; any circuit can be easily modified to fulfill this requirement. Now, notice that the random coin γ_1 chosen by P_1 in any given multiplication gate is independent of all other coins. Therefore, the simulated output messages x_2, y_1 that $\mathcal{S}$ already sent only influence the coins of multiplication gates that lead to output lines; the coins of all other multiplication gates can be chosen uniformly and independently of x_2, y_1. The coins for multiplication gates leading to output lines are chosen as follows: For the i^{th} output line belonging to P_2's output, $\mathcal{S}$ sets P_1's coin γ_1 to equal the i^{th} bit of y_1. (Recall that P_1's random coin γ_1 equals its output from the gate; therefore, P_1's output from the gate equals its appropriate share in P_2's output, as required.) Furthermore, for the i^{th} output line belonging to P_1's output, $\mathcal{S}$ sets P_1's random coin γ_1 to equal the i^{th} bit of $x \oplus x_2$. (Therefore, P_1's reconstructed output equals x, as required; furthermore, this reconstructed value is independent from the intermediary information learned by the adversary.)

2. *P_2 is the first party corrupted:* Upon the corruption of P_2, simulator $\mathcal{S}$ receives P_2's output y (P_2 has no input). Then, $\mathcal{S}$ must generate P_2's view of the execution. $\mathcal{S}$ begins by choosing $r_2 \in_R \{0,1\}^m$ and setting P_2's input to the lines corresponding to $C_{\mathcal{F}}$'s random tape to r_2. In addition, it chooses the shares of the internal state of $\mathcal{F}$ to be a random string c_2. (As above, in the first activation of $C_{\mathcal{F}}$, the string c_2 is set to 0.) Next, notice that from this point on, P_2 is deterministic (and thus it needs no random tape). Also, notice that the value that P_2 receives in each oblivious transfer is uniformly distributed. Therefore, $\mathcal{S}$ simulates P_2 receiving a random bit for every oblivious transfer ($\mathcal{S}$ works in this way irrespective of when P_2 was corrupted). If this corruption occurs before the output stage has been simulated, then the above description is complete (and accurate). However, if the corruption occurs after the simulation of the output stage, then the following changes must be made. First, as above, the random bits chosen for

P_2's outputs from the oblivious transfers define P_2's shares in all the output lines. Now, if the output stage has already been simulated then the string x_2 sent by P_2 to P_1 and the string y_1 sent by P_1 to P_2 have already been fixed. Thus, as in the previous case, $\mathcal{S}$ chooses the output bits of the oblivious transfers so that they are consistent with these strings. In particular, let y be P_2's output (this is known to $\mathcal{S}$ since P_2 is corrupted) and define $y_2 = y \oplus y_1$. Then, $\mathcal{S}$ defines P_2's output-bit of the oblivious transfer that is associated with the i^{th} bit of its shares of its own output to be the i^{th} bit of y_2. Likewise, the output from the oblivious transfer associated with the i^{th} bit of P_2's share of P_1's output is set to equal the i^{th} bit of x_2.

We note that in the above description, $\mathcal{S}$ generates the corrupted party's view of the *current* activation. In addition, it must also generate the party's view for all the activations in the past. Observe that the only dependence between activations is that the shares of the state string input into a given activation equal the shares of the state string output from the preceding activation. Thus, the simulation of prior activations is exactly the case of simulation where the corruption occurs after the output stage has been completed. The only difference is that $\mathcal{S}$ defines the shares of the state string so that they are consistent between consecutive activations.

Simulation of the second corruption: As before, we differentiate between the corruptions of P_1 and P_2:

1. *P_2 is the second party corrupted:* Upon corrupting P_2, simulator $\mathcal{S}$ obtains P_2's output in this activation and all its inputs and outputs from previous activations. Furthermore, since the functionality is adaptively well-formed, $\mathcal{S}$ obtains the random tape used by the ideal functionality $\mathcal{F}$ in its computation. Next, $\mathcal{S}$ computes the internal state of $\mathcal{F}$ in this activation, based on all the past inputs, outputs and the internal random tape of $\mathcal{F}$ (this can be computed efficiently). Let c be this state string and let r equal $\mathcal{F}$'s m-bit random tape. Then, P_2 sets c_2 such that $c = c_1 \oplus c_2$, where c_1 was randomly chosen upon P_1's corruption. ($\mathcal{S}$ also makes sure that the output state information from the previous execution equals the input state information from this execution. This is easily accomplished because output gates are always immediately preceded by multiplication gates, and so independent random coins are used.) Similarly, $\mathcal{S}$ sets $r_2 = r_1 \oplus r$, where r equals $\mathcal{F}$'s random tape and r_1 equals the random string chosen upon P_1's corruption (for simulating P_1's share of the random tape of $C_{\mathcal{F}}$).

   We now proceed to the rest of the simulation. In the case we are considering here, P_1 has already been corrupted. Therefore, all the tables for the oblivious transfers have already been defined. It thus remains to show which values P_2 receives from each of these gate evaluations. However, this is immediately defined by P_2's input and the oblivious transfer tables. Thus, all the values received by P_2 from this point on, including the output values, are fully defined, and $\mathcal{S}$ can directly compute them.

2. *P_1 is the second party corrupted:* The simulation by $\mathcal{S}$ here begins in the same way as when P_2 is the second party corrupted. That is, $\mathcal{S}$ corrupts P_1 and obtains the random tape of $\mathcal{F}$. Then, $\mathcal{S}$ defines the appropriate state share string c_1, and random tape share string r_1 (in the same way as above). In addition, $\mathcal{S}$ obtains P_1's input value v and defines the appropriate share v_1 (choosing it so that $v_1 \oplus v_2 = v$). This defines all the inputs into the circuit $C_{\mathcal{F}}$. Given this information, $\mathcal{S}$ constructs the tables for all the oblivious transfers. Recall that P_2 is already corrupted. Therefore, the bits

that it receives from each oblivious transfer are already defined. Now, for each gate (working from the inputs to the outputs), $\mathcal{S}$ works as follows. Let γ_2 be the output that P_2 received from some oblivious transfer. Furthermore, assume that $\mathcal{S}$ holds the input shares of both parties for the gate in question (this can be assumed because $\mathcal{S}$ works bottom-up, from the input lines to the output lines). Then, $\mathcal{S}$ checks what the real (unmasked) output bit of the gate should be, let this value be γ. Given that P_2 received γ_2 and the output value should be γ, simulator $\mathcal{S}$ sets P_1's random-bit in defining this table to be $\gamma_1 = \gamma_2 \oplus \gamma$. $\mathcal{S}$ continues in this way for all the gates evaluated in the simulation before P_1 was corrupted. We note that if the corruption occurred after the output stage, then the output strings sent are defined by the outputs of the gates, as computed above.

Output and output delivery: $\mathcal{S}$ delivers the output from $\mathcal{F}$ to (an uncorrupted) party after $\mathcal{A}$ delivers the corresponding output message to the party in the simulation. This takes care of the outputs of corrupted parties. For a corrupted party P_i ($i \in \{1,2\}$), simulator $\mathcal{S}$ copies the contents of the simulated P_i's output tape (as written by $\mathcal{A}$) onto the output tape of the ideal process party P_i.

Analysis of $\mathcal{S}$. We show that no environment $\mathcal{Z}$ can distinguish the case where it interacts with $\mathcal{S}$ and $\mathcal{F}$ in the ideal process from the case where it interacts with $\mathcal{A}$ and $\Pi_{\mathcal{F}}$ in the $\mathcal{F}_{\mathrm{OT}}$-hybrid model. In fact, we demonstrate that $\mathcal{Z}$'s view is distributed *identically* in the two interactions.

The proof proceeds by induction on the number of activations in a run of $\mathcal{Z}$. Recall that in each activation, $\mathcal{Z}$ reads the output tapes of P_1, P_2, and the adversary, and then activates either P_1, P_2 or the adversary with some input value. (One should not confuse activations of a party, as is the intention here, with activations of the functionality and protocol.) We actually prove a somewhat stronger claim: Let ζ_i^{R} denote the random variable describing the state of $\mathcal{Z}$ at the onset of the i^{th} activation in an interaction with adversary $\mathcal{A}$ and parties running $\Pi_{\mathcal{F}}$ in the $\mathcal{F}_{\mathrm{OT}}$-hybrid model, and let α_i^{R} denote the random variable describing the state of $\mathcal{A}$ at this point in the interaction. Let ζ_i^{I} denote the random variable describing the state of $\mathcal{Z}$ at the onset of its i^{th} activation in an interaction with adversary $\mathcal{S}$ in the ideal process for functionality $\mathcal{F}$, and let α_i^{I} denote the random variable describing the state of the simulated $\mathcal{A}$ within $\mathcal{S}$ at this point in the interaction. We show that for all i, the pairs $(\zeta_i^{\mathrm{R}}, \alpha_i^{\mathrm{R}})$ and $(\zeta_i^{\mathrm{I}}, \alpha_i^{\mathrm{I}})$ are identically distributed. More precisely, Let $i > 0$. Then, for any values a_1, a_2, b_1, b_2 we show:

$$\Pr\left[(\zeta_{i+1}^{\mathrm{R}}, \alpha_{i+1}^{\mathrm{R}}) = (b_1, b_2) \,\middle|\, (\zeta_i^{\mathrm{R}}, \alpha_i^{\mathrm{R}}) = (a_1, a_2)\right] = \Pr\left[(\zeta_{i+1}^{\mathrm{I}}, \alpha_{i+1}^{\mathrm{I}}) = (b_1, b_2) \,\middle|\, (\zeta_i^{\mathrm{I}}, \alpha_i^{\mathrm{I}}) = (a_1, a_2)\right] \tag{4.3}$$

That is, assume that the states of $\mathcal{Z}$ and $\mathcal{A}$ at the onset of some activation of $\mathcal{Z}$ have some arbitrary (fixed) values a_1 and a_2, respectively. Then the joint distribution of the states of $\mathcal{Z}$ and $\mathcal{A}$ at the onset of the next activation of $\mathcal{Z}$ is the same regardless of whether we are in the "real interaction" with $\Pi_{\mathcal{F}}$, or in the ideal process. (In the interaction with $\Pi_{\mathcal{F}}$ the probability is taken over the random choices of the uncorrupted parties. In the ideal process the probability is taken over the random choices of $\mathcal{S}$ and $\mathcal{F}$.)

Asserting Eq. (4.3), recall that in the i^{th} activation $\mathcal{Z}$ first reads the output tapes of P_1, P_2, and the adversary. (We envision that these values are written on a special part of the incoming communication tape of $\mathcal{Z}$, and are thus part of its state $\zeta_i^{\mathrm{R}} = \zeta_i^{\mathrm{I}}$.) Next, $\mathcal{Z}$ either activates some uncorrupted party with some input x, or activates the adversary with input x. We treat these cases separately:

$\mathcal{Z}$ **activates an uncorrupted party with some input value** x. In this case, in the interaction with $\Pi_{\mathcal{F}}$, the activated party sends out a request to the other party to evaluate an activation of $C_{\mathcal{F}}$, plus a random share of x. This message becomes part of the state of $\mathcal{A}$ (who sees all messages sent). In the ideal process, $\mathcal{S}$ (who sees that the party sent a message to $\mathcal{F}$) generates a simulated message for $\mathcal{A}$ with the same distribution. (Recall that this message is just a uniformly distributed string.)

$\mathcal{Z}$ **activates the adversary with some input value** x. Recall that in the interaction with $\Pi_{\mathcal{F}}$ adversary $\mathcal{A}$ is now activated, reads x, and in addition has access to the messages sent by the parties and by the various copies of $\mathcal{F}_{\mathrm{OT}}$ since its last activation. (We envision that this information is written on $\mathcal{A}$'s incoming communication tape.) Next, $\mathcal{A}$ can either deliver a message to some party, modify the input/output tapes of some already corrupted party or corrupt a currently honest party. Finally, $\mathcal{A}$ writes some value on its output tape and completes its activation. In the ideal process, $\mathcal{S}$ forwards x to $\mathcal{A}$ and activates $\mathcal{A}$. Next, $\mathcal{S}$ provides $\mathcal{A}$ with additional information representing the messages sent by the parties and also, in case of party corruption, the internal state of the corrupted party.

We proceed in four steps. First, we show that the contents of $\mathcal{A}$'s incoming communication tape has the same distribution in both interactions. Second, we show that the effect of message delivery on the states of $\mathcal{A}$ and $\mathcal{Z}$ is the same in both interactions. Third, we demonstrate that the information learned by $\mathcal{A}$ upon corrupting a party has the same distribution in both interactions. Finally, we demonstrate that $\mathcal{A}$'s view regarding the states of the already corrupted parties has the same distribution in both interactions.

New messages seen by $\mathcal{A}$. Each message seen by $\mathcal{A}$ is of one of the following possible types:

- *An input-sharing message as described above:* As mentioned above, in this case in both interactions $\mathcal{A}$ sees an m-bit long random string, representing a share of the sender's new input value.
- *A message from a party to some copy of* $\mathcal{F}_{\mathrm{OT}}$*:* In this case, in both interactions, $\mathcal{A}$ only gets notified that some message was sent from the party to the copy of $\mathcal{F}_{\mathrm{OT}}$.
- *A message from some copy of* $\mathcal{F}_{\mathrm{OT}}$ *to* P_2*:* In both interactions, if P_2 is uncorrupted then $\mathcal{A}$ does not see the contents of this message. If P_2 is corrupted then this message consists of a single random bit that is independent from the states of $\mathcal{A}$ and $\mathcal{Z}$ so far. (This bit is P_2's share of the corresponding line of the circuit.)
- *An output message from one party to another:* Here one party sends its share of some output line to the other party (who is designated to get the value of this line.) In both interactions, if the recipient party is uncorrupted then this message consists of a single random bit α that is independent from the states of $\mathcal{A}$ and $\mathcal{Z}$ so far. If the recipient is corrupted then $\mathcal{A}$ already has β, the recipient's share of that line. In the interaction with $\Pi_{\mathcal{F}}$, the value $\gamma = \alpha \oplus \beta$ is the value of this output line in $C_{\mathcal{F}}$. In the ideal process, $\gamma = \alpha \oplus \beta$ is the corresponding value generated by $\mathcal{F}$. The distribution of c (given the states of $\mathcal{A}$ and $\mathcal{Z}$ so far) is identical in both cases; this is the case because $C_{\mathcal{F}}$ correctly represents an activation of $\mathcal{F}$.

Messages delivered by $\mathcal{A}$. If $\mathcal{A}$ delivers an output message to some party in an execution of $\Pi_{\mathcal{F}}$, then this party outputs the (correct) value derived from the corresponding output lines of $C_{\mathcal{F}}$. This output value, γ^{R}, becomes part of the state of $\mathcal{Z}$ (to be read by $\mathcal{Z}$ at the onset of its next activation.) If $\mathcal{A}$ delivers an output message to some party in the ideal process, then $\mathcal{S}$ (who runs $\mathcal{A}$) delivers the corresponding message from $\mathcal{F}$ to this party. Consequently, this party outputs the value, γ^{I}, sent by $\mathcal{F}$. Since $C_{\mathcal{F}}$ correctly represents the computation of $\mathcal{F}$, we have that γ^{R} and γ^{I} are identically distributed.

If $\mathcal{A}$ delivers to some party P_i a message that is not an output message then P_i outputs nothing. (P_i may send other messages, but these messages will only become part of the state of $\mathcal{A}$ in its next activation. This next activation of $\mathcal{A}$ occurs after the next activation of $\mathcal{Z}$.)

Corruption of the first party. In the interaction with $\Pi_{\mathcal{F}}$, upon corrupting the first party $\mathcal{A}$ sees all the past inputs and outputs of this party. In addition, it sees all the shares of this party from the input lines, the random input lines, the internal state input lines, and all the internal lines of the circuit $C_{\mathcal{F}}$; these shares are all random values distributed independently from the states of $\mathcal{A}$ and $\mathcal{Z}$ so far. In the ideal process, $\mathcal{S}$ provides $\mathcal{A}$ with identically distributed information.

Corruption of the second party. In the interaction with $\Pi_{\mathcal{F}}$, upon corrupting the second party $\mathcal{A}$ sees the same information as in the first corruption, namely all the past inputs and outputs of this party, as well as the shares of this party from the input lines, the random input lines, the internal state input lines, and all the internal lines of the circuit $C_{\mathcal{F}}$. Here, however, this information determines the actual values of all types of input lines to the circuit, plus the values of all the internal lines of the circuit. (The values of the random input lines to the circuit are uniformly distributed. All other lines are uniquely determined by the states of $\mathcal{Z}$ and $\mathcal{A}$ at this point.) In the ideal process, $\mathcal{S}$ provides $\mathcal{A}$ with identically distributed information. (This can be seen from the code of $\mathcal{S}$.)

This completes the analysis of $\mathcal{S}$ and the proof of the claim. ■

Using the composition theorem, Proposition 4.4.3 follows from Claims 4.4.1, 4.4.2, and 4.4.4.

4.5 Universally Composable Commitments

We describe our new universally-composable non-interactive commitment scheme. Our construction is in the common reference string model, and assumes only the existence of trapdoor permutations. (If the common reference string must come from the uniform distribution, then we actually require trapdoor permutations with dense public descriptions [27].) UC commitment schemes are protocols that securely realize the multi-session ideal commitment functionality $\mathcal{F}_{\mathrm{MCOM}}$ presented in Figure 4.4. Note that $\mathcal{F}_{\mathrm{MCOM}}$ is in fact a re-formulation of $\hat{\mathcal{F}}_{\mathrm{COM}}$, the multi-session extension of the single-session ideal commitment functionality, $\mathcal{F}_{\mathrm{COM}}$, presented in [16].

Informally speaking, in order to achieve universal composability against adaptive adversaries, a commitment scheme must have the following two properties:

- *Polynomial equivocability:* the simulator (i.e., the adversary in the ideal process) should be able to produce many commitments for which it can decommit to both 0 and 1, using the same reference string. (An additional property is actually needed for adaptive security; see below.) Of course, the real committer must be able to decommit to only a single value (as required by the binding property of commitment schemes).

- *Simulation extractability:* the simulator should be able to extract the contents of any valid commitment generated by the adversary, even after having supplied an adversary with an arbitrary number of equivocable commitments.

Functionality $\mathcal{F}_{\text{MCOM}}$

$\mathcal{F}_{\text{MCOM}}$ proceeds as follows, running with parties $P_1, \ldots, P_n$ and an adversary $\mathcal{S}$:

- **Commit Phase:** Upon receiving a message $(\mathsf{commit}, sid, ssid, P_i, P_j, b)$ from P_i, where $b \in \{0,1\}$, record the tuple $(ssid, P_i, P_j, b)$ and send the message $(\mathsf{receipt}, sid, ssid, P_i, P_j)$ to P_j and $\mathcal{S}$. Ignore any future commit messages with the same $ssid$ from P_i to P_j.
- **Reveal Phase:** Upon receiving a message $(\mathsf{reveal}, sid, ssid)$ from P_i: If a tuple $(ssid, P_i, P_j, b)$ was previously recorded, then send the message $(\mathsf{reveal}, sid, ssid, P_i, P_j, b)$ to P_j and $\mathcal{S}$. Otherwise, ignore.

Figure 4.4: The ideal commitment functionality

We remark that in the equivocable commitment protocols of [28, 29] each copy of the reference string can be used for only a single commitment. Furthermore, they are not extractable. In contrast, [16] show how to use a single copy of the reference string for multiple commitments (although they rely on specific cryptographic assumptions).

We describe our construction in phases. First we describe a new non-interactive variant of the Feige-Shamir trapdoor commitment scheme [36], which is at the heart of our construction. Then we show how to transform this scheme into one that is universally composable.

Underlying standard commitment. In our construction we use a non-interactive, perfectly binding commitment scheme with pseudorandom commitments; denote this scheme by Com. An example of such a scheme is the standard non-interactive commitment scheme based on a one-way permutation f and a hard-core predicate b of f. In order to commit to a bit σ in this scheme, one computes $Com(\sigma) = \langle f(U_k), b(U_k) \oplus \sigma \rangle$, where U_k is the uniform distribution over $\{0,1\}^k$. The Com scheme is computationally secret and produces pseudorandom commitments: that is, the distribution ensembles $\{Com(0)\}$, $\{Com(1)\}$, and $\{U_{k+1}\}$ are all computationally indistinguishable.

Non-interactive Feige-Shamir trapdoor commitments. We briefly describe a non-interactive version of the Feige-Shamir trapdoor commitment scheme [36], which is based on the zero-knowledge proof for Hamiltonicity of Blum [11]. (We are able to obtain a non-interactive version of this scheme by utilizing the common reference string.) First, we obtain a graph G (with q nodes), so that it is hard to find a Hamiltonian cycle in G within polynomial-time. This is achieved as follows: choose $x \in_R \{0,1\}^k$ and compute $y = f(x)$, where f is a one-way function. Then, use the reduction of the language $\{y \mid \exists x \text{ s.t. } y = f(x)\}$ to that of Hamiltonicity, to obtain a graph G so that finding a Hamiltonian cycle in G is equivalent to finding the preimage x of y. The one-wayness of f implies the difficulty of finding a Hamiltonian cycle in G. This graph G, or equivalently the string y, is placed in the common reference string accessible by both parties. Now, in order to commit to 0, the committer commits to a random permutation of G using the underlying commitment scheme Com (and decommits by revealing the entire graph and the permutation). In order to commit to 1, the committer commits to a graph containing a randomly labeled q-cycle only (and decommits by opening this cycle only). Note that this commitment scheme is binding because the ability to decommit to both 0 and 1 implies that the committer knows a Hamiltonian cycle in G. The important property of the [36] scheme that we use here is equivocability (or what they call the trapdoor property). That is, given a Hamiltonian cycle in G, it is possible to generate commitments that are indistinguishable from legal ones, and yet have the property that one can decommit to both a 0 and a 1. In particular, after committing to a random permutation of G, it is possible to decommit to 0 in the same way as above. However, it is also possible to decommit to 1 by only revealing the (known) Hamiltonian cycle in G.

Adaptively secure commitments. We proceed to describe our first step for obtaining adaptive security. In general, the following additional property is required of the simulator: Let c be a commitment produced by the simulator who knows a Hamiltonian cycle in G. Then, for any $b \in \{0,1\}$, the simulator should be able to produce a random string r_b, so that the honest committer, given input b and random tape r_b, outputs the commitment c. Thus, the simulator can provide an "explanation" of its actions, as if it were an honest committer. We note that the [36] scheme does not have this property.

The adaptively secure scheme, denoted aHC (for *adaptive Hamiltonian Commitment*), differs from [36] in the way commitments are generated. That is:

- To commit to a 0, the sender picks a random permutation π of the nodes of G, and commits to the entries of the adjacency matrix of the permuted graph one by one, using Com. To decommit, the sender sends π and decommits to every entry of the adjacency matrix. The receiver verifies that the graph it received is $\pi(G)$. (This is the same as in the [36] scheme.)
- To commit to a 1, the sender chooses a randomly labeled q-cycle, and for all the entries in the adjacency matrix corresponding to edges on the q-cycle, it uses Com to commit to 1 values. For all the other entries, it simply produces random values from U_{k+1} (for which it does not know the decommitment). To decommit, the sender opens only the entries corresponding to the randomly chosen q-cycle in the adjacency matrix. (This is the point where our scheme differs to that of [36]. That is, in [36] the edges that are not on the q-cycle are sent as commitments to 0. Here, random strings are sent instead.)

By the above description, the length of the random string used in order to commit to 0 is different from the length of the random string used in order to commit to 1. Nevertheless, we pad the lengths so that they are equal (the reason why this is needed will be explained below). We denote by $\mathsf{aHC}(b;r)$ a commitment of the bit b using randomness r, and by $\mathsf{aHC}(b)$ the distribution $\mathsf{aHC}(b;U_{|r|})$.

This commitment scheme has the property of being computationally secret; i.e., the distribution ensembles $\{\mathsf{aHC}(0)\}$ and $\{\mathsf{aHC}(1)\}$ are computationally indistinguishable for any graph G. Also, given the opening of any commitment to both a 0 and 1, one can extract a Hamiltonian cycle in G. Therefore, the committer cannot decommit to both 0 and 1, and the binding property holds. Finally, as with the scheme of [36], given a Hamiltonian cycle in G, a simulator can generate a commitment to 0 and later open it to both 0 and 1. (This is because the simulator knows a simple q-cycle in G itself.) Furthermore, in contrast to [36], here the simulator can also produce a random tape for the sender, explaining the commitment as a commitment to either 0 or 1. Specifically, the simulator generates each commitment string c as a commitment to 0. If, upon corruption of the sender, the simulator has to demonstrate that c is a commitment to 0 then all randomness is revealed. To demonstrate that c was generated as a commitment to 1, the simulator opens the commitments to the edges in the q-cycle and claims that all the unopened commitments are merely uniformly chosen strings (rather than commitments to the rest of G). This can be done since commitments produced by the underlying commitment scheme Com are pseudorandom. This therefore gives us polynomial equivocability, where the same reference string can be reused polynomially-many times.

Achieving simulation extractability. As discussed above, the commitment scheme aHC has the equivocability property, as required. However, a UC commitment scheme must also have the *simulation extractability* property. We must therefore modify our scheme in such a way that we add extractability without sacrificing equivocability. Simulation-extractability alone could be achieved by including a public-key for an encryption scheme secure against adaptive chosen-ciphertext attacks (CCA2) [32] into the common reference string, and having the committer send an encryption of the decommitment information along with the commitment itself. A simulator knowing the associated decryption key can decrypt and obtain the decommitment information, thereby extracting the committed value from any adversarially prepared commitment. (The reason that we use a CCA2-secure encryption scheme will become evident in the proof. Intuitively, the reason is that in the simulated interaction extracting the committed value involves ciphertext *decryptions*. Thus by interacting with the simulator the adversary essentially has access to a decryption oracle for the encryption scheme.) However, just encrypting the decommitment information destroys the equivocability of the overall scheme, since such an encryption is binding even to a simulator. In order to regain equivocability, we use encryption schemes with pseudorandom ciphertexts. This is used in the following way. Given any equivocable commitment, there are two possible decommitment strings (by the binding property, only one can be efficiently found but they both exist). The commitment is sent along with two ciphertexts: one ciphertext is an encryption of the decommitment information known to the committer and the other ciphertext is just a uniformly distributed string. In this way, equivocability is preserved because a simulator knowing both decommitment strings can encrypt them both and later claim that it only knows the decryption to one and that the other was uniformly chosen. A problem with this solutions is that there is no known CCA2-secure scheme with pseudorandom ciphertexts (and assuming only trapdoor permutations). We therefore use double encryption. That is, first the value is encrypted using a CCA2-secure scheme, which may result in a ciphertext which is not pseudorandom, and then this ciphertext is re-encrypted using an encryption scheme with pseudorandom ciphertexts. (The second scheme need only be secure against chosen plaintext attacks.)

For the CCA2-secure scheme, denoted E_{cca}, we can use any known scheme based on trapdoor permutations with the (natural) property that any ciphertext has at most one valid decryption. This property holds for all known such encryption schemes, and in particular for the scheme of [32]. For the second encryption scheme, denoted E, we use the standard encryption scheme based on trapdoor-permutations and hard-core predicates [49], where the public key is a trapdoor permutation f, and the private key is f^{-1}. Here encryption of a bit b is $f(x)$ where x is a randomly chosen element such that the hard-core predicate of x is b. Note that encryptions of both 0 and 1 are pseudorandom. The commitment scheme, called UC Adaptive Hamiltonicity Commitment UAHC, is presented in Figure 4.5.

Let $\mathcal{F}_{\text{CRS}}$ denote the common reference string functionality (that is, $\mathcal{F}_{\text{CRS}}$ provides all parties with a common, public string drawn from the distribution described in Figure 4.5). Then, we have:

Proposition 4.5.1 *Assuming the existence of trapdoor permutations, Protocol* UAHC *of Figure* 4.5 *securely realizes* $\mathcal{F}_{\text{MCOM}}$ *in the* $\mathcal{F}_{\text{CRS}}$*-hybrid model.*

Proof: Let $\mathcal{A}$ be a malicious, adaptive adversary that interacts with parties running the above protocol in the $\mathcal{F}_{\text{CRS}}$-hybrid model. We construct an ideal process adversary $\mathcal{S}$ with access to $\mathcal{F}_{\text{MCOM}}$, which simulates a real execution of Protocol UAHC with $\mathcal{A}$ such that no environment $\mathcal{Z}$ can distinguish the ideal process with $\mathcal{S}$ and $\mathcal{F}_{\text{MCOM}}$ from a real execution of UAHC with $\mathcal{A}$.

[12]As we have mentioned, the length of the random string r is the same for the case of $b = 0$ and $b = 1$. This is necessary because otherwise it would be possible to distinguish commitments merely by looking at the lengths of C_0 and C_1.

Protocol UAHC

- **Common Reference String:** The string consists of a random image y of a one-way function f (this y determines the graph G), and public-keys for the encryption schemes E and E_{cca}. (The security parameter k is implicit.)
- **Commit Phase:**
  1. On input $(\mathsf{commit}, sid, ssid, P_i, P_j, b)$ where $b \in \{0,1\}$, party P_i computes $z = \mathsf{aHC}(b;r)$ for a uniformly distributed string of the appropriate length. Next, P_i computes $C_b = E(E_{cca}(P_i, P_j, sid, ssid, r))$ using random coins s, and sets C_{1-b} to a random string of length $|C_b|$.[12] Finally, P_i records $(sid, ssid, P_j, r, s, b)$, and sends $c = (sid, ssid, P_i, z, C_0, C_1)$ to P_j.
  2. P_j receives and records c, and outputs $(\mathsf{receipt}, sid, ssid, P_i, P_j)$. P_j ignores any later commit messages from P_i with the same $(sid, ssid)$.
- **Reveal Phase:**
  1. On input $(\mathsf{reveal}, sid, ssid)$, party P_i retrieves $(sid, ssid, P_j, r, s, b)$ and sends $(sid, ssid, r, s, b)$ to P_j.
  2. Upon receiving $(sid, ssid, r, s, b)$ from P_i, P_j checks that it has a tuple $(sid, ssid, P_i, z, C_0, C_1)$. If yes, then it checks that $z = \mathsf{aHC}(b;r)$ and that $C_b = E(E_{cca}(P_i, P_j, sid, ssid, r))$, where the ciphertext was obtained using random coins s. If both these checks succeed, then P_j outputs $(\mathsf{reveal}, sid, ssid, P_i, P_j, b)$. Otherwise, it ignores the message.

Figure 4.5: The commitment protocol UAHC

Recall that $\mathcal{S}$ interacts with the ideal functionality $\mathcal{F}_{\text{MCOM}}$ and with the environment $\mathcal{Z}$. The ideal adversary $\mathcal{S}$ starts by invoking a copy of $\mathcal{A}$ and running a simulated interaction of $\mathcal{A}$ with the environment $\mathcal{Z}$ and parties running the protocol. (We refer to the interaction of $\mathcal{S}$ in the ideal process as *external interaction.* The interaction of $\mathcal{S}$ with the simulated $\mathcal{A}$ is called *internal interaction.*) We fix the following notation. First, the session and sub-session identifiers are respectively denoted by sid and $ssid$. Next, the committing party is denoted P_i and the receiving party P_j. Finally, C denotes a ciphertext generated from $E(\cdot)$, and c denotes a ciphertext generated from $E_{cca}(\cdot)$. Simulator $\mathcal{S}$ proceeds as follows:

Initialization step: The common reference string (CRS) is chosen by $\mathcal{S}$ in the following way (recall that $\mathcal{S}$ chooses the CRS for the simulated $\mathcal{A}$ by itself):

1. $\mathcal{S}$ chooses a string $x \in_R \{0,1\}^k$ and computes $y = f(x)$, where f is the specified one-way function.
2. $\mathcal{S}$ runs the key-generation algorithm for the CCA2-secure encryption scheme, obtaining a public-key E_{cca} and a secret-key D_{cca}.
3. $\mathcal{S}$ runs the key-generation algorithm for the CPA-secure encryption scheme with pseudorandom ciphertexts, obtaining a public-key E and a secret-key D.

Then, $\mathcal{S}$ sets the common reference string to equal (y, E_{cca}, E) and locally stores the triple (x, D_{cca}, D). (Recall that y defines a Hamiltonian graph G and knowing x is equivalent to knowing a Hamiltonian cycle in G.)

Simulating the communication with $\mathcal{Z}$: Every input value that $\mathcal{S}$ receives from $\mathcal{Z}$ is written on $\mathcal{A}$'s input-tape (as if coming from $\mathcal{A}$'s environment). Likewise, every output value written by $\mathcal{A}$ on its own output tape is copied to $\mathcal{S}$'s own output tape (to be read by $\mathcal{S}$'s environment $\mathcal{Z}$).

Simulating "commit" activations where the committer is uncorrupted: Upon receiving a $(\mathsf{receipt}, sid, ssid, P_i, P_j)$ message from $\mathcal{F}_{\text{MCOM}}$ when P_i is not corrupted, $\mathcal{S}$ simulates a real commit message from P_i as follows. $\mathcal{S}$ computes $z \leftarrow \mathsf{aHC}(0)$ along with two strings r_0 and r_1 such that r_b constitutes a decommitment of z to b. (As we have described, since $\mathcal{S}$ knows a Hamiltonian cycle in G, it is able to do this.) Next, $\mathcal{S}$ computes $C_0 \leftarrow E(E_{cca}(P_i, P_j, sid, ssid, r_0))$ using random coins s_0, and $C_1 \leftarrow E(E_{cca}(P_i, P_j, sid, ssid, r_1))$ using random coins s_1. Then, in the internal interaction, $\mathcal{S}$ simulates P_i sending $c = (sid, ssid, P_i, z, C_0, C_1)$ to P_j and stores (c, r_0, s_0, r_1, s_1).

Simulating "reveal" activations where the committer is uncorrupted: Upon receiving a $(\mathsf{reveal}, sid, ssid, P_i, P_j, b)$ message from $\mathcal{F}_{\text{MCOM}}$ when P_i is not corrupted, $\mathcal{S}$ generates a simulated decommitment message from the real-model P_i: this message is $(sid, ssid, r_b, s_b, b)$, where r_b and s_b are as generated in the previous item. $\mathcal{S}$ then simulates for $\mathcal{A}$ the event where P_i sends this message to P_j in the internal interaction.

Simulating corruption of parties: When $\mathcal{A}$ issues a "corrupt P_i" command in the internal (simulated) interaction, $\mathcal{S}$ first corrupts the ideal model party P_i and obtains the values of all its unopened commitments. Then, $\mathcal{S}$ prepares the internal state of P_i to be consistent with these commitment values in the same way as shown above. That is, in a real execution party P_i stores the tuple $(sid, ssid, P_j, r, s, b)$ for every commitment c. In the simulation, $\mathcal{S}$ defines the stored tuple to be $(sid, ssid, P_j, r_b, s_b, b)$ where r_b and s_b are as generated above.

Simulating "commit" activations where the committer is corrupted: When $\mathcal{A}$, controlling corrupted party P_i, sends a commitment message $(sid, ssid, P_i, z, C_0, C_1)$ to an uncorrupted party P_j in the internal (simulated) interaction, $\mathcal{S}$ works as follows. If a commitment from P_i to P_j using identifiers $(sid, ssid)$ was sent in the past, then $\mathcal{S}$ ignores the message. Otherwise, informally speaking, $\mathcal{S}$ must extract the commitment bit committed to by $\mathcal{A}$. Simulator $\mathcal{S}$ begins by decrypting both C_0 and C_1 obtaining ciphertexts c_0 and c_1 and then decrypting each of c_0 and c_1. There are three cases here:

1. *Case 1:* For some $b \in \{0,1\}$, c_b decrypts to $(P_i, P_j, sid, ssid, r)$ where r is the correct decommitment information for z as a commitment to b, and c_{1-b} does *not* decrypt to a decommitment to $1-b$. Then, $\mathcal{S}$ sends $(\mathsf{commit}, sid, ssid, P_i, P_j, b)$ to $\mathcal{F}_{\text{MCOM}}$ and stores the commitment string.
2. *Case 2:* Neither c_0 or c_1 decrypt to $(P_i, P_j, sid, ssid, r)$ where r is the appropriate decommitment information for z (and sid and $ssid$ are the correct identifiers from the commitment message). In this case, $\mathcal{S}$ sends $(\mathsf{commit}, sid, ssid, P_i, P_j, 0)$ to $\mathcal{F}_{\text{MCOM}}$. (The commitment string is not stored, since it will never be opened correctly.)
3. *Case 3:* c_0 decrypts to $(P_i, P_j, sid, ssid, r_0)$ and c_1 decrypts to $(P_i, P_j, sid, ssid, r_1)$, where r_0 is the correct decommitment information for z as a commitment to 0 and r_1 is the correct decommitment information for z as a commitment to 1. Furthermore, the identifiers in the decryption information are the same as in the commitment message. In this case, $\mathcal{S}$ outputs a special $\mathsf{failure}$ symbol and halts.

Simulating "reveal" activations where the committer is corrupted: When $\mathcal{A}$, controlling corrupted party P_i, sends a reveal message $(sid, ssid, r, s, b)$ to an uncorrupted party P_j

in the internal (simulated) interaction, $\mathcal{S}$ works as follows. $\mathcal{S}$ first checks that a tuple $(sid, ssid, P_i, z, C_0, C_1)$ is stored and that r and s constitute a proper decommitment to b. If the above holds, then $\mathcal{S}$ sends $(\mathsf{reveal}, sid, ssid, P_i, P_j)$ to $\mathcal{F}_{\text{MCOM}}$. Otherwise, $\mathcal{S}$ ignores the message.

Delivery of messages: $\mathcal{S}$ delivers a message from $\mathcal{F}_{\text{MCOM}}$ in the external interaction when $\mathcal{A}$ delivers the corresponding message in the simulated interaction. For example, $\mathcal{S}$ delivers a $\mathsf{receipt}$ message in the external interaction when $\mathcal{A}$ delivers the appropriate $(sid, ssid, P_i, z, C_0, C_1)$ message in the simulated internal interaction.

We now prove that $\mathcal{Z}$ cannot distinguish an interaction of Protocol UAHC with $\mathcal{A}$ from an interaction in the ideal process with $\mathcal{F}_{\text{MCOM}}$ and $\mathcal{S}$. In order to show this, we examine several hybrid experiments:

(I) *Real interaction:* This is the interaction of $\mathcal{Z}$ with $\mathcal{A}$ and Protocol UAHC.

(II) *Real interaction with partially fake commitments:* This is the interaction of $\mathcal{Z}$ with $\mathcal{A}$ and Protocol UAHC, except that: (i) The Hamiltonian Cycle to G is provided to all honest parties, but this information is not revealed upon corruption. (ii) In honest party commitments, a commitment to b is generated by computing $z \leftarrow \mathsf{aHC}(0)$ and strings r_0, r_1 such that r_0 and r_1 are correct decommitments to 0 and 1, respectively. (This is just like the simulator.) Then, C_b is computed as an encryption to $E(E_{cca}(P_i, P_j, sid, ssid, r_b))$. However, unlike the simulator, C_{1-b} is still chosen as a uniformly distributed string. Again, this modification is not revealed upon corruption (i.e., the honest party decommits to b as in a real interaction).

(III) *Real interaction with completely fake commitments:* This is the same as (II), except that in commitments generated by honest parties, the ciphertext C_{1-b} equals $E(E_{cca}(P_i, P_j, sid, ssid, r_{1-b}))$ as generated by $\mathcal{S}$, rather than being chosen uniformly. Commitments are opened in the same way as the simulator.

(IV) *Simulated interaction:* This is the interaction of $\mathcal{Z}$ with $\mathcal{S}$, as described above.

Our aim is to show that interactions (I) and (IV) are indistinguishable to $\mathcal{Z}$, or in other words that $\mathcal{Z}$'s output at the end of interaction (I) is deviates only negligibly from $\mathcal{Z}$'s output at the end of interaction (IV). We prove this by showing that each consecutive pair of interactions are indistinguishable to $\mathcal{Z}$. (Abusing notation, we use the term "distribution i" to denote both "interaction i", and "$\mathcal{Z}$'s output from interaction i".)

The fact that distributions (I) and (II) are computationally indistinguishable is derived from the pseudorandomness of the underlying commitment scheme aHC. This can be seen as follows. The only difference between the two distributions is that even commitments to 1 are computed by $z \leftarrow \mathsf{aHC}(0)$. However, the distribution ensembles $\{\mathsf{aHC}(0)\}$ and $\{\mathsf{aHC}(1)\}$ are indistinguishable. Furthermore, these ensembles remain indistinguishable when the decommitment information to 1 is supplied. That is, $\{\mathsf{aHC}(0), r_1\}$ and $\{\mathsf{aHC}(1), r\}$ are also indistinguishable, where r_1 is the (simulator) decommitment of $\mathsf{aHC}(0)$ to 1, and r is the (prescribed) decommitment of $\mathsf{aHC}(1)$ to 1. (A standard hybrid argument is employed to take into account the fact that many commitments and decommitments occur in any given execution.)

Next, distributions (II) and (III) are indistinguishable due to the pseudorandomness of encryptions under E. In particular, the only difference between the distributions is that in (II) the ciphertext C_{1-b} is uniformly chosen, whereas in (III) ciphertext C_{1-b} equals $E(E_{cca}(P_i, P_j, sid, ssid, r_{1-b}))$.

Intuitively, CPA security suffices because in order to emulate experiments (II) and (III), no decryption oracle is needed. In order to formally prove this claim, we use the "left-right" oracle formulation of security for encryption schemes [7]. In this formulation of security, there is a "left-right" oracle (LR-oracle) which has a randomly chosen and hidden value $b \in \{0,1\}$ built into it. When queried with a pair of plaintexts (a_0, a_1), the oracle returns $E(a_b)$. Equivalently, the oracle can be queried with a single message a such that it returns $E(a)$ if $b = 0$ and a uniformly distributed string if $b = 1$. This reflects the fact that here the security lies in the pseudorandomness of the ciphertext, rather than due to the indistinguishability of encryptions. (We stress that the LR-oracle *always* uses the same bit b.) A polynomial-time attacker is successful in this model if it succeeds in guessing the bit b with a non-negligible advantage. For chosen-plaintext security, this attacker is given access to the LR-oracle for the encryption scheme E. We now construct an adversary who carries out a chosen-plaintext attack on E and distinguishes encryptions to strings of the form $E_{cca}(P_i, P_j, sid, ssid, r_{1-b})$ from uniformly chosen strings. This adversary emulates experiments (II) and (III) by running $\mathcal{Z}$ and all the parties. However, when an honest party is supposed to generate C_{1-b}, the attacker hands the LR-oracle the query $E_{cca}(P_i, P_j, sid, ssid, r_{1-b})$ and receives back C' which either equals $E(E_{cca}(P_i, P_j, sid, ssid, r_{1-b}))$ or is uniformly distributed. The attacker then sets $C_{1-b} = C'$. This emulation can be carried out given the encryption-key E only (i.e., no decryption key is required). Now, if $b = 1$ for the LR-oracle, then the attacker perfectly emulates experiment (II). Furthermore, if $b = 0$ then the attacker perfectly emulates experiment (III). Finally, as we have mentioned, the above emulation is carried out using a chosen-plaintext attack on E only. Therefore, if $\mathcal{Z}$ can distinguish experiments (II) and (III), then the attacker can guess the bit b of the LR-oracle with non-negligible advantage. This is in contradiction to the CPA-security of E.

Finally, we consider the hybrid experiments (III) and (IV). The only difference between these experiments is that in experiment (III) the checks causing $\mathcal{S}$ to output $\mathsf{failure}$ are not carried out. We therefore conclude that it suffices to show that $\mathcal{S}$ outputs $\mathsf{failure}$ with at most negligible probability. In order to prove this, we again consider a sequence of hybrid experiments:

(V) *Simulation with partially real encryptions:* This is the same as (IV), except that $\mathcal{S}$ is given (say, by $\mathcal{F}$) the true values of the inputs for all uncorrupted parties. Then, when generating simulated commitments for uncorrupted parties, $\mathcal{S}$ replaces C_{1-b} with $E(E_{cca}(P_i, P_j, sid, ssid, 0^{|r_{1-b}|}))$, where b is the true input value.

(VI) *Simulation with nearly real commitments:* This is the same as (V), except that in the simulated commitments generated for uncorrupted parties, $\mathcal{S}$ computes $z \leftarrow \mathsf{aHC}(b)$ where b is the true value of the commitment (instead of always computing $z \leftarrow \mathsf{aHC}(0)$).

We now claim that the probability that $\mathcal{S}$ outputs $\mathsf{failure}$ in experiment (IV) is negligibly close to the probability that it outputs $\mathsf{failure}$ in experiment (V). The only difference between these experiments relates to the encryption value of C_{1-b}. The proof relies on the chosen-ciphertext security of the scheme E_{cca}. (Chosen ciphertext security is required because the emulation of experiments (IV) and (V) requires access to a decryption oracle: Recall that $\mathcal{S}$ must decrypt ciphertexts in the simulation of commit-activations where the committer is corrupted.) Formally, we prove this claim using the "left-right" oracle formulation of security for encryption schemes. Recall that according to this definition, an attacker is given access to an LR-oracle that has a randomly chosen bit b internally hardwired. The attacker can then query the oracle with pairs (a_0, a_1) and receives back $E_{cca}(a_b)$. When considering CCA2-security, the attacker is given access to the LR-oracle as well as a decryption oracle for E_{cca} which works on all ciphertexts except for those output by the LR-oracle.

We argue that if $\mathcal{S}$ outputs $\mathsf{failure}$ with probabilities that are non-negligibly far apart in experiments (IV) and (V), then $\mathcal{Z}$ together with $\mathcal{A}$ can be used to construct a successful CCA2 attacker against E_{cca} in the LR-model. We now describe the attacker. The attacker receives the public key for E_{cca}. It then simulates experiment (IV) by playing $\mathcal{Z}$, $\mathcal{A}$ and $\mathcal{S}$ as above, except for the following differences:

1. The public key for E_{cca} is given to $\mathcal{S}$ externally and $\mathcal{S}$ does not have the decryption key.

2. When generating a simulated commitment for an honest party P_i, the attacker computes $z \leftarrow \mathsf{aHC}(0)$ and decommitment strings r_0 and r_1 to 0 and 1, respectively. Furthermore, the attacker computes $C_b \leftarrow E(E_{cca}(P_i, P_j, sid, ssid, r_b))$ as $\mathcal{S}$ does. However, for C_{1-b}, the attacker queries the LR-oracle with the pair $((P_i, P_j, sid, ssid, r_{1-b}), (P_i, P_j, sid, ssid, 0^{|r_{1-b}|}))$. When the LR-oracle responds with a ciphertext c', the attacker sets $C_{1-b} \leftarrow E(c')$.

3. When $\mathcal{S}$ obtains a commitment $(sid, ssid, P_j, z, C_0, C_1)$ from $\mathcal{A}$ controlling a corrupted party P_i, then the attacker decrypts C_0 and C_1 using the decryption key for E and obtains c_0 and c_1. There are two cases:

   - *Case 1:* A ciphertext c_b came from a commitment previously generated for an honest party by $\mathcal{S}$. Now, if this generated commitment was not from P_i to P_j, then c_b cannot constitute a valid decommitment because the encryption does not contain the pair (P_i, P_j) in this order. On the other hand, if the previous commitment was from P_i to P_j, then the sub-session identifiers must be different and therefore it still cannot be a valid decommitment. (Recall that P_j will ignore a second commitment from P_i with the same identifiers.) In this case, the attacker acts just as $\mathcal{S}$ would for ciphertexts that do not decrypt to valid decommitment information. (Notice that the attacker does not need to use the decryption oracle in this case.)
   - *Case 2:* A ciphertext c_b was not previously generated by $\mathcal{S}$. Then, except with negligible probability, this ciphertext could not have been output by the LR-oracle. Therefore, the attacker can query its decryption oracle and obtain the corresponding plaintext. Given this plaintext, the attacker proceeds as $\mathcal{S}$ does.

Analyzing the success probability of the attacker, we make the following observations. If the LR-oracle uses Left encryptions (i.e., it always outputs a ciphertext c' that is an encryption of $(P_i, P_j, sid, ssid, r_{1-b})$), then the resulting simulation is negligibly close to experiment (IV). (The only difference is in the case that a ciphertext c_b generated by $\mathcal{A}$ coincides with a ciphertext output by the LR-oracle. However, this occurs with only negligible probability, otherwise E_{cca} does not provide correctness.) On the other hand, if the LR-oracle uses Right encryptions (i.e, it always outputs a ciphertext c' that is an encryption of $(P_i, P_j, sid, ssid, 0^{|r_{1-b}|})$), then the resulting simulation is negligibly close to experiment (V). Therefore, by the CCA2-security of E_{cca}, the probability that $\mathcal{Z}$ outputs 1 from experiment (IV) must be negligibly close to the probability that it outputs 1 in experiment (V). By having $\mathcal{Z}$ output 1 if and only if $\mathcal{S}$ outputs a $\mathsf{failure}$ symbol, we have that the probability that $\mathcal{S}$ outputs $\mathsf{failure}$ in the two experiments is negligibly close.

We now proceed to show that the probability that $\mathcal{S}$ outputs $\mathsf{failure}$ in experiments (V) and (VI) is negligibly close. This follows from the indistinguishability of commitments $\{\mathsf{aHC}(0)\}$ and $\{\mathsf{aHC}(1)\}$. (A standard hybrid argument is used to take into account the fact that many commitments are generated by $\mathcal{S}$ during the simulation.) Here we use the fact that in both experiments (V) and (VI) the ciphertext C_{1-b} is independent from the rest of the commitment.

Finally, to complete the proof, we show that in experiment (VI) the probability that $\mathcal{S}$ outputs $\mathsf{failure}$ is negligible. The main observation here is that in experiment (VI), $\mathcal{S}$ does not use knowledge of a Hamiltonian cycle in G. Now, if $\mathcal{S}$ outputs $\mathsf{failure}$ when simulating commit activations for a corrupted party, then this means that it obtains a decommitment to 0 and to 1 for some commitment string z. However, by the construction of the commitment scheme, this means that $\mathcal{S}$ obtains a Hamiltonian cycle (and equivalently a pre-image of $y = f(x)$). Since $\mathcal{S}$ can do this with only negligible probability we have that this event can also only occur with negligible probability. We conclude that $\mathcal{S}$ outputs failure in experiment (VI), and therefore in experiment (IV), with only negligible probability. (Formally speaking, given $\mathcal{S}$ we construct an inverter for f that proceeds as described above.) This completes the hybrid argument, demonstrating that $\mathcal{Z}$ can distinguish experiments (I) and (IV) with only negligible probability. ■

4.6 Universally Composable Zero-Knowledge

We present and discuss the ideal zero-knowledge functionality $\mathcal{F}_{\text{ZK}}$. This functionality plays a central role in our general construction of protocols for realizing any two-party functionality. Specifically, our protocol for realizing the commit-and-prove functionality is constructed and analyzed in a hybrid model with access to $\mathcal{F}_{\text{ZK}}$ (i.e., in the $\mathcal{F}_{\text{ZK}}$-hybrid model). Using the universal composition theorem, the construction can be composed with known protocols that securely realize $\mathcal{F}_{\text{ZK}}$, either in the $\mathcal{F}_{\text{MCOM}}$-hybrid model or directly in the common reference string (CRS) model, to obtain protocols for realizing any two-party functionality in the CRS model. (Actually, here we use universal composition with joint state. See more details below.)

In the zero-knowledge functionality, parameterized by a relation R, the prover sends the functionality a statement x to be proven along with a witness w. In response, the functionality forwards the statement x to the verifier if and only if $R(x, w) = 1$ (i.e., if and only if it is a correct statement). Thus, in actuality, this is a proof of knowledge in that the verifier is assured that the prover actually *knows* w (and has explicitly sent w to the functionality), rather than just being assured that such a w exists. The zero-knowledge functionality, $\mathcal{F}_{\text{ZK}}$, is presented in Figure 4.6.

Functionality $\mathcal{F}_{\text{ZK}}$

$\mathcal{F}_{\text{ZK}}$ proceeds as follows, running with a prover P, a verifier V and an adversary $\mathcal{S}$, and parameterized with a relation R:

- Upon receiving (ZK-prover, sid, x, w) from P, do: if $R(x, w) = 1$, then send (ZK-proof, sid, x) to V and $\mathcal{S}$ and halt. Otherwise, halt.

Figure 4.6: The single-session $\mathcal{F}_{\text{ZK}}$ functionality

Let us highlight several technical issues that motivate the present formalization. First, notice that the functionality is parameterized by a single relation (and thus a different copy of $\mathcal{F}_{\text{ZK}}$ is used for every different relation required). Nonetheless, the relation R may actually index any polynomial number of predetermined relations for which the prover may wish to prove statements. This can be implemented by separating the statement x into two parts: x_1 that indexes the relation to be used and x_2 that is the actual statement. Then, define $R((x_1, x_2), w) \stackrel{\text{def}}{=} R_{x_1}(x_2, w)$. (Note that in this case the set of relations to be indexed is fixed and publicly known.)[13]

[13]Another possibility is to parameterize $\mathcal{F}_{\text{ZK}}$ by a polynomial $q(\cdot)$. Then, P_i sends the functionality a triple (x, w, C_R), where C_R is a two-input binary circuit of size at most $q(|x|)$. (This circuit defines the relation being used.) The ideal functionality then sends P_j the circuit C_R and the bit $C_R(x, w)$. This approach has the advantage that the relations to be used need not be predetermined and fixed.

Second, the functionality is defined so that only correct statements (i.e., values x such that $R(x, w) = 1$) are received by P_2 in the prove phase. Incorrect statements are ignored by the functionality, and the receiver P_2 receives no notification that an attempt at cheating in a proof took place. This convention simplifies the description and analysis of our protocols. We note, however, that this is not essential. Error messages can be added to the functionality (and realized) in a straightforward way. Third, we would like to maintain the (intuitively natural) property that a prover can always cause the verifier to reject, even if for *every* w it holds that $R(x, w) = 1$ (e.g., take $R = \{0,1\}^* \times \{0,1\}^*$). This technicality is solved by defining a special witness input symbol "$\perp$" such that for every relation R and every x, $R(x, \perp) = 0$.

Note that each copy of the functionality handles only a single proof (with a given prover and a given verifier). This is indeed convenient for protocols in the $\mathcal{F}_{\text{ZK}}$-hybrid model, since a new copy of $\mathcal{F}_{\text{ZK}}$ can be invoked for each new proof (or, each "session"). However, directly realizing $\mathcal{F}_{\text{ZK}}$ in the $\mathcal{F}_{\text{CRS}}$-hybrid model and using the universal composition theorem would result in an extremely inefficient composed protocol, where a new instance of the reference string is needed for each proof. Instead, we make use of universal composition with joint state, as follows. We start by defining functionality $\hat{\mathcal{F}}_{\text{ZK}}$, the multi-session extension of $\mathcal{F}_{\text{ZK}}$, and recall known protocols that securely realize $\hat{\mathcal{F}}_{\text{ZK}}$ using a single short instance of the common string. We then use the JUC theorem to compose protocols in the $\mathcal{F}_{\text{ZK}}$-hybrid model with protocols that securely realize $\hat{\mathcal{F}}_{\text{ZK}}$.

The definition of $\hat{\mathcal{F}}_{\text{ZK}}$, the multi-session extension of $\mathcal{F}_{\text{ZK}}$, follows from the definition of $\mathcal{F}_{\text{ZK}}$ and the general definition of multi-session extensions (see Section 4.3.2). Nonetheless, for sake of clarity we explicitly present functionality $\hat{\mathcal{F}}_{\text{ZK}}$ in Figure 4.7. An input to $\hat{\mathcal{F}}_{\text{ZK}}$ is expected to contain two types of indices: the first one, sid, is the SID that differentiates messages to $\hat{\mathcal{F}}_{\text{ZK}}$ from messages sent to other functionalities. The second index, $ssid$, is the sub-session ID and is unique per "sub-session" (i.e., per input message).

Functionality $\hat{\mathcal{F}}_{\text{ZK}}$

$\hat{\mathcal{F}}_{\text{ZK}}$ proceeds as follows, running with parties $P_1, \ldots, P_n$ and an adversary $\mathcal{S}$, and parameterized with a relation R:

- Upon receiving $(\mathsf{ZK\text{-}prover}, sid, ssid, P_i, P_j, x, w)$ from P_i: If $R(x, w) = 1$, then send the message $(\mathsf{ZK\text{-}proof}, sid, ssid, P_i, P_j, x)$ to P_j and $\mathcal{S}$. Otherwise, ignore.

Figure 4.7: The multi-session zero-knowledge functionality

In the case of static adversaries, De Santis et al. present a protocol that securely realizes $\hat{\mathcal{F}}_{\text{ZK}}$, for any NP relation, in the common reference string (CRS) model [26]. This is done assuming existence of trapdoor one-way permutations. Furthermore, the protocol is "non-interactive", in the sense that it consists of a single message from the prover to the verifier. In the case of adaptive adversaries, Canetti and Fischlin show a three-round protocol that securely realizes $\hat{\mathcal{F}}_{\text{ZK}}$ in the $\mathcal{F}_{\text{MCOM}}$-hybrid model, where $\mathcal{F}_{\text{MCOM}}$ is the multi-session universally composable commitment functionality (see Section 4.5 below). The protocol uses a single copy of $\mathcal{F}_{\text{MCOM}}$.[14]

[14]Actually, the zero-knowledge functionality in [16] is only "single session" (and has some other technical differences from $\hat{\mathcal{F}}_{\text{ZK}}$). Nonetheless, it is easy to see that by using $\mathcal{F}_{\text{MCOM}}$ and having the prover first check that it's input x and w is such that $(x, w) \in R$, their protocol securely realizes $\hat{\mathcal{F}}_{\text{ZK}}$.

4.7 The Commit-and-Prove Functionality $\mathcal{F}_{\text{CP}}$

In this section we define the "commit-and-prove" functionality, $\mathcal{F}_{\text{CP}}$, and present protocols for securely realizing it. As discussed in Section 4.2, this functionality, which is a generalization of the commitment functionality, is central for constructing the protocol compiler. As in the case of $\mathcal{F}_{\text{ZK}}$, the $\mathcal{F}_{\text{CP}}$ functionality is parameterized by a relation R. The first stage is a commit phase in which the receiver obtains a commitment to some value w. The second phase is more general than plain decommitment. Rather than revealing the committed value, the functionality receives some value x from the committer, sends x to the receiver, and asserts whether $R(x, w) = 1$. To see that this is indeed a generalization of a commitment scheme, take R to be the identity relation and $x = w$. Then, following the prove phase, the receiver obtains w and is assured that this is the value that was indeed committed to in the commit phase.

In fact, $\mathcal{F}_{\text{CP}}$ is even more general, in the following ways. First it allows the committer to commit to multiple secret values w_i, and then have the relation R depend on all these values in a single proof. (This extension is later needed for dealing with reactive protocols, where inputs may be received over time.) Second, the committer may ask to prove multiple statements with respect to the same set of secret values. These generalizations are dealt with as follows. When receiving a new $(\mathsf{commit}, sid, w)$ request from the committer, $\mathcal{F}_{\text{CP}}$ adds the current w to the already existing list $\overline{w}$ of committed values. When receiving a $(\mathsf{CP\text{-}prover}, sid, x)$ request, $\mathcal{F}_{\text{CP}}$ evaluates R on x and the current list $\overline{w}$. Functionality $\mathcal{F}_{\text{CP}}$ is presented in Figure 4.8.

Functionality $\mathcal{F}_{\text{CP}}$

$\mathcal{F}_{\text{CP}}$ proceeds as follows, running with a committer C, a receiver V and an adversary $\mathcal{S}$, and is parameterized by a value k and a relation R:

Commit Phase: Upon receiving a message $(\mathsf{commit}, sid, w)$ from C where $w \in \{0,1\}^k$, append the value w to the list $\overline{w}$, and send the message $(\mathsf{receipt}, sid)$ to V and $\mathcal{S}$. (Initially, the list $\overline{w}$ is empty.)

Prove Phase: Upon receiving a message $(\mathsf{CP\text{-}prover}, sid, x)$ from C, where $x \in \{0,1\}^{\text{poly}(k)}$, compute $R(x, \overline{w})$: If $R(x, \overline{w}) = 1$, then send V and $\mathcal{S}$ the message $(\mathsf{CP\text{-}proof}, sid, x)$. Otherwise, ignore the message.

Figure 4.8: The commit-and-prove functionality

As in the case of $\mathcal{F}_{\text{ZK}}$, the $\mathcal{F}_{\text{CP}}$ functionality is defined so that only correct statements (i.e., values x such that $R(x, w) = 1$) are received by V in the prove phase. Incorrect statements are ignored by the functionality, and the receiver V receives no notification that an attempt at cheating in a proof took place.

4.7.1 Securely Realizing $\mathcal{F}_{\text{CP}}$ for static adversaries

We present protocols for securely realizing the $\mathcal{F}_{\text{CP}}$ functionality in the $\mathcal{F}_{\text{ZK}}$-hybrid model, for both static and adaptive adversaries. We first concentrate on the case of static adversaries, since it is significantly simpler than the adaptive case, and therefore serves as a good warm-up.

The commit phase and the prove phase of the protocol each involve a single invocation of $\mathcal{F}_{\text{ZK}}$. (The relation used in each phase is different.) In the commit phase the committer commits to a value using a standard commitment scheme, and proves knowledge of the decommitment value through $\mathcal{F}_{\text{ZK}}$. Thus we obtain a "commit-with-knowledge" protocol, in which the simulator can extract the committed value.

Specifically, let C be a perfectly binding commitment scheme, and denote by $C(w;r)$ a commitment to a string w using a random string r. For simplicity, we use a non-interactive commitment scheme. Such schemes exist assuming the existence of 1–1 one-way functions, see [45]. (Alternatively, we could use the Naor scheme [70] that can be based on *any* one-way function, rather than requiring 1–1 one-way functions. In this scheme, the receiver sends an initial message and then the committer commits. This changes the protocol and analysis only slightly. We note that in fact, the use of perfect binding is not essential and computational binding actually suffices, as will be the case in Section 4.7.2.) Loosely speaking, the protocol begins by C sending $c = C(w;r)$ to V, and then proving knowledge of the pair (w,r). In our terminology, this consists of the committer C sending $(\mathsf{ZK\text{-}prover}, sid_C, c = C(w;r), (w,r))$ to $\mathcal{F}_{\text{ZK}}$, which is parameterized by the following relation R_C:

$$R_C = \{(c,(w,r)) \mid c = C(w;r)\} \tag{4.4}$$

That is, R_C is the relation of pairs of commitments with their decommitment information. In addition, the committer C keeps the list $\overline{w}$ of all the values w committed to. It also keeps the lists $\overline{r}$ and $\overline{c}$ of the corresponding random values and commitment values.

When the receiver V receives $(\mathsf{ZK\text{-}proof}, sid_C, c)$ from $\mathcal{F}_{\text{ZK}}$, it accepts c as the commitment string and adds c to its list $\overline{c}$ of accepted commitments. (Note that in the $\mathcal{F}_{\text{ZK}}$-hybrid model, V is guaranteed that C "knows" the decommitment, in the sense that C explicitly sent the decommitment value to $\mathcal{F}_{\text{ZK}}$.)

The prove phase of the protocol also involves invoking $\mathcal{F}_{\text{ZK}}$ where the relation R_P parameterizing the $\mathcal{F}_{\text{ZK}}$ functionality is defined as follows. Let R be the relation parameterizing $\mathcal{F}_{\text{CP}}$. Then, R_P is defined by:

$$R_P \stackrel{\text{def}}{=} \{((x,\overline{c}),(\overline{w},\overline{r})) \mid \forall i,\ c_i = C(w_i;r_i) \ \&\ R(x,\overline{w}) = 1\} \tag{4.5}$$

That is, R_P confirms that $\overline{c}$ is the vector of commitments to $\overline{w}$, and that $R(x,\overline{w}) = 1$. Thus, the prove phase consists of the committer proving some NP-statement regarding the values committed to previously. (The value x is the NP-statement and the values committed to, plus the randomness used, comprise the "witness" for x). Upon receiving the message $(\mathsf{ZK\text{-}proof}, sid_P, (x,\overline{c}))$ from $\mathcal{F}_{\text{ZK}}$, the receiver accepts if $\overline{c}$ equals the list of commitments that it had previously received. (The receiver must check $\overline{c}$ because this is what ensures that the witness being used is indeed the list of values previously committed to, and nothing else.) Finally, note that if $R \in \mathcal{NP}$, then so too is R_P.

We denote by $\mathcal{F}^{\text{C}}_{\text{ZK}}$ and $\mathcal{F}^{\text{P}}_{\text{ZK}}$ the copies of $\mathcal{F}_{\text{ZK}}$ from the commit phase and prove phase respectively (i.e., $\mathcal{F}^{\text{C}}_{\text{ZK}}$ is parameterized by R_C and $\mathcal{F}^{\text{P}}_{\text{ZK}}$ is parameterized by R_P). Formally, the two copies of $\mathcal{F}_{\text{ZK}}$ are differentiated by using session identifiers sid_C and sid_P, respectively. (E.g., one can define $sid_C = sid\circ$'C' and $sid_P = sid\circ$'P', where sid is the session identifier of the protocol for realizing $\mathcal{F}_{\text{CP}}$ and "$\circ$" denotes concatenation.) The protocol, using a perfectly binding non-interactive commitment scheme C, is presented in Figure 4.9.

Proposition 4.7.1 *Assuming that C is a secure (perfectly binding) commitment scheme, Protocol* SCP *of Figure* 4.9 *securely realizes $\mathcal{F}_{\text{CP}}$ in the $\mathcal{F}_{\text{ZK}}$-hybrid model, for* static *adversaries.*

Proof: Let $\mathcal{A}$ be a static adversary who operates against Protocol SCP in the $\mathcal{F}_{\text{ZK}}$-hybrid model. We construct an ideal-process adversary (or simulator) $\mathcal{S}$ such that no environment $\mathcal{Z}$ can tell with non-negligible probability whether it is interacting with $\mathcal{A}$ and parties running Protocol SCP in the $\mathcal{F}_{\text{ZK}}$-hybrid model or with $\mathcal{S}$ in the ideal process for $\mathcal{F}_{\text{CP}}$. As usual, $\mathcal{S}$ will run a simulated copy of $\mathcal{A}$ and will use $\mathcal{A}$ in order to interact with $\mathcal{Z}$ and $\mathcal{F}_{\text{CP}}$. For this purpose, $\mathcal{S}$ will "simulate for

Protocol SCP

- **Auxiliary Input:** A security parameter k.
- **Commit phase:**
  1. On input $(\mathsf{commit}, sid, w)$, where $w \in \{0,1\}^k$, C chooses a random string r of length sufficient for committing to w in scheme C, and sends $(\mathsf{ZK\text{-}prover}, sid_C, C(w;r), (w,r))$ to $\mathcal{F}_{\mathrm{ZK}}^{\mathrm{C}}$, where $\mathcal{F}_{\mathrm{ZK}}^{\mathrm{C}}$ is parameterized by the relation R_C defined in Eq. (4.4). In addition, C stores in a vector $\overline{w}$ the list of all the values w that it has sent to $\mathcal{F}_{\mathrm{ZK}}$, and in vectors $\overline{r}$ and $\overline{c}$ the corresponding lists of random strings and commitment values.
  2. When receiving $(\mathsf{ZK\text{-}proof}, sid_C, c)$ from $\mathcal{F}_{\mathrm{ZK}}^{\mathrm{C}}$, V outputs $(\mathsf{receipt}, sid)$, and adds c to its list of commitments $\overline{c}$. (Initially, $\overline{c}$ is empty.)
- **Prove phase:**
  1. On input $(\mathsf{CP\text{-}prover}, sid, x)$, C sends $(\mathsf{ZK\text{-}prover}, sid_P, (x,\overline{c}), (\overline{w},\overline{r}))$ to $\mathcal{F}_{\mathrm{ZK}}^{\mathrm{P}}$, where $\overline{w},\overline{r},\overline{c}$ are the above-define vectors and $\mathcal{F}_{\mathrm{ZK}}^{\mathrm{P}}$ is parameterized by the relation R_P defined in Eq. (4.5).
  2. When receiving $(\mathsf{ZK\text{-}proof}, sid_P, (x,\overline{c}))$ from $\mathcal{F}_{\mathrm{ZK}}^{\mathrm{P}}$, V proceeds as follows. If its list of commitments equals $\overline{c}$, then it outputs $(\mathsf{CP\text{-}proof}, sid, x)$. Otherwise, it ignores the message.

Figure 4.9: A protocol for realizing $\mathcal{F}_{\mathrm{CP}}$ for static adversaries

$\mathcal{A}$" an interaction with parties running Protocol SCP, where the interaction will match the inputs and outputs seen by $\mathcal{Z}$ in its interaction with $\mathcal{S}$ in the ideal process for $\mathcal{F}_{\mathrm{CP}}$. We use the term *external communication* to refer to $\mathcal{S}$'s communication with $\mathcal{Z}$ and $\mathcal{F}_{\mathrm{CP}}$. We use the term *internal communication* to refer to $\mathcal{S}$'s communication with the simulated $\mathcal{A}$.

Recall that $\mathcal{A}$ is a static adversary and therefore the choice of which parties are under its control (i.e., corrupted) is predetermined. When describing $\mathcal{S}$, it suffices to describe its reaction to any one of the possible external activations (inputs from $\mathcal{Z}$ and messages from $\mathcal{F}_{\mathrm{CP}}$) and any one of the possible outputs or outgoing messages generated internally by $\mathcal{A}$. This is done below. For clarity, we group these activities according to whether or not the committing party C is corrupted. Simulator $\mathcal{S}$ proceeds as follows:

Simulating the communication with the environment: Every input value coming from $\mathcal{Z}$ (in the external communication) is forwarded to the simulated $\mathcal{A}$ (in the internal communication) as if coming from $\mathcal{A}$'s environment. Similarly, every output value written by $\mathcal{A}$ on its output tape is copied to $\mathcal{S}$'s own output tape (to be read by the external $\mathcal{Z}$).

Simulating the case where the committer is uncorrupted: In this case, $\mathcal{A}$ expects to see the messages sent by $\mathcal{F}_{\mathrm{ZK}}^{\mathrm{C}}$ and $\mathcal{F}_{\mathrm{ZK}}^{\mathrm{P}}$ to V. (Notice that the only messages sent in the protocol are to and from $\mathcal{F}_{\mathrm{ZK}}^{\mathrm{C}}$ and $\mathcal{F}_{\mathrm{ZK}}^{\mathrm{P}}$; therefore, the only messages seen by $\mathcal{A}$ are those sent by these functionalities. This holds regardless of whether the receiver V is corrupted or not.) In the ideal process, $\mathcal{S}$ receives the $(\mathsf{receipt},\ldots)$ and $(\mathsf{CP\text{-}proof},\ldots)$ messages that V receives from $\mathcal{F}_{\mathrm{CP}}$. It constructs the appropriate $\mathcal{F}_{\mathrm{ZK}}^{\mathrm{C}}$ messages given the $\mathsf{receipt}$ messages from $\mathcal{F}_{\mathrm{CP}}$, and the appropriate $\mathcal{F}_{\mathrm{ZK}}^{\mathrm{P}}$ messages given the $\mathsf{CP\text{-}proof}$ messages from $\mathcal{F}_{\mathrm{CP}}$. This is done as follows:

- Whenever $\mathcal{S}$ receives a message $(\mathsf{receipt}, sid)$ from $\mathcal{F}_{\mathrm{CP}}$ where C is uncorrupted, $\mathcal{S}$ computes $c = C(0^k; r)$ for a random r and (internally) passes $\mathcal{A}$ the message $(\mathsf{ZK\text{-}proof}, sid_C, c)$, as $\mathcal{A}$ would receive from $\mathcal{F}_{\mathrm{ZK}}^{\mathrm{C}}$ in a real protocol execution. Furthermore, $\mathcal{S}$ adds the value c

to its list of simulated-commitment values $\overline{c}$. (It is stressed that the commitment here is to an unrelated value, however by the hiding property of commitments and the fact that all commitments are of length k, this is indistinguishable from a real execution.)

- Whenever $\mathcal{S}$ receives a message $(\mathsf{CP\text{-}proof}, sid, x)$ from $\mathcal{F}_{\text{CP}}$ where C is uncorrupted, $\mathcal{S}$ internally passes $\mathcal{A}$ the message $(\mathsf{ZK\text{-}proof}, sid_P, (x, \overline{c}))$, as $\mathcal{A}$ would receive from $\mathcal{F}_{\text{ZK}}^{\text{P}}$ in a protocol execution, where $\overline{c}$ is the current list of commitment values generated in the simulation of the commit phase.

Simulating the case where the committer is corrupted: Here, $\mathcal{A}$ controls C and generates the messages that C sends during an execution of Protocol SCP.[15] Intuitively, in this case $\mathcal{S}$ must be able to extract the decommitment value w from $\mathcal{A}$ during the commit phase of the protocol simulation. This is because, in the ideal process, $\mathcal{S}$ must explicitly send the value w to $\mathcal{F}_{\text{CP}}$ (and must therefore know the value being committed to). Fortunately, this extraction is easy for $\mathcal{S}$ to do because $\mathcal{A}$ works in the $\mathcal{F}_{\text{ZK}}$-hybrid model, and any message sent by $\mathcal{A}$ to $\mathcal{F}_{\text{ZK}}$ is seen by $\mathcal{S}$ during the simulation. In particular, $\mathcal{S}$ obtains the ZK-proof message sent by $\mathcal{A}$ to $\mathcal{F}_{\text{ZK}}^{\text{C}}$, and this message is valid only if it explicitly contains the decommitment. The simulation is carried out as follows:

- Whenever the simulated $\mathcal{A}$ internally delivers a message of the form $(\mathsf{ZK\text{-}prover}, sid_C, c, (w, r))$ from a corrupted C to $\mathcal{F}_{\text{ZK}}^{\text{C}}$, simulator $\mathcal{S}$ checks that $c = C(w; r)$. If yes, then $\mathcal{S}$ externally sends $(\mathsf{commit}, sid, w)$ to $\mathcal{F}_{\text{CP}}$ and internally passes $(\mathsf{ZK\text{-}proof}, sid_C, c)$ to $\mathcal{A}$ (as if coming from $\mathcal{F}_{\text{ZK}}^{\text{C}}$). Furthermore, $\mathcal{S}$ adds c to its list of received commitments $\overline{c}$. Otherwise, $\mathcal{S}$ ignores the message.
- Whenever $\mathcal{A}$ internally generates a message of the form $(\mathsf{ZK\text{-}prover}, sid_P, (x, \overline{c}), (\overline{w}, \overline{r}))$ going from C to $\mathcal{F}_{\text{ZK}}^{\text{P}}$, simulator $\mathcal{S}$ acts as follows. First, $\mathcal{S}$ checks that $\overline{c}$ equals its list of commitments and that $((x, \overline{c}), (\overline{w}, \overline{r})) \in R_P$. If yes, then $\mathcal{S}$ internally passes $(\mathsf{ZK\text{-}proof}, sid_P, (x, \overline{c}))$ to $\mathcal{A}$ (as if received from $\mathcal{F}_{\text{ZK}}^{\text{P}}$) and externally sends the prover message $(\mathsf{CP\text{-}prover}, sid, x)$ to $\mathcal{F}_{\text{CP}}$. If no, then $\mathcal{S}$ does nothing.

Output delivery: It remains to describe when (if at all) $\mathcal{S}$ delivers the receipt and CP-proof messages sent to V from $\mathcal{F}_{\text{CP}}$ in the ideal process. This is decided by simply having $\mathcal{S}$ deliver a receipt message to V when $\mathcal{A}$ delivers the corresponding ZK-proof message of the commit phase in the simulation. Likewise, $\mathcal{S}$ delivers a CP-proof message to V when $\mathcal{A}$ delivers the corresponding ZK-proof message of the prove phase in the simulation.

We show that for every environment $\mathcal{Z}$ it holds that:

$$\text{IDEAL}_{\mathcal{F}_{\text{CP}}, \mathcal{S}, \mathcal{Z}} \stackrel{\text{c}}{\equiv} \text{HYBRID}^{\mathcal{F}_{\text{ZK}}}_{\text{SCP}, \mathcal{A}, \mathcal{Z}} \tag{4.6}$$

We first assert the following claim regarding the case where the committer is corrupted: the receiver V accepts a proof in the protocol execution if and only if in the ideal model simulation, V receives $(\mathsf{CP\text{-}proof}, sid, x)$ from $\mathcal{F}_{\text{CP}}$. This can be seen as follows. First, note that if $\mathcal{A}$ (controlling C) sends a ZK-prover message containing a different vector of commitments to that sent in previous commit activations, then $\mathcal{S}$ does not send any CP-prover message to $\mathcal{F}_{\text{CP}}$. Likewise, in such a case, V ignores the ZK-proof message. Simulator $\mathcal{S}$ also checks that $((x, \overline{c}), (\overline{w}, \overline{r})) \in R_P$ before sending any CP-prover message to $\mathcal{F}_{\text{CP}}$. Thus, if this does not hold, no CP-proof message is received by

[15]We assume without loss of generality that the receiver V is uncorrupted, since carrying out an interaction where both participants are corrupted bears no effect on the view of $\mathcal{Z}$.

V. Likewise, in a protocol execution, if $((x,\overline{c}),(\overline{w},\overline{r})) \notin R_P$, then V receives no $\mathsf{CP\text{-}proof}$ message. Finally, we note that by the (perfect) binding property of the commitment scheme, if $\mathcal{A}$ tries to use a different vector of witnesses than that committed to in the commit phase, then this is detected by V and $\mathcal{S}$, and the message is ignored. (By the perfect binding of the commitment scheme, the vector $\overline{c}$ defines a unique witness vector $\overline{w}$ that can be used.) We conclude that when $\mathcal{S}$ sends a $\mathsf{CP\text{-}prover}$ message to $\mathcal{F}_{\text{CP}}$ the following holds: $R(x,\overline{w}) = 1$ if and only if $R_P((x,\overline{c}),(\overline{w},\overline{r})) = 1$, where $\overline{c}$ is the vector of commitments sent by the corrupted committer. Thus, V outputs $(\mathsf{CP\text{-}proof}, sid, x)$ in a protocol execution if and only if $\mathcal{F}_{\text{CP}}$ sends $(\mathsf{CP\text{-}proof}, sid, x)$ to V in the ideal model simulation.

We proceed to demonstrate Eq. (4.6). Since $\mathcal{S}$ obtains the messages sent by $\mathcal{A}$ to both the $\mathcal{F}_{\text{ZK}}^{\text{C}}$ and $\mathcal{F}_{\text{ZK}}^{\text{P}}$ functionalities, most of the simulation is perfect and the messages seen by $\mathcal{A}$ are exactly the same as it would see in a hybrid execution of Protocol SCP. There is, however, one case where the simulation *is* different from a real execution. When the committer is uncorrupted, $\mathcal{S}$ receives a $(\mathsf{receipt}, sid)$ message from $\mathcal{F}_{\text{CP}}$ and must generate the message that $\mathcal{A}$ would see from $\mathcal{F}_{\text{ZK}}^{\text{C}}$ in the protocol. Specifically, $\mathcal{S}$ sends $(\mathsf{ZK\text{-}proof}, sid, c)$ to $\mathcal{A}$, where $c = C(0^k; r)$. That is, $\mathcal{S}$ passes $\mathcal{A}$ a commitment to a value that is unrelated to C's input. In contrast, in a real execution of Protocol SCP, the value c seen by $\mathcal{A}$ is $c = C(w; r)$, where w is C's actual input. Intuitively, by the hiding property of the commitment scheme C, these two cases are indistinguishable. Formally, assume that there exists an adversary $\mathcal{A}$, an environment $\mathcal{Z}$ and an input z to $\mathcal{Z}$, such that the IDEAL and HYBRID distributions can be distinguished. Then, we construct a distinguisher D for the commitment scheme C. That is, the distinguisher D chooses some challenge w, receives a commitment c that is either to 0^k or to w, and can tell with non-negligible probability which is the case.

Distinguisher D invokes the environment $\mathcal{Z}$, the party C and the simulator $\mathcal{S}$ (which runs $\mathcal{A}$ internally) on the following simulated interaction. First, a number i is chosen at random in $[t]$, where t is a bound on the running time of $\mathcal{Z}$. Then, for the first $i-1$ commitments c generated by $\mathcal{S}$, distinguisher D sets $c = C(0^k; r)$. When $\mathcal{S}$ is about to generate the i^{th} commitment, D declares the corresponding value w to be the challenge value, and obtains a test value c^*. (This w is the value that the simulated $\mathcal{Z}$ hands the uncorrupted committer C.) Then, $\mathcal{S}$ uses c^* as the commitment value for the i^{th} commitment. The rest of the commitments in the simulation are generated as normal commitments to the corresponding input values provided by $\mathcal{Z}$. When $\mathcal{Z}$ halts with an output value, D outputs whatever $\mathcal{Z}$ outputs and halts.

Analysis of the success probability of D is done via a standard hybrid argument and is omitted. We obtain that D succeeds in breaking the commitment with advantage p/t, where p is the advantage in which $\mathcal{Z}$ distinguishes between an interaction in the hybrid model and an interaction in the ideal process (and t is the bound on $\mathcal{Z}$'s running time). ■

On sufficient assumptions for realizing $\mathcal{F}_{\text{CP}}$: For simplicity, Protocol SCP uses a non-interactive commitment scheme, which can be based on 1–1 one-way functions. However, as we have mentioned, the commit-phase of Protocol SCP can be modified to use Naor's commitment scheme [70] (which in turn can use *any* one-way function). In this case, V begins by sending the receiver message of the [70] scheme, and then C sends the commit message, using $\mathcal{F}_{\text{ZK}}^{\text{C}}$ as in Protocol SCP. Thus, we have that $\mathcal{F}_{\text{CP}}$ can be securely realized in the $\mathcal{F}_{\text{ZK}}$-hybrid model, assuming the existence of *any* one-way function.

4.7.2 Securely Realizing $\mathcal{F}_{\text{CP}}$ for adaptive adversaries

We now present a protocol for securely realizing functionality $\mathcal{F}_{\text{CP}}$ in the $\mathcal{F}_{\text{ZK}}$-hybrid model, in the presence of adaptive adversaries. The difference between this protocol and Protocol SCP for static adversaries is in the properties of the underlying commitment scheme C in use. Essentially, here we use a commitment scheme that is "adaptively secure". That is, a simulator (having access to some trapdoor information) can generate "dummy commitments" that can later be opened in several ways.[16] In order to achieve this, the commit phase of the protocol will now involve *two* invocations of $\mathcal{F}_{\text{ZK}}$. As in the case of Protocol SCP, the relations used by the invocations of $\mathcal{F}_{\text{ZK}}$ in the commit phase are different from the relation used in the prove phase. Thus, for sake of clarity we use three different copies of $\mathcal{F}_{\text{ZK}}$, two for the commit messages and one for the prove messages.

The specific commitment scheme C used in the commit phase here is the aHC commitment that lies at the core of the universally composable commitment scheme of Section 4.5. Recall that this scheme uses a common reference string containing an image y of a one-way function f. However, here we work in the $\mathcal{F}_{\text{ZK}}$-hybrid model and do not have access to a common reference string. Thus, the common reference string is "replaced" by interaction via the $\mathcal{F}_{\text{ZK}}$ functionality. That is, the basic commitment scheme is as follows. The receiver V chooses a random value t and sends the committer C the value $s = f(t)$. Next, C uses s to commit to its input value, as defined in the aHC commitment scheme. That is, C first obtains a Hamiltonian graph G such that finding a Hamiltonian cycle in G is equivalent to computing the preimage t of s. (This is obtained by reducing the NP-language $\{s \mid \exists t \text{ s.t. } s = f(t)\}$ to Hamiltonicity.) Then, in order to commit to 0, C chooses a random permutation π of the nodes of G and commits to the edges of the permuted graph one-by-one, using a non-interactive commitment scheme Com with pseudorandom commitments. (Such a scheme can be obtained using one-way permutations, see Section 4.5.) On the other hand, in order to commit to 1, C chooses a randomly labeled cycle over the same number of nodes as in G. Then, C uses Com to commit to these entries and produces random values for the rest of the adjacency matrix. As was shown in Section 4.5, this commitment scheme is both hiding and binding, and also has the property that given a preimage t of s, it is possible to generate a "dummy commitment" that can be later explained as a commitment to both 0 and 1. (Thus, t is essentially a trapdoor.) We denote a commitment of this type by $\mathsf{aHC}_s(w; r)$, where s is the image of the one-way function being used, w is the value being committed to, and r is the randomness used in generating the commitment.

The commitment scheme aHC as described above requires that the underlying one-way function be a permutation. However, by using interaction, we can implement Com using the commitment scheme of Naor [70] (this scheme also has pseudorandom commitments). We thus obtain that aHC can be implemented using any one-way function. For simplicity, the protocol is written for Com that is non-interactive (and therefore assumes one-way permutations). However, it is not difficult to modify it so that the [70] scheme can be used instead.

As in the case of Protocol SCP, the first stage of the adaptive protocol (denoted ACP for adaptive commit-and-prove) involves carrying out the commit phase of the above-described commitment scheme via invocations of $\mathcal{F}_{\text{ZK}}$. This use of $\mathcal{F}_{\text{ZK}}$ enables the simulator to extract the committed value from the committing party. In addition, here $\mathcal{F}_{\text{ZK}}$ is also used to enable the simulator to obtain the trapdoor information needed for carrying out the adaptive simulation. Thus the protocol begins by the receiver first choosing a random string t and computing $s = f(t)$. Next, it sends

[16]The property actually required is that the simulator can generate a "commitment" c such that given any w at a later stage, it can find randomness r such that $c = C(w; r)$. This is needed for the adaptive corruption of the committing party. See Section 4.5 for more discussion.

s to the committer and, in addition, proves that it knows the preimage t. Loosely speaking, in the $\mathcal{F}_{\text{ZK}}$-hybrid model, this involves sending a $(\mathsf{ZK\text{-}prover}, sid, s, t)$ message to $\mathcal{F}_{\text{ZK}}$ and having the functionality send C the message $(\mathsf{ZK\text{-}proof}, sid, s)$ if $s = f(t)$. We note that this step is carried out only once, even if many values are later committed to. Thus, the same s is used for many commitments.

Let $\mathcal{F}_{\text{ZK}}^{\text{T}}$ denote the copy of $\mathcal{F}_{\text{ZK}}$ used for proving knowledge of the trapdoor/preimage. Then, $\mathcal{F}_{\text{ZK}}^{\text{T}}$ is parameterized by the relation R_T defined as follows:

$$R_T \stackrel{\text{def}}{=} \{(s,t) \mid s = f(t)\} \tag{4.7}$$

$\mathcal{F}_{\text{ZK}}$ is used twice more; once more in the commit phase and once in the prove phase. These copies of $\mathcal{F}_{\text{ZK}}$ are denoted $\mathcal{F}_{\text{ZK}}^{\text{C}}$ and $\mathcal{F}_{\text{ZK}}^{\text{P}}$, respectively. The uses of $\mathcal{F}_{\text{ZK}}$ here are very similar to the static case (Protocol SCP). We therefore proceed directly to defining the relations R_C and R_P that parameterize $\mathcal{F}_{\text{ZK}}^{\text{C}}$ and $\mathcal{F}_{\text{ZK}}^{\text{P}}$, respectively:

$$R_C \stackrel{\text{def}}{=} \{((s,c),(w,r)) \mid c = \mathsf{aHC}_s(w;r)\} \tag{4.8}$$

$$R_P \stackrel{\text{def}}{=} \{((x,s,\overline{c}),(\overline{w},\overline{r})) \mid \forall i, c_i = \mathsf{aHC}_s(w_i;r_i) \;\&\; R(x,\overline{w}) = 1\} \tag{4.9}$$

The only difference between the definition of R_C and R_P here and in the static case is that here the value s is included as well. This is because a *pair* (s,c) binds the sender to a single value w, whereas c by itself does not. The protocol for the adaptive case is presented in Figure 4.10. (As in the static case, we formally differentiate the copies of $\mathcal{F}_{\text{ZK}}$ by session identifiers sid_T, sid_C and sid_P.)

Proposition 4.7.2 *Assuming the existence of one-way functions, Protocol* ACP *of Figure* 4.10 *securely realizes $\mathcal{F}_{\text{CP}}$ in the $\mathcal{F}_{\text{ZK}}$-hybrid model, in the presence of adaptive adversaries.*

Proof (sketch): The proof of the above proposition follows similar lines to the proof of Proposition 4.7.1. However, here the adversary $\mathcal{A}$ can adaptively corrupt parties. Therefore, the simulator $\mathcal{S}$ must deal with instructions from $\mathcal{A}$ to corrupt parties during the simulation. When given such a "corrupt" command, $\mathcal{S}$ corrupts the ideal model party and receives its input (and possibly its output). Then, given these values, $\mathcal{S}$ must provide $\mathcal{A}$ with random coins such that the simulated transcript generated so far is consistent with this revealed input and output. (An additional "complication" here is that the binding property of C is only computational. Thus, the validity of the simulation will be demonstrated by a reduction to the binding property of C.)

More precisely, Let $\mathcal{A}$ be an adaptive adversary who operates against Protocol ACP in the $\mathcal{F}_{\text{ZK}}$-hybrid model. We construct a simulator $\mathcal{S}$ such that no environment $\mathcal{Z}$ can tell with non-negligible probability whether it is interacting with $\mathcal{A}$ and parties running Protocol ACP in the $\mathcal{F}_{\text{ZK}}$-hybrid model or with $\mathcal{S}$ in the ideal process for $\mathcal{F}_{\text{CP}}$. Simulator $\mathcal{S}$ will operate by running a simulated copy of $\mathcal{A}$ and will use $\mathcal{A}$ in order to interact with $\mathcal{Z}$ and $\mathcal{F}_{\text{CP}}$. $\mathcal{S}$ works in a similar way to the simulator in the static case (see the proof of Proposition 4.7.1), with the following changes:

1. $\mathcal{S}$ records the pair (s,t) from the initialization phase of an execution. In the case where the receiver is uncorrupted, this pair is chosen by $\mathcal{S}$ itself. In the case where the receiver is corrupted this pair is chosen by the simulated $\mathcal{A}$, and $\mathcal{S}$ obtains both s and t from the message that the corrupted receiver sends to $\mathcal{F}_{\text{ZK}}^{\text{T}}$.

2. Whenever an uncorrupted party C commits to an unknown value w, simulator $\mathcal{S}$ hands $\mathcal{A}$ a commitment to 0^k as the commitment value. More precisely, whenever $\mathcal{S}$ receives from $\mathcal{F}_{\text{CP}}$

Protocol ACP

- **Auxiliary Input:** A security parameter k, and a session identifier sid.
- **Initialization phase:**
  The first time that the committer C wishes to commit to a value using the identifier sid, parties C and V execute the following before proceeding to the commit phase:
  1. C sends sid to V to indicate that it wishes to initiate a commit activation.
  2. Upon receiving sid from C, the receiver V chooses $t \in_R \{0,1\}^k$, computes $s = f(t)$ (where f is a one-way function), and sends $(\mathsf{ZK\text{-}prover}, sid_T, s, t)$ to $\mathcal{F}_{\text{ZK}}^{\text{T}}$, where $\mathcal{F}_{\text{ZK}}^{\text{T}}$ is parameterized by the relation R_T defined in Eq. (4.7). V records the value s.
  3. Upon receiving $(\mathsf{ZK\text{-}proof}, sid_T, s)$ from $\mathcal{F}_{\text{ZK}}^{\text{T}}$, C records the value s.
- **Commit phase:**
  1. On input $(\mathsf{commit}, sid, w)$ (where $w \in \{0,1\}^k$), C computes $c = \mathsf{aHC}_s(w;r)$ for a random r and using the s it received in the initialization phase. C then sends $(\mathsf{ZK\text{-}prover}, sid_C, (s,c), (w,r))$ to $\mathcal{F}_{\text{ZK}}^{\text{C}}$, where $\mathcal{F}_{\text{ZK}}^{\text{C}}$ is parameterized by the relation R_C defined in Eq. (4.8). In addition, C stores in a vector $\overline{w}$ the list of all the values w that were sent, and in vectors $\overline{r}$ and $\overline{c}$ the corresponding lists of random strings and commitment values.
  2. Upon receiving $(\mathsf{ZK\text{-}proof}, sid_C, (s',c))$ from $\mathcal{F}_{\text{ZK}}^{\text{C}}$, V verifies that s' equals the string s that it sent in the initialization phase, outputs $(\mathsf{receipt}, sid)$ and adds the value c to its list $\overline{c}$. (Initially, $\overline{c}$ is empty.) If $s' \neq s$, then V ignores the message.
- **Prove phase:**
  1. On input $(\mathsf{CP\text{-}prover}, sid, x)$, C sends $(\mathsf{ZK\text{-}prover}, sid_P, (x,s,\overline{c}), (\overline{w},\overline{r}))$ to $\mathcal{F}_{\text{ZK}}^{\text{P}}$, where $\overline{c}$, $\overline{w}$ and $\overline{r}$ are the vectors described above, and $\mathcal{F}_{\text{ZK}}^{\text{P}}$ is parameterized by the relation R_P defined in Eq. (4.9).
  2. Upon receiving $(\mathsf{ZK\text{-}proof}, sid, (x,s',\overline{c}))$ from $\mathcal{F}_{\text{ZK}}^{\text{P}}$, V verifies that s' is the string that it sent in the initialization phase, and that its list of commitments equals $\overline{c}$. If so, then it outputs $(\mathsf{CP\text{-}proof}, sid, x)$. Otherwise, it ignores the message.

Figure 4.10: A protocol for realizing $\mathcal{F}_{\text{CP}}$ for adaptive adversaries

a message $(\mathsf{receipt}, sid)$ where C is uncorrupted, $\mathcal{S}$ computes $c = \mathsf{aHC}_s(0^k;r)$ for a random r, and hands $\mathcal{A}$ the message $(\mathsf{ZK\text{-}proof}, sid_C, (s,c))$, as if coming from $\mathcal{F}_{\text{ZK}}^{\text{C}}$. (Recall that by the aHC scheme, a commitment to 0 can be opened as either 0 or 1, given the trapdoor information t; see Section 4.5.)

3. When the simulated $\mathcal{A}$ internally corrupts C, simulator $\mathcal{S}$ first externally corrupts C in the ideal process for $\mathcal{F}_{\text{CP}}$ and obtains the vector of values $\overline{w}$ that C committed to so far. Next, $\mathcal{S}$ prepares for $\mathcal{A}$ a simulated internal state of C in Protocol ACP as follows. Apart from the vector of committed values $\overline{w}$, the only hidden internal state that C keeps in Protocol ACP is a vector of random strings $\overline{r}$ that were used to commit to each w_i in $\overline{w}$. That is, for each input value w_i in $\overline{w}$, adversary $\mathcal{A}$ expects to see a value r_i such that $c_i = \mathsf{aHC}_s(w_i, r_i)$, where c_i is the corresponding commitment value that $\mathcal{S}$ generated and handed to $\mathcal{A}$ in the simulation of commitments by an uncorrupted C (see step 2 above). Thus, for every i, $\mathcal{S}$ generates the appropriate value r_i using the trapdoor t, and then hands the list $\overline{r}$ to $\mathcal{A}$. (See Section 4.5 for a description of exactly how this randomness is generated.)

4. When the simulated $\mathcal{A}$ internally corrupts V, $\mathcal{S}$ provides $\mathcal{A}$ with a simulated internal state of V. This state consists of the preimage t, plus the messages that V receives from $\mathcal{F}_{\text{ZK}}$. All this information is available to $\mathcal{S}$.

The analysis of the above simulator is very similar to the static case (Proposition 4.7.1). The main difference is that here the commitment is only computationally binding. Thus the following bad event is potentially possible: When the committer C is corrupted, the simulated $\mathcal{A}$ commits to a series of values $\overline{w}$ with corresponding commitment values $\overline{c}$. Later, in the prove phase, $\mathcal{A}$ then generates a message $(\textsf{ZK-prover}, sid_P, (x,\overline{c}), (\overline{w}',\overline{r}'))$ to send to $\mathcal{F}_{\text{ZK}}^{\text{P}}$, where $\overline{w}' \neq \overline{w}$ and yet for every i, it holds that $c_i = \textsf{aHC}_s(w_i', r_i')$. Furthermore, $R(x,\overline{w}') = 1$ and $R(x,\overline{w}) = 0$. In other words, the bad event corresponds to a case where in the ideal process $\mathcal{F}_{\text{CP}}$ does not send a $(\textsf{CP-proof}, sid, x)$ message (because $R(x,\overline{w}) = 0$), whereas V does output such a message in a real execution of Protocol ACP (because $R_P((x,s,\overline{c})(\overline{w}',\overline{r}')) = 1$ and the vector of commitments $\overline{c}$ is as observed by V). (We note that given that this event does not occur, the correctness of the simulation carried out by $\mathcal{S}$ follows the same argument as in the proof of Proposition 4.7.1.)

We conclude the proof by showing that this bad event occurs with negligible probability, or else $\mathcal{Z}$ and $\mathcal{A}$ can be used to construct an algorithm that breaks the binding property of the $\textsf{aHC}$ commitment scheme. It suffices to show that $\mathcal{A}$ cannot generate a message $(\textsf{ZK-prover}, sid_P, (x,\overline{c}), (\overline{w}',\overline{r}'))$ where $\overline{w}' \neq \overline{w}$ and yet for every i, it holds that $c_i = \textsf{aHC}_s(w_i', r_i')$. Intuitively, this follows from the binding property of $\textsf{aHC}$ (see Section 4.5). In particular, let $\mathcal{Z}$ and $\mathcal{A}$ be such that the bad event occurs with non-negligible probability during the above ideal-process simulation by $\mathcal{S}$. Then, we construct a machine M who receives s and with non-negligible probability outputs a commitment c along with (w_1, r_1) and (w_2, r_2), where $c = \textsf{aHC}_s(w_1; r_1) = \textsf{aHC}_s(w_2; r_2)$ and $w_1 \neq w_2$.

M invokes $\mathcal{S}$ on $\mathcal{Z}$ and $\mathcal{A}$, and emulates the ideal process, while playing the roles of the ideal functionality and the uncorrupted parties C and V. Simulator $\mathcal{S}$ is the same as described above, with the following two important differences:

- Instead of $\mathcal{S}$ choosing the pair (s,t) itself in step 1 of its instructions above, it uses the value s that M receives as input. (Recall that M receives s and is attempting to contradict the binding property of the commitment relative to this s.)

- If C commits to any values before it is corrupted, the simulation is modified as follows. Instead of $\mathcal{S}$ providing $\mathcal{A}$ with a commitment $c = \textsf{aHC}_s(0^k; r)$, machine M provides $\mathcal{S}$ with the input w being committed to and then $\mathcal{S}$ provides $\mathcal{A}$ with $c = \textsf{aHC}_s(w; r)$. Upon corruption of C, simulator $\mathcal{S}$ then provides $\mathcal{A}$ directly with the list of random values $\overline{r}$ used in generating the commitments. M can give $\mathcal{S}$ these values because it plays the uncorrupted C in the emulation and therefore knows the w values.[17]

If during the emulation by M, the above-described bad event occurs, then M outputs c and the two pairs (w_1, r_1) and (w_2, r_2). In order to analyze the success probability of M, first notice that the views of $\mathcal{Z}$ and $\mathcal{A}$ in this simulation by M are indistinguishable from their views in an ideal process execution. The only difference between the executions is that $\mathcal{A}$ does not receive "dummy commitments" to 0^k, but real commitments to w. By the hiding property of the commitments, these are indistinguishable. Therefore, the probability that $\mathcal{A}$ generates the messages constituting a bad event in M's emulation is negligibly close to the probability that the bad event occurs in the ideal process simulation by $\mathcal{S}$. The key point here is that $\mathcal{S}$ does *not need to know the trapdoor t*

[17]A more "natural" definition of M would be to have it run $\mathcal{S}$ in the same way as in the simulation, even before C is corrupted. In such a case, M would need to know the trapdoor in order to proceed when C is corrupted. However, it is crucial here that M not know the trapdoor (because the binding property only holds when the trapdoor is not known).

in order to carry out the emulation. In particular, M carries out its emulation with s only, and without knowing t. Therefore the binding property of the commitment scheme aHC_s must hold with respect to M. However, by the contradicting hypothesis, the bad event occurs in M's emulation with non-negligible probability. This contradicts the binding property of the commitment scheme. ■

4.8 Two-Party Secure Computation for Malicious Adversaries

In this section, we show how to obtain universally composable general secure computation in the presence of *malicious* adversaries. Loosely speaking, we present a protocol compiler that transforms any protocol that is designed for the semi-honest adversarial model into a protocol that guarantees essentially the same behavior in the presence of malicious adversaries. The compiler is described in Section 4.8.1. Then, Section 4.8.2 ties together all the components of the construction, from Sections 4.4, 4.6, 4.7 and 4.8.1.

4.8.1 The Protocol Compiler

As discussed in Section 4.2.2, the $\mathcal{F}_{\rm CP}$ functionality is used to construct a protocol compiler that transforms any non-trivial protocol that securely realizes some two-party functionality $\mathcal{F}$ in the presence of semi-honest adversaries (e.g., the protocol of Section 4.4), into a non-trivial protocol that securely realizes $\mathcal{F}$ in the presence of malicious adversaries. In this section we present the compiler (the same compiler is valid for both static and adaptive adversaries). Now, let Π be a two-party, reactive protocol. Without loss of generality, we assume that Π works by a series of activations, where in each activation, only one of the parties has an input. This is consistent with our description of general two-party functionalities, see Section 4.3.3. For the sake of simplicity, we also assume that the lengths of the random tapes specified by Π for all activations is k.

The compiled protocol $\mathsf{Comp}(\Pi)$ is described in Figure 4.11 below. It uses two copies of $\mathcal{F}_{\rm CP}$: one for when P_1 is the committer and one for when P_2 is the committer. These copies of $\mathcal{F}_{\rm CP}$ are denoted $\mathcal{F}_{\rm CP}^1$ and $\mathcal{F}_{\rm CP}^2$, respectively, and are formally identified by session identifiers sid_1 and sid_2 (where sid_i can be taken as $sid \circ i$). The description of the compiler is from the point of view of party P_1; P_2's instructions are analogous.

Loosely speaking, the effect of the compiler on the adversary's capabilities, is that the (malicious) adversary must exhibit semi-honest behavior, or else its cheating will be detected. Recall that a semi-honest adversary follows the protocol instructions exactly, according to a fixed input and a uniformly distributed random input. The following proposition asserts that for every malicious adversary $\mathcal{A}$ participating in an execution of the compiled protocol (in the $\mathcal{F}_{\rm CP}$-hybrid model), there exists a semi-honest adversary $\mathcal{A}'$ that interacts with the original protocol in the plain real-life model such that for every environment $\mathcal{Z}$, the output distributions in these two interactions are *identical*. Thus, essentially, a malicious adversary is reduced to semi-honest behavior. We note that the compiler does not use any additional cryptographic construct other than access to $\mathcal{F}_{\rm CP}$. Consequently, the following proposition holds unconditionally, and even if the adversary and environment are computationally unbounded.

Proposition 4.8.1 *Let Π be a two-party protocol and let* $\mathsf{Comp}(\Pi)$ *be the protocol obtained by applying the compiler of Figure 4.11 to Π. Then, for every* malicious *adversary $\mathcal{A}$ that interacts with* $\mathsf{Comp}(\Pi)$ *in the $\mathcal{F}_{\rm CP}$-hybrid model there exists a* semi-honest *adversary $\mathcal{A}'$ that interacts with*

$\mathsf{Comp}(\Pi)$

Party P_1 proceeds as follows (the code for party P_2 is analogous):

1. *Random tape generation:* When activating $\mathsf{Comp}(\Pi)$ for the first time with session identifier sid, party P_1 proceeds as follows:
   (a) *Choosing a random tape for P_1:*
      i. P_1 chooses $r_1^1 \in_R \{0,1\}^k$ and sends $(\mathsf{commit}, sid_1, r_1^1)$ to $\mathcal{F}_{\text{CP}}^1$. ($P_2$ receives a $(\mathsf{receipt}, sid_1)$ message, chooses $r_1^2 \in_R \{0,1\}^k$ and sends (sid, r_1^2) to P_1.)
      ii. When P_1 receives a message (sid, r_1^2) from P_2, it sets $r_1 \stackrel{\text{def}}{=} r_1^1 \oplus r_1^2$ (r_1 will serve as P_1's random tape for the execution of Π).
   (b) *Choosing a random tape for P_2:*
      i. P_1 waits to receive a message $(\mathsf{receipt}, sid_2)$ from $\mathcal{F}_{\text{CP}}^2$ (this occurs after P_2 sends a commit message $(\mathsf{commit}, sid_2, r_2^2)$ to $\mathcal{F}_{\text{CP}}^2$). It then chooses $r_2^1 \in_R \{0,1\}^k$ and sends (sid, r_2^1) to P_2. (P_2 sets $r_2 = r_2^1 \oplus r_2^2$ to be its random tape for the execution of Π.)
2. *Activation due to new input:* When activated with input (sid, x), party P_1 proceeds as follows.
   (a) *Input commitment:* P_1 sends $(\mathsf{commit}, sid_1, x)$ to $\mathcal{F}_{\text{CP}}^1$ and adds x to the list of inputs $\overline{x}$ (this list is initially empty and contains P_1's inputs from all the previous activations of Π). Note that at this point P_2 receives the message $(\mathsf{receipt}, sid_1)$ from $\mathcal{F}_{\text{CP}}^1$.
   (b) *Protocol computation:* Let $\overline{m}_1$ be the series of Π-messages that P_1 received from P_2 in all the activations of Π until now ($\overline{m}_1$ is initially empty). P_1 runs the code of Π on its input list $\overline{x}$, messages $\overline{m}_1$, and random tape r_1 (as generated above).
   (c) *Outgoing message transmission:* For any outgoing message m that Π instructs P_1 to send to P_2, P_1 sends $(\mathsf{CP\text{-}prover}, sid_1, (m, r_1^2, \overline{m}_1))$ to $\mathcal{F}_{\text{CP}}^1$ where the relation R_Π for $\mathcal{F}_{\text{CP}}^1$ is defined as follows:

   $$R_\Pi = \left\{((m, r_1^2, \overline{m}_1), (\overline{x}, r_1^1)) \mid m = \Pi(\overline{x}, r_1^1 \oplus r_1^2, \overline{m}_1)\right\}$$

   In other words, P_1 proves that m is the correct next message generated by Π when the input sequence is $\overline{x}$, the random tape is $r_1 = r_1^1 \oplus r_1^2$ and the series of incoming Π-messages equals $\overline{m}_1$. (Recall that r_1^1 and all the elements of $\overline{x}$ were committed to by P_1 in the past using commit invocations of $\mathcal{F}_{\text{CP}}^1$, and that r_1^2 is the random string sent by P_2 to P_1 in Step 1(a)ii above.)
3. *Activation due to incoming message:* When activated with incoming message $(\mathsf{CP\text{-}proof}, sid_2, (m, r_2^1, \overline{m}_2))$ from $\mathcal{F}_{\text{CP}}^2$, P_1 first verifies that the following conditions hold. (We note that $\mathcal{F}_{\text{CP}}^2$ is parameterized by the same relation R_Π as $\mathcal{F}_{\text{CP}}^1$.)
   (a) r_2^1 is the string that P_1 sent to P_2 in Step 1(b)i above.
   (b) $\overline{m}_2$ equals the series of Π-messages received by P_2 from P_1 (i.e., P_1's outgoing messages) in all the activations until now.

   If any of these conditions fail, then P_1 ignores the message. Otherwise, P_1 appends m to its list of incoming Π-messages $\overline{m}_1$ and proceeds as in Steps 2b and 2c.
4. *Output:* Whenever Π generates an output value, $\mathsf{Comp}(\Pi)$ generates the same output value.

Implicit in the above protocol specification is the fact that P_1 and P_2 only consider messages that are associated with the specified identifier sid.

Figure 4.11: The compiled protocol $\mathsf{Comp}(\Pi)$

Π *in the plain real-life model, such that for every environment* $\mathcal{Z}$*,*

$$\text{REAL}_{\Pi,\mathcal{A}',\mathcal{Z}} \equiv \text{HYBRID}^{\mathcal{F}_{\text{CP}}}_{\mathsf{Comp}(\Pi),\mathcal{A},\mathcal{Z}}$$

An immediate corollary of this proposition is that any protocol that securely realizes some two-party functionality $\mathcal{F}$ in the semi-honest model can be compiled into a protocol that securely realizes $\mathcal{F}$ in the malicious model. This holds both for static and adaptive adversaries.

Corollary 4.8.2 *Let* $\mathcal{F}$ *be a two-party functionality and let* Π *be a non-trivial protocol that securely realizes* $\mathcal{F}$ *in the real-life model and in the presence of semi-honest adversaries. Then* $\mathsf{Comp}(\Pi)$ *is a non-trivial protocol that securely realizes* $\mathcal{F}$ *in the* $\mathcal{F}_{\text{CP}}$*-hybrid model and in the presence of malicious adversaries.*

We note that the proposition and corollary hold both for the case of adaptive adversaries and for the case of static adversaries. Here we prove the stronger claim, relating to adaptive adversaries. We now prove the proposition.

Proof of Proposition 4.8.1: Intuitively, a malicious adversary cannot cheat because the validity of each message that it sends is verified using the $\mathcal{F}_{\text{CP}}$ functionality. Therefore, it has no choice but to play in a semi-honest manner (or be detected cheating).

More precisely, let $\mathcal{A}$ be a malicious adversary interacting with $\mathsf{Comp}(\Pi)$ in the $\mathcal{F}_{\text{CP}}$-hybrid model. We construct a semi-honest adversary $\mathcal{A}'$ that interacts with Π in the plain real-life model, such that no environment $\mathcal{Z}$ can tell whether it is interacting with $\mathsf{Comp}(\Pi)$ and $\mathcal{A}$ in the $\mathcal{F}_{\text{CP}}$-hybrid model, or with Π and $\mathcal{A}'$ in the plain real-life model. As usual, $\mathcal{A}'$ works by running a simulated copy of $\mathcal{A}$ and using the messages sent by $\mathcal{A}$ as a guide for its interaction with Π and $\mathcal{Z}$. We use the term *external communication* to refer to the communication of $\mathcal{A}'$ with $\mathcal{Z}$ and Π. The term *internal communication* is used to refer to the communication of $\mathcal{A}'$ with the simulated $\mathcal{A}$. Before describing $\mathcal{A}'$, we note the difference between this proof and all previous ones in this paper. Until now, we constructed an *ideal process* adversary $\mathcal{S}$ from a hybrid or real model adversary $\mathcal{A}$. In contrast, here we construct a *real model* adversary $\mathcal{A}'$ from a hybrid model adversary $\mathcal{A}$. Furthermore, previously both $\mathcal{S}$ and $\mathcal{A}$ were malicious adversaries, whereas here $\mathcal{A}$ is malicious and $\mathcal{A}'$ is semi-honest. We now describe $\mathcal{A}'$:

First, $\mathcal{A}'$ runs a simulated copy of $\mathcal{A}$ and simulates for $\mathcal{A}$ the $\mathsf{Comp}(\Pi)$ messages relating to the generation of the random string of both parties. Next, $\mathcal{A}'$ translates each message externally sent in Π to the corresponding message (or sequence of messages) in $\mathsf{Comp}(\Pi)$. Each message sent by the simulated $\mathcal{A}$ (supposedly by a corrupted party running $\mathsf{Comp}(\Pi)$) is translated back to a Π-message and sent externally. The rationale of this behavior is that if the simulated $\mathcal{A}$ (controlling the corrupted party) deviates from the protocol, then this would have been detected by the partner in $\mathsf{Comp}(\Pi)$, and thus $\mathcal{A}'$ can ignore that message. If $\mathcal{A}$ does not deviate from the protocol, then $\mathcal{A}'$ can forward the messages sent by $\mathcal{A}$ to the other party as this is allowed behavior for a semi-honest party. More precisely, $\mathcal{A}'$ proceeds as follows.

Simulating the communication with the environment: Every input value coming from $\mathcal{Z}$ (in the external communication) is forwarded to the simulated $\mathcal{A}$ (in the internal communication) as if coming from $\mathcal{A}$'s environment. Similarly, every output value written by $\mathcal{A}$ on its output tape is copied to $\mathcal{A}'$'s own output tape (to be read by the external $\mathcal{Z}$).

Simulating the "random tape generation" phase: When the first activation of Π takes place, $\mathcal{A}'$ internally simulates the "random tape generation" phase of $\mathsf{Comp}(\Pi)$. Here we separately deal with each corruption case:

1. *Both parties are not corrupted:* $\mathcal{A}'$ simulates both parties' messages from this stage. That is, in order to simulate the generation of P_1's random tape, $\mathcal{A}'$ internally passes $\mathcal{A}$ the message $(\mathsf{receipt}, sid_1)$, as if coming from $\mathcal{F}^1_{\text{CP}}$. Furthermore, $\mathcal{A}'$ chooses a random r_1^2, records the value, and simulates P_2 sending P_1 the message (sid, r_1^2) of Step 1(a)ii in Figure 4.11. The simulation of P_2's random tape is analogous.
2. P_1 *is not corrupted and* P_2 *is corrupted:* We begin with the generation of P_1's random tape. As above, $\mathcal{A}'$ begins by internally passing $\mathcal{A}$ the message $(\mathsf{receipt}, sid_1)$, as if coming from $\mathcal{F}^1_{\text{CP}}$. Then, $\mathcal{A}'$ obtains and records the message (sid, r_1^2) from the corrupted P_2 (controlled by $\mathcal{A}$ in $\mathsf{Comp}(\Pi)$).

   We now proceed to the generation of P_2's random tape. $\mathcal{A}'$ obtains from $\mathcal{A}$ the message $(\mathsf{commit}, sid_2, r_2^2)$, as sent by P_2 to $\mathcal{F}^2_{\text{CP}}$ in an execution of $\mathsf{Comp}(\Pi)$. Now, let r_2 equal the random tape of the corrupted P_2 in the external execution of Π ($\mathcal{A}'$ knows this value because it can read all of the corrupted P_2's tapes). Then, $\mathcal{A}'$ sets $r_2^1 = r_2 \oplus r_2^2$ and internally passes $\mathcal{A}$ the message (sid, r_2^1), as if sent by P_1 to P_2. (Recall that $\mathcal{A}'$ is semi-honest and thus it cannot modify P_2's random tape r_2 for Π. $\mathcal{A}'$ therefore "forces" $\mathcal{A}$ to use this exact same random tape in the simulated execution of $\mathsf{Comp}(\Pi)$.)
3. P_1 *is corrupted and* P_2 *is not corrupted:* The simulation of this case is analogous to the previous one. In particular, for the generation of the corrupted P_1's random tape, $\mathcal{A}'$ first receives a message $(\mathsf{commit}, sid_1, r_1^1)$ from $\mathcal{A}$ and simulates P_2 sending (sid, r_1^2) to P_1, where $r_1^2 = r_1 \oplus r_1^1$ and r_1 equals the random tape of the real party P_1 executing Π.
4. *Both parties are corrupted:* When both parties are corrupted, the entire simulation is straightforward. ($\mathcal{A}'$ simply runs both malicious parties and at the end, copies the contents of their output tapes to the output tapes of the semi-honest parties running Π.) We therefore ignore this case from now on.

Simulating an activation due to new input: We deal with the case that P_1 is not corrupted separately from the case that P_1 is corrupted. Recall that the input commitment phase consists only of P_1 sending a commit message to $\mathcal{F}^1_{\text{CP}}$. First, if party P_1 is not corrupted, then $\mathcal{A}'$ learns that the external P_1 received new input from the fact that it sends its first message of the execution of Π. In response, $\mathcal{A}'$ simulates the input commitment step by internally passing $(\mathsf{receipt}, sid_1)$ to $\mathcal{A}$ (as $\mathcal{A}$ expects to receive from $\mathcal{F}^1_{\text{CP}}$ in a real execution of $\mathsf{Comp}(\Pi)$).

If P_1 *is* corrupted, then $\mathcal{A}'$ receives a message $(\mathsf{commit}, sid_1, x)$ from $\mathcal{A}$ (who controls P_1 in $\mathsf{Comp}(\Pi)$). Then, $\mathcal{A}'$ adds x to the list $\overline{x}$ of inputs committed to by P_1 and passes $\mathcal{A}$ the string $(\mathsf{receipt}, sid_1)$, as if coming from $\mathcal{F}^1_{\text{CP}}$. Furthermore, $\mathcal{A}'$ sets P_1's input tape to equal x. (Recall that a semi-honest adversary is allowed to modify the input values that the environment writes on the input tape of a corrupted party. Formally, when the environment $\mathcal{Z}$ notifies the semi-honest $\mathcal{A}'$ of the value that it wishes to write on P_1's input tape, $\mathcal{A}'$ simulates for $\mathcal{A}$ the malicious model where $\mathcal{Z}$ writes directly to P_1's input tape. Then, when $\mathcal{A}$ sends the message $(\mathsf{commit}, sid_1, x)$ in the simulation, $\mathcal{A}'$ externally instructs $\mathcal{Z}$ to write the value x (as committed to by $\mathcal{A}$) on P_1's input tape. See Section 4.3.1 for an exact description of how values are written to the parties' input tapes in the semi-honest model.)

Dealing with protocol messages sent externally by uncorrupted parties: If an uncorrupted party P_1 externally sends P_2 a message m in the execution of Π, then $\mathcal{A}'$ internally passes $\mathcal{A}$

the message $(\textsf{CP-proof}, sid_1, (m, r_1^2, \overline{m}_1))$, where r_1^2 is the value recorded by $\mathcal{A}'$ in the simulated generation of P_1's random tape above, and $\overline{m}_1$ is the series of all Π-messages received by P_1 so far. Similarly, if an uncorrupted party P_2 sends P_1 a message m in the execution of Π, then $\mathcal{A}'$ internally passes $\mathcal{A}$ the message $(\textsf{CP-proof}, sid_2, (m, r_2^1, \overline{m}_2))$, where r_2^1 and $\overline{m}_2$ are the analogous values to the previous case.

Next, $\mathcal{A}'$ externally delivers messages sent from P_1 to P_2 (resp., from P_2 to P_1) in the execution of Π when $\mathcal{A}$ delivers the corresponding $(\textsf{CP-proof}, \ldots)$ message from $\mathcal{F}_{\text{CP}}^1$ to P_2 (resp., from $\mathcal{F}_{\text{CP}}^2$ to P_1) in the simulated execution of $\textsf{Comp}(\Pi)$.

Dealing with protocol messages sent internally by corrupted parties: Assume that P_1 is corrupted. If $\mathcal{A}$, controlling P_1, sends a message $(\textsf{CP-prover}, sid_1, (m, {r'}_1^2, \overline{m}'_1))$, then $\mathcal{A}'$ works as follows. First, $\mathcal{A}'$ has seen all the messages $\overline{m}_1$ received by P_1 and can check that $\overline{m}'_1 = \overline{m}_1$. Likewise, $\mathcal{A}'$ checks that ${r'}_1^2 = r_1^2$ (recall that r_1^2 is the value recorded by $\mathcal{A}'$ in the simulated generation of P_1's random tape above). Finally, $\mathcal{A}'$ checks that $m = \Pi(\overline{x}, r_1^1 \oplus r_1^2, \overline{m}_1)$. (Notice that since P_1 is corrupted, $\mathcal{A}'$ has all the necessary information to carry out these checks.) If all of the above is true, then $\mathcal{A}'$ internally passes $\mathcal{A}$ the message $(\textsf{CP-proof}, sid_1, (m, {r'}_1^2, \overline{m}'_1))$, as $\mathcal{A}$ expects to receive from $\mathcal{F}_{\text{CP}}^1$. Then, when $\mathcal{A}$ delivers this $(\textsf{CP-proof}, \ldots)$ message from $\mathcal{F}_{\text{CP}}^1$ to P_2 in the simulation, $\mathcal{A}$ externally delivers the message that P_1, running Π, has written on its outgoing communication tape for P_2.[18] If any of these checks fail, then $\mathcal{A}'$ does nothing. (That is, no message is externally delivered from P_1 to P_2 at this point.) The case of $\mathcal{A}$ sending a $\textsf{CP-proof}$ message in the name of a corrupt P_2 is analogous.

Dealing with corruption of parties:[19] When the simulated $\mathcal{A}$ internally corrupts a party P_1, $\mathcal{A}'$ first externally corrupts P_1 and obtains all of P_1's past inputs and outputs, and its random tape. Next, $\mathcal{A}'$ prepares for $\mathcal{A}$ a simulated internal state of P_1 in protocol $\textsf{Comp}(\Pi)$. This is done as follows. The only additional internal state that P_1 keeps in $\textsf{Comp}(\Pi)$ is the random string r_1^1 (this is the string that P_1 commits to in the random tape generation phase of $\textsf{Comp}(\Pi)$). Then, $\mathcal{A}'$ sets $r_1^1 = r_1 \oplus r_1^2$, where r_1 is P_1's random string for Π and r_1^2 is the string that P_2 sent to P_1 in the internal simulated interaction with $\mathcal{A}$ of $\textsf{Comp}(\Pi)$. Thus, in the above way, $\mathcal{A}'$ prepares a simulated internal state of P_1 in $\textsf{Comp}(\Pi)$ and internally passes it to $\mathcal{A}$. $\mathcal{A}'$ works in an analogous way upon the corruption of P_2.

We argue that $\mathcal{Z}$'s view of the interaction with $\mathcal{A}'$ and parties running Π in the real-life semi-honest model is *identical* to its view of the interaction with $\mathcal{A}$ and parties running $\textsf{Comp}(\Pi)$ in the $\mathcal{F}_{\text{CP}}$-hybrid model. (In particular, the view of the simulated $\mathcal{A}$ within $\mathcal{A}'$ is identical to its view in a real interaction with the same $\mathcal{Z}$ and $\textsf{Comp}(\Pi)$ in the $\mathcal{F}_{\text{CP}}$-hybrid model.) This can be seen by observing the computational steps in an interaction of $\mathcal{Z}$ with $\mathcal{A}'$ and Π. The cases where an uncorrupted party sends a message are immediate. To see that this holds also in the case where $\mathcal{A}'$ delivers messages sent by corrupted parties, recall that $\mathcal{A}'$ forces the random input of a corrupted P_1 in the internal execution of $\textsf{Comp}(\Pi)$ with $\mathcal{A}$ to be the random tape of the semi-honest party P_1 externally executing Π. Furthermore, $\mathcal{A}'$ modifies the input tape of the external party P_1 so that it is the same input as committed to by $\mathcal{A}$. We therefore have that the input and random tapes that malicious $\mathcal{A}$ committed to for the internal P_1 are exactly the same as the input and random tapes used by the external P_1. Now, $\mathcal{A}'$ obtains all the inputs committed to by a corrupted

[18] This point requires some elaboration. Notice that if all checks were successful, then the message that P_1 would send in an execution of Π equals m. This is because external P_1 in Π and internal P_1 in $\textsf{Comp}(\Pi)$ both have the same inputs, random tapes and series of incoming messages. Therefore, their outgoing messages are also the same.

[19] In the case of static adversaries the simulation remains the same with the exception that this case is ignored.

P_1 in the simulated interaction with $\mathcal{A}$. Consequently, $\mathcal{A}'$ is able to verify at every step if the message m sent by $\mathcal{A}$, in the name of a corrupted P_1 in the simulated interaction, is according to the protocol specification. If yes, then we are guaranteed that P_1 generates the exact same message m in the external execution of Π. Thus, P_2 receives the same Π-message in the execution of Π (where the adversary $\mathcal{A}'$ is semi-honest) and in the execution of $\mathsf{Comp}(\Pi)$ (where the adversary $\mathcal{A}$ is malicious). Furthermore, we are guaranteed that whenever $\mathcal{A}'$ delivers a message m in the external execution of Π, the simulated $\mathcal{A}$ generated and delivered a valid corresponding message to $\mathcal{F}_{\text{CP}}$. Finally, the internal state that $\mathcal{A}$ receives from $\mathcal{A}'$ upon corrupting a party is exactly the same as it receives in a real execution of $\mathsf{Comp}(\Pi)$. In particular, observe that in the simulation of the random tape generation phase when P_1 is not corrupted, $\mathcal{A}$ receives no information about r_1^1 (it only sees a $(\mathsf{receipt}, sid_1)$ message). Therefore, $\mathcal{A}'$ can choose r_1^1 as any value that it wishes upon the corruption of P_1, and in particular it can set it to equal $r_1 \oplus r_1^2$ (recall that P_1 indeed uses the random tape r_1; therefore this is consistent with its true internal state). We conclude that the ensembles REAL and HYBRID are identical. ■

4.8.2 Conclusions

Combining Proposition 4.4.3 and Corollary 4.8.2, we have that for any two-party ideal functionality $\mathcal{F}$, there exists a protocol that securely realizes $\mathcal{F}$ in the $\mathcal{F}_{\text{CP}}$-hybrid model (in the presence of malicious adversaries). Combining this with Proposition 4.7.1, and using the universal composition theorem (Theorem 4.3.3), we obtain universally composable general two-party computation in the $\mathcal{F}_{\text{ZK}}$-hybrid model. That is,

Theorem 4.8.3 (Theorem 4.2.2 – formally restated): *Assume that trapdoor permutations exist. Then, for any well-formed two-party ideal functionality $\mathcal{F}$, there exists a non-trivial protocol that securely realizes $\mathcal{F}$ in the $\mathcal{F}_{\text{ZK}}$-hybrid model in the presence of* malicious, static *adversaries. Furthermore, if two-party augmented non-committing encryption protocols exist, then for any adaptively well-formed two-party ideal functionality $\mathcal{F}$, there exists a non-trivial protocol that securely realizes $\mathcal{F}$ in the $\mathcal{F}_{\text{ZK}}$-hybrid model in the presence of* malicious, adaptive *adversaries.*

Recall that, under the assumption that trapdoor permutations exist, functionality $\hat{\mathcal{F}}_{\text{ZK}}$ (the multi-session extension of $\mathcal{F}_{\text{ZK}}$) can be securely realized in the $\mathcal{F}_{\text{CRS}}$-hybrid model by protocols that uses a single copy of the reference string. We can thus use the universal composition with joint state theorem (Theorem 4.3.4) to obtain the following corollary:

Corollary 4.8.4 *Assume that trapdoor permutations exist. Then, for any well-formed two-party ideal functionality $\mathcal{F}$, there exists a non-trivial protocol that securely realizes $\mathcal{F}$ in the $\mathcal{F}_{\text{CRS}}$-hybrid model in the presence of* malicious, static *adversaries. Furthermore, if two-party augmented non-committing encryption protocols exist, then for any adaptively well-formed two-party functionality $\mathcal{F}$, there exists a non-trivial protocol that securely realizes $\mathcal{F}$ in the $\mathcal{F}_{\text{CRS}}$-hybrid model in the presence of* malicious, adaptive *adversaries. In both cases, the protocol uses a single copy of $\mathcal{F}_{\text{CRS}}$.*

4.9 Multi-party Secure Computation

This section extends the two-party constructions of Sections 4.5–4.8 to the multi-party setting, thereby proving Theorem 4.2.3. The results here relate to a multi-party network where subsets of the parties wish to realize arbitrary (possibly reactive) functionalities of their local inputs. Furthermore, there is an adaptive adversary that can corrupt *any* number of the parties (in particular, no honest majority is assumed). Throughout, we continue to assume a completely asynchronous network without guaranteed message delivery.

This section is organized as follows. We start by showing how to obtain UC multi-party computation in the presence of semi-honest adversaries. Next we define a basic broadcast primitive which will be essential for our protocols in the case of malicious adversaries. We then generalize the UC commitment, zero-knowledge and $\mathcal{F}_{\text{CP}}$ functionalities to the multi-party case. Finally, we construct a multi-party protocol compiler using the generalized $\mathcal{F}_{\text{CP}}$, and obtain UC multi-party computation in the malicious adversarial model. In our presentation below, we assume familiarity with the two-party constructions.

4.9.1 Multi-party Secure Computation for Semi-Honest Adversaries

In this section, we sketch the construction of non-trivial protocols that securely realize any adaptively well-formed functionality $\mathcal{F}$ for semi-honest adversaries. (Recall the definition of adaptively well-formed functionalities in Section 4.3.3.) The construction is a natural extension of the construction for the two-party case. We assume that the set $\mathcal{P}$ of participating parties in any execution is fixed and known; let this set be $P_1, \ldots, P_\ell$. Then, the input lines to the circuit (comprising of the input value, random coins and internal state of the functionality) are shared amongst all ℓ parties. That is, for every input bit a to the circuit, the parties hold random bits $\alpha_1, \ldots, \alpha_\ell$, respectively, under the constraint that $\alpha = \oplus_{i=1}^{\ell} \alpha_i$. Next, the parties compute the circuit inductively from the inputs to outputs so that at every step, they hold shares of the lines already computed. Once the circuit is fully computed, the parties reconstruct the outputs, as required. We now proceed to prove the following proposition:

Proposition 4.9.1 *Assume that trapdoor permutations exist. Then, for any well-formed (multi-party) ideal functionality $\mathcal{F}$, there exists a non-trivial protocol that securely realizes $\mathcal{F}$ in the presence of* semi-honest, static *adversaries. Furthermore, if two-party augmented non-committing encryption protocols exist, then for any adaptively well-formed (multi-party) functionality $\mathcal{F}$, there exists a non-trivial protocol that securely realizes $\mathcal{F}$ in the presence of* semi-honest, adaptive *adversaries.*

As in the two-party case, for adaptive adversaries we assume the existence of *two-party* augmented non-committing encryption protocols. Indeed, as in the two-party case this assumption is needed only to securely realize the two-party functionality $\mathcal{F}_{\text{OT}}^4$, which plays a central role even in the multi-party case.

We begin our proof of Proposition 4.9.1 by presenting a non-trivial multi-party protocol $\Pi_\mathcal{F}$ that securely realizes any adaptively well-formed functionality $\mathcal{F}$ in the $\mathcal{F}_{\text{OT}}$-hybrid model. (We prove the proposition for the adaptive case only, the static case is easily derived.) We start by defining a boolean circuit $C_\mathcal{F}$ that represents an activation of $\mathcal{F}$. The circuit $C_\mathcal{F}$ has $3m$ input lines: m lines represent the input value sent to $\mathcal{F}$ in this activation (i.e., this is the input held by one of the parties). The additional $2m$ input lines are used for $\mathcal{F}$'s random coins and for holding $\mathcal{F}$'s state at the onset of the activation. The circuit also has m output lines for each party and m output lines for final state of $\mathcal{F}$ after the activation (a total of $m\ell + m$ lines). For more details on how $\mathcal{F}$ and $C_\mathcal{F}$ are defined, see the description for the two-party case in Section 4.4.2 (the extensions to the multi-party case are straightforward).

Protocol $\Pi_{\mathcal{F}}$ (for securely realizing $\mathcal{F}$): Let the set of participating parties equal $\mathcal{P} = \{P_1, \ldots, P_\ell\}$. We state the protocol for an activation in which P_1 sends a message to $\mathcal{F}$. When activated with input (sid, v) for P_1 where $|v| \leq m$, the protocol first pads v to length m (according to some standard encoding), and sends a message to all the parties in $\mathcal{P}$, asking them to participate in a joint evaluation of $C_{\mathcal{F}}$. Next, the parties do the following:

1. **Input Preparation Stage:**

   - *Input value:* P_1 starts by sharing its input v with all parties. That is, P_1 chooses ℓ random strings $v_1, \ldots, v_\ell \in_R \{0,1\}^m$ with the constraint that $\oplus_{i=1}^{\ell} v_i = v$. Then, P_1 sends (sid, v_i) to P_i for every $2 \leq i \leq \ell$, and stores v_1.

   - *Internal state:* At the onset of each activation, the parties hold shares of the current internal state of $\mathcal{F}$. That is, let c denote the current internal state of $\mathcal{F}$, where $|c| = m$ and m is an upper bound on the size of the state string stored by $\mathcal{F}$. Then, party P_i holds $c_i \in \{0,1\}^m$ and all the c_i's are random under the restriction that $\oplus_{i=1}^{\ell} c_i = c$. (In the first activation of $\mathcal{F}$, the internal state is empty and so the parties hold fixed shares 0 that denote the empty state.)

   - *Random coins:* Upon the *first* activation of $\mathcal{F}$ only, each party P_i locally chooses a random string $r_i \in_R \{0,1\}^m$. The strings $r_1, \ldots, r_\ell$ then constitute shares of the random coins $r = \oplus_{i=1}^{\ell} r_i$ to be used by $C_{\mathcal{F}}$ in all activations.

   At this point, the parties hold (random) shares of every input line of $C_{\mathcal{F}}$.

2. **Circuit Evaluation:** The parties proceed to evaluate the circuit $C_{\mathcal{F}}$ in a gate-by-gate manner. Let α and β denote the bit-values of the input lines to a given gate. Then every P_i holds bits α_i, β_i such that $\alpha = \sum_{i=1}^{\ell} \alpha_i$ and $\beta = \sum_{i=1}^{\ell} \beta_i$. The gates are computed as follows:

   - *Addition gates:* If the gate is an addition gate, then each P_i locally sets its share of the output line of the gate to be $\gamma_i = \alpha_i + \beta_i$. (Thus $\sum_{i=1}^{\ell} \gamma_i = \sum_{i=1}^{\ell}(\alpha_i + \beta_i) = \alpha + \beta = \gamma$.)

   - *Multiplication gates:* If the gate is a multiplication gate, then the parties need to compute their shares of $\gamma = \left(\sum_{i=1}^{\ell} \alpha_i\right)\left(\sum_{i=1}^{\ell} \beta_i\right)$. The key to carrying out this computation is the following equality:

   $$\left(\sum_{i=1}^{\ell} \alpha_i\right)\left(\sum_{i=1}^{\ell} \beta_i\right) = \ell \cdot \sum_{i=1}^{\ell} \alpha_i \beta_i + \sum_{1 \leq i < j \leq \ell} (\alpha_i + \alpha_j) \cdot (\beta_i + \beta_j)$$

   (See [44, Section 3.2.2] for a justification of this equality.) Notice that each party can compute a share of the first sum locally (by simply computing $\alpha_i \cdot \beta_i$ and multiplying the product by ℓ). Shares of the second sum can be computed using activations of the *two-party* oblivious transfer functionality $\mathcal{F}_{\mathrm{OT}}^4$. (That is, for each pair i and j, parties P_i and P_j compute shares of $(\alpha_i + \alpha_j) \cdot (\beta_i + \beta_j)$. This is exactly the same computation as in the two-party case and can be carried out using $\mathcal{F}_{\mathrm{OT}}^4$.) After computing all of the shares, each party P_i locally sums its shares into a value γ_i, and we have that $\sum_{i=1}^{\ell} \gamma_i = \gamma$, as required.

3. **Output stage:** Following the above stage, the parties hold shares of all the output lines of the circuit $C_{\mathcal{F}}$. Each output line of $C_{\mathcal{F}}$ is either an output addressed to one of the parties $P_1, \ldots, P_\ell$, or belongs to the internal state of $C_{\mathcal{F}}$ after the activation. The activation concludes as follows:

- *P_i's output (for every i):* For every $j \neq i$, party P_j sends P_i all of its shares in P_i's output lines. P_i then reconstructs every bit of its output value by adding the appropriate shares, and writes the result on its output tape.
- *Internal state:* $P_1, \ldots, P_\ell$ all locally store the shares that they hold for the internal state lines of $C_\mathcal{F}$. (These shares are to be used in the next activation.)

Recall that since we are working in an asynchronous network, there is no guarantee on the order of message delivery and messages may be delivered "out of order". In contrast, to maintain correctness the protocol must be executed according to its prescribed order (e.g., new activations must begin only after previous ones have completed and gates may be evaluated only after the shares of the input lines are known). As in the two-party case, this is dealt with by assigning unique identifiers to every message sent during all activations. A full description of how this can be achieved appears in Section 4.4.2. By having the parties store messages that arrive before they are relevant in appropriate buffers (where the time that a message becomes relevant is self-evident from the unique tags), we have that all honest parties process the messages in correct order. Thus, it makes no difference whether or not the adversary delivers the messages according to the prescribed order and we can assume that all messages *are* delivered in order.

This completes the description of $\Pi_\mathcal{F}$. We now sketch the proof that $\Pi_\mathcal{F}$ securely realizes any adaptively well-formed multi-party functionality $\mathcal{F}$:

Claim 4.9.2 *Let $\mathcal{F}$ be an adaptively well-formed multi-party functionality. Then, protocol $\Pi_\mathcal{F}$ securely realizes $\mathcal{F}$ in the $\mathcal{F}_{\rm OT}$-hybrid model, in the presence of semi-honest, adaptive adversaries.*

Proof (sketch): The proof of this claim is very similar to the two-party case (i.e., Claim 4.4.4). First, it is clear that $\Pi_\mathcal{F}$ correctly computes $\mathcal{F}$ (i.e., all parties receive outputs that are distributed according to $\mathcal{F}$). Next, we show the existence of a simulator for $\Pi_\mathcal{F}$. The basis for the simulator's actions is the fact that, as long as there is at least one uncorrupted party, all the intermediary values seen by the parties are uniformly distributed.

Let $\mathcal{A}$ be a semi-honest, adaptive adversary; we construct a simulator $\mathcal{S}$ for the ideal process $\mathcal{F}$. Simulator $\mathcal{S}$ internally invokes $\mathcal{A}$ and works as follows:

Simulating the communication with $\mathcal{Z}$: The input values received by $\mathcal{S}$ from $\mathcal{Z}$ are written on $\mathcal{A}$'s input tape, and the output values of $\mathcal{A}$ are copied to $\mathcal{S}$'s own output tape.

Simulation of the input stage: Recall that in this stage, the only messages sent are random strings $v_2, \ldots, v_\ell$ that P_1 sends to $P_2, \ldots, P_\ell$. Thus, the simulation of this stage involves simulating P_1 sending $\ell - 1$ random strings $v_2, \ldots, v_\ell$ to $P_2, \ldots, P_\ell$. (If P_1 is corrupted, then $v_2, \ldots, v_\ell$ are chosen according to P_1's random tape. Otherwise, $\mathcal{S}$ chooses each v_i uniformly.)

Simulation of the circuit evaluation stage: The addition gates require no simulation since they constitute local computation only. The multiplication gates involve simulation of pair-wise oblivious transfer calls to $\mathcal{F}_{\rm OT}$. We describe the simulation of these oblivious transfers separately for each corruption case.

1. *Oblivious transfers run with an uncorrupted receiver:* In the case that the receiver is not corrupted, the only message seen by $\mathcal{A}$ in a call to $\mathcal{F}_{\rm OT}$ is the session-identifier used. This is therefore easily simulated by $\mathcal{S}$. (If the sender is corrupted, then its input table to $\mathcal{F}_{\rm OT}$ is seen by $\mathcal{A}$. However, this is already defined because it is a function of the sender's view which is known to $\mathcal{A}$.)

2. *Oblivious transfers run with an uncorrupted sender and a corrupted receiver:* In this case, the receiver obtains a uniformly distributed bit γ_2 as output from the oblivious transfer. Therefore, $\mathcal{S}$ merely chooses γ_2 uniformly.
3. *Oblivious transfers run with a corrupted sender and receiver:* Simulation is straightforward when both participating parties are corrupted (all input values and random tapes are already defined).

Simulation of the output stage: $\mathcal{S}$ simulates the parties sending strings in the output stage in order to reconstruct their outputs. First, we note that the shares of the input lines, for any party P_j that is already corrupted, are already defined. (This is because $\mathcal{A}$ holds the view of P_j and this view defines the shares that P_j holds of all the output lines.) This means that the strings that P_j sends in the output stage are also defined. Now, $\mathcal{S}$ defines the strings received by a party P_i in the output stage as follows. If P_i is not corrupted, then $\mathcal{S}$ simulates all the other uncorrupted parties sending P_i uniformly distributed strings. If P_i is corrupted, then $\mathcal{S}$ has P_i's output y_i. $\mathcal{S}$ uses this to choose random strings for the honest parties so that the exclusive-or of these strings along with the defined output strings sent by the corrupted parties equals y_i. (Thus, P_i's output is reconstructed to y_i, as required.) Simulator $\mathcal{S}$ carries out this simulation for all parties $P_1, \ldots, P_\ell$.

Simulation of corruptions before the last honest party is corrupted: When some party P_i is corrupted, $\mathcal{S}$ should provide $\mathcal{A}$ with the internal state of P_i for all the activations of $\mathcal{F}$ (i.e., for all the evaluations of $C_\mathcal{F}$) so far. All the evaluations are dealt with independently from each other (except that P_i's output shares of $\mathcal{F}$'s internal state from one evaluation equals its input shares of $\mathcal{F}$'s internal state in the following evaluation). Also all evaluations, except perhaps for the current one, are complete. Here we describe how $\mathcal{S}$ deals with a single, complete activation. (If the current activation is not complete then $\mathcal{S}$ follows its instructions until the point where P_i is corrupted.)

Upon the corruption of party P_i, simulator $\mathcal{S}$ receives P_i's input x_i and output y_i, and should generate P_i's view of the simulated protocol execution. This view should be consistent with the messages sent in the simulation so far. We begin with the simulation of P_i's view of the input stage. If $i = 1$ (i.e., P_1 is the party that is corrupted), then $\mathcal{S}$ obtains the input value v. Let $v_2, \ldots, v_\ell$ be the random strings that P_1 sent $P_2, \ldots, P_\ell$ in the simulated interaction. Then, $\mathcal{S}$ defines P_1's share of the input to equal v_1 so that $\oplus_{i=1}^{\ell} v_i = v$. $\mathcal{S}$ continues for any P_i (i.e., not just for $i = 1$) as follows. $\mathcal{S}$ chooses random strings $r_i \in_R \{0,1\}^m$ and $c_i \in_R \{0,1\}^s$ and sets P_i's inputs to $C_\mathcal{F}$'s random-coins and internal state to be r_i and c_i, respectively.

Having completed the simulation of P_i's view of the input stage, $\mathcal{S}$ proceeds to simulate P_i's view in the oblivious transfers of the protocol execution. Below we describe the simulation for all the multiplication gates except for those immediately preceding output lines (these will be dealt with separately below). We distinguish four cases:

1. *Oblivious transfers run with P_i as sender and an uncorrupted P_j as receiver:* Recall that in every oblivious transfer, the sender inputs a random-bit γ_1 to mask the outcome. In this case, $\mathcal{S}$ simply chooses γ_1 uniformly. (This is the random bit that P_i supposedly chose upon computing this gate.)
2. *Oblivious transfers run with an uncorrupted P_j as sender and P_i as receiver:* In this case, in the execution of $\Pi_\mathcal{F}$, party P_i receives a uniformly distributed bit γ_2 as output from the oblivious transfer. Therefore, $\mathcal{S}$ chooses γ_2 uniformly.

3. *Oblivious transfers run with P_i as sender and an already corrupted P_j as receiver:* P_j is already corrupted and therefore the value γ_2 that it received from this oblivious transfer has already been fixed in the simulation. Furthermore, both P_i and P_j's circuit inputs v_i, v_j, r_i, r_j and c_i, c_j have been fixed, as too have their views for all the multiplication gates leading to this one. ($\mathcal{S}$ computes the view inductively from the inputs to the outputs.) Thus, the input lines to this oblivious transfer are fixed, as too is P_j's output from the oblivious transfer. This fully defines the oblivious transfer table that P_1 constructs in the protocol execution (as well as its "random" bit γ_1). Therefore, $\mathcal{S}$ constructs the table according to the protocol instructions.
4. *Oblivious transfers run with an already corrupted P_j as sender and P_i as receiver:* As in the previous case, the input lines and the random-bit γ_1 that P_j inputs into the oblivious transfer are fixed. Since P_i's input into this oblivious transfer is also already fixed, this fully defines the bit γ_2 that P_i receives as output.

As we have mentioned, there is a difference regarding the simulation of multiplication gates that precede output lines. (As in the two-party case, we assume for simplicity that every output line is preceded by a multiplication gate.) We describe the simulation of these gates together with the output stage. During the simulation of the output stage, P_i received uniformly distributed strings y_i^j from every party P_j (the strategy for choosing these values is described in the item on "simulation of the output stage"). Note that all the y_i^j's (for $j \neq i$) are defined and fixed.[20] Upon the corruption of P_i, simulator $\mathcal{S}$ receives P_i's output string y_i. The aim of $\mathcal{S}$ is to have P_i's output lines define shares y_i^i such that $\oplus_{j=1}^{\ell} y_i^j = y_i$ (and thus P_i's output reconstruction will be as required). This is done as follows. Recall that the evaluation of each multiplication gate is comprised of a series of oblivious transfers between all pairs of parties. Since P_i is not the last honest party to be corrupted, there exists at least one honest party P_l with which P_i runs a pairwise oblivious transfer in the computation of this gate. All the oblivious transfers of this gate apart from this one are simulated as described above. These simulations all provide bit-shares to P_i: let b denote the sum of these shares. It remains to simulate this last oblivious transfer between P_i and P_l. Let γ_i be the bit of y_i^i that P_i is supposed to receive as its share of the output line that follows from this multiplication gate. Now, the specific oblivious transfer between P_i and P_l defines one share of the output bit γ_i, and all the other shares have already been fixed and sum to b. Thus, the aim of $\mathcal{S}$ is to have P_i's output from the oblivious transfer with P_l equal $\gamma_i + b$ (and thus P_i's overall output from the gate will be γ_i as required). However, P_l is not corrupted. Therefore, whether P_i is the sender or P_l is the sender, P_i's output can be chosen arbitrarily by $\mathcal{S}$. (See the first 2 of the 4 simulation cases above; in those cases, $\mathcal{S}$ merely chooses the output bit randomly.) Thus, $\mathcal{S}$ sets the output bit to be $\gamma_i + b$ and P_i receives the correct bit. This completes the simulation for the corruption of P_i.

Simulation of the corruption of the last honest party: Let P_i be the party that is corrupted last. If $i = 1$, then $\mathcal{S}$ obtains the input-value v into this activation. (As above, in this case, $\mathcal{S}$ defines P_1's share of the input v_1 to be so that $\oplus_{j=1}^{\ell} v_j = v$.) In all cases, $\mathcal{S}$ obtains the output-value y_i that P_i receives. Furthermore, since $\mathcal{F}$ is adaptively well-formed, $\mathcal{S}$ obtains the random tape of $\mathcal{F}$. Given this information, $\mathcal{S}$ computes the internal state of $\mathcal{F}$ in the beginning of this activation. Let c be this state string and let r equal $\mathcal{F}$'s length-m random

[20] Actually, corruption can happen in the middle of the output stage and in such a case only some of the output strings may be fixed. In such a case, first (internally) fix all the output strings and then continue as here.

tape. Now, all other parties are corrupted and thus the shares of the random tape r_j are fixed for every $j \neq i$. $\mathcal{S}$ takes P_i's share of the random tape to be r_i so that $\oplus_{j=1}^{\ell} r_j = r$. Likewise, except for c_i, all the shares of the state input c_j are fixed. $\mathcal{S}$ thus defines c_i so that $\oplus_{j=1}^{\ell} c_j = c$. This completes the simulation of the input stage.

Next $\mathcal{S}$ simulates the circuit evaluation stage, working from the input gates to the output gates, in the same way as described above (i.e., for the case that P_i is not the last honest party corrupted). Notice above that when P_i runs an oblivious transfer with an already corrupted party, then all the inputs and outputs are essentially fixed. Thus, $\mathcal{S}$ merely computes the bit that P_i should see in each oblivious transfer as in the above cases. We therefore have that the simulation of this stage is a deterministic process. This simulation also defines the output shares that P_i receives, thus concluding the simulation.

Output delivery: $\mathcal{S}$ delivers the output from $\mathcal{F}$ to an uncorrupted party P_i when $\mathcal{A}$ delivers all the output shares y_i^j that parties P_j send P_i in the simulation.

It remains to show that no environment $\mathcal{Z}$ can distinguish the case that it interacts with $\mathcal{S}$ and $\mathcal{F}$ in the ideal process or with $\mathcal{A}$ and $\Pi_{\mathcal{F}}$ in the $\mathcal{F}_{\text{OT}}$-hybrid model. The analysis is similar to the one for the two-party case and is omitted. ■

4.9.2 Authenticated Broadcast

In order to obtain our result, we assume that each set of parties that engage in a protocol execution have access to an *authenticated* broadcast channel. The broadcast channel is modeled by the ideal broadcast functionality, $\mathcal{F}_{\text{BC}}$, as defined in Figure 4.12. In our protocols for malicious adversaries, all communication among the parties is carried out via $\mathcal{F}_{\text{BC}}$.

Functionality $\mathcal{F}_{\text{BC}}$

$\mathcal{F}_{\text{BC}}$ proceeds as follows, running with parties $P_1, \ldots, P_n$ and an adversary $\mathcal{S}$:

- Upon receiving a message $(\mathsf{broadcast}, sid, \mathcal{P}, x)$ from P_i, where $\mathcal{P}$ is a set of parties, send $(\mathsf{broadcast}, sid, P_i, \mathcal{P}, x)$ to all parties in $\mathcal{P}$ and to $\mathcal{S}$, and halt.

Figure 4.12: The ideal broadcast functionality

Note that the $\mathcal{F}_{\text{BC}}$-hybrid model does not guarantee delivery of messages, nor does it provide any synchrony guarantees for the messages that are delivered. It only guarantees that no two uncorrupted parties in $\mathcal{P}$ will receive two different message with the same *sid*. We now show that $\mathcal{F}_{\text{BC}}$ can be securely realized, for any number of corrupted parties and without any setup assumptions.

Protocol 2 (universally composable broadcast):

- **Input:** *P_i has input* $(\mathsf{broadcast}, sid, \mathcal{P}, x)$.
- **The Protocol:**
  1. *P_i sends $(sid, \mathcal{P}, x)$ to all parties.*
  2. *Denote by $(\mathcal{P}^j, x^j)$ the message received by P_j in the previous round. Then, every party P_j (for $j \neq i$) sends its message $(sid, \mathcal{P}^j, x^j)$ to all other parties.*

3. *Denote the message received by P_j from P_k in the previous round by $(\mathcal{P}_k^j, x_k^j)$ (recall that x^j denotes the value P_j received from P_i in the first round). Then, P_j outputs* $(\mathsf{broadcast}, sid, P_i, \mathcal{P}, x^j)$ *if it belongs to the set $\mathcal{P}$, if it received* all *the messages $(\mathcal{P}_k^j, x_k^j)$ and this is the only message that it saw (i.e., if it received $(\mathcal{P}_k^j, x_k^j)$ from every P_k and it holds that all these messages are the same). Otherwise, it outputs nothing.*

   We note that if P_j did not receive any value in the first round, then it does not output anything.

Proposition 4.9.3 *Protocol* 2 *securely realizes $\mathcal{F}_{\text{BC}}$ (in the universally composable framework).*

Proof: Let $\mathcal{A}$ be a real-model adversary attacking Protocol 2. We construct an ideal model adversary $\mathcal{S}$ for $\mathcal{A}$ that interacts with $\mathcal{F}_{\text{BC}}$. We differentiate between two cases: in the first the dealer is corrupted (and thus is controlled by $\mathcal{A}$), and in the second it is honest. Let P_i be the dealer in this execution.

- *Case* 1 – *P_i is corrupt:* In the first round, $\mathcal{A}$ (controlling P_i) sends messages to ℓ of the honest parties for some ℓ; denote these messages by $(\mathcal{P}_{i_1}, x_{i_1}), \ldots, (\mathcal{P}_{i_\ell}, x_{i_\ell})$. Simulator $\mathcal{S}$ receives all these messages and then simulates the messages sent by the honest parties in the second round. Furthermore, $\mathcal{S}$ obtains all the messages sent by $\mathcal{A}$ in the second round.

  Now, if there exist j and k ($1 \leq j, k \leq \ell$) such that $(\mathcal{P}_{i_j}, x_{i_j}) \neq (\mathcal{P}_{i_k}, x_{i_k})$, then $\mathcal{S}$ sends nothing to $\mathcal{F}_{\text{BC}}$. Otherwise, let x be the message sent by $\mathcal{A}$. Then, $\mathcal{S}$ sends x to the ideal functionality $\mathcal{F}_{\text{BC}}$. Next, $\mathcal{S}$ defines the set of honest parties to whom to deliver the output $(\mathsf{broadcast}, sid, P_i, \mathcal{P}, x)$ from $\mathcal{F}_{\text{BC}}$. This set of parties is defined to be those parties in $\mathcal{P}$ to whom $\mathcal{A}$ delivers all the second round messages and whose messages all contain the same x. $\mathcal{S}$ concludes by delivering the messages from the $\mathcal{F}_{\text{BC}}$ functionality to these parties, and only to these parties.

- *Case* 2 – *P_i is honest:* $\mathcal{S}$ receives $(\mathsf{broadcast}, sid, P_i, \mathcal{P}, x)$ from $\mathcal{F}_{\text{BC}}$ and simulates P_i's sending x to all the parties controlled by $\mathcal{A}$. Then, $\mathcal{S}$ receives back messages sent by $\mathcal{A}$ to the honest parties. $\mathcal{S}$ defines the set of parties to whom to deliver output as those in $\mathcal{P}$ who receive all the second round messages and who only see x. Then, $\mathcal{S}$ delivers the messages from the $\mathcal{F}_{\text{BC}}$ functionality to these and only to these parties.

We claim that the global output of an ideal execution with $\mathcal{S}$ is *identically distributed* to the global output of a real execution with $\mathcal{A}$. We first deal with the case that P_i is corrupt. If $\mathcal{A}$ sends two different messages in round 1 (i.e., if there exist j and k such that $(\mathcal{P}_{i_j}, x_j) \neq (\mathcal{P}_{i_j}, x_k)$), then by the protocol definition, all honest parties will see both $(\mathcal{P}_{i_j}, x_j)$ and $(\mathcal{P}_{i_k}, x_k)$. (Here it is important that parties do not output $(\mathcal{P}, x)$ unless seeing the round 2 messages of all parties.) Therefore, in a real execution all honest parties will output nothing. This is identical to the case that $\mathcal{S}$ does not send anything to $\mathcal{F}_{\text{BC}}$ in an ideal execution. In contrast, if $\mathcal{A}$ sends the same message x to all honest parties in the first round, then the outputs depend on what $\mathcal{A}$ sends in the second round. Since $\mathcal{S}$ receives all these messages from $\mathcal{A}$, it can see which parties would output $(\mathcal{P}, x)$ and which parties would output nothing. $\mathcal{S}$ thus delivers the $(\mathsf{broadcast}, \ldots)$ messages from $\mathcal{F}_{\text{BC}}$ only to the parties which would output x in the real model. We conclude that the output is identical.

In the case that P_i is honest, $\mathcal{A}$ can cause honest parties to output nothing (rather than x) by sending them messages $(\mathcal{P}', x') \neq (\mathcal{P}, x)$ in the second round or by not delivering messages. As above, $\mathcal{S}$ receives all these messages and therefore its delivery of $(\mathsf{broadcast}, \ldots)$ messages from $\mathcal{F}_{\text{BC}}$ accurately represents exactly what happens in a real execution. ■

4.9.3 One-to-Many Commitment, Zero-Knowledge and Commit-and-Prove

One-to-Many UC commitment. We begin by defining a *one-to-many* extension of the UC commitment functionality, denoted $\mathcal{F}^{1:\mathrm{M}}_{\mathrm{MCOM}}$. In this functionality, *one* party commits to a value to *many* receivers. The formal definition appears in Figure 4.13. Similarly to the two-party case, the commitment functionality is presented as a *multi-session functionality.* From here on, the JUC theorem of [21] is applied and we consider only single-session functionalities (see Section 4.3.2 for more explanation). We denote the single session analog to $\mathcal{F}^{1:\mathrm{M}}_{\mathrm{MCOM}}$ by $\mathcal{F}^{1:\mathrm{M}}_{\mathrm{COM}}$.

Functionality $\mathcal{F}^{1:\mathrm{M}}_{\mathrm{MCOM}}$

$\mathcal{F}^{1:\mathrm{M}}_{\mathrm{MCOM}}$ proceeds as follows, running with parties $P_1, \ldots, P_n$ and an adversary $\mathcal{S}$:

- **Commit Phase:** Upon receiving a message $(\mathsf{commit}, sid, ssid, \mathcal{P}, b)$ from P_i where $\mathcal{P}$ is a set of parties and $b \in \{0,1\}$, record the tuple $(ssid, P_i, \mathcal{P}, b)$ and send the message $(\mathsf{receipt}, sid, ssid, P_i, \mathcal{P})$ to all the parties in $\mathcal{P}$ and to $\mathcal{S}$. Ignore any future commit messages with the same *ssid*.
- **Prove Phase:** Upon receiving a message $(\mathsf{reveal}, sid, ssid)$ from P_i: If a tuple $(ssid, P_i, \mathcal{P}, b)$ was previously recorded, then send the message $(\mathsf{reveal}, sid, ssid, b)$ to all parties in $\mathcal{P}$ and to $\mathcal{S}$. Otherwise, ignore.

Figure 4.13: One-to-Many multi-session commitment

The key observation in realizing the $\mathcal{F}^{1:\mathrm{M}}_{\mathrm{MCOM}}$ functionality is that Protocol UAHC (of Section 4.5) that securely realizes the two-party commitment functionality $\mathcal{F}_{\mathrm{MCOM}}$ is *non-interactive.* Therefore, the one-to-many extension is obtained by simply having the committer broadcast the commitment string of Protocol UAHC to all the participating parties on the broadcast channel. The proof that this protocol realizes $\mathcal{F}^{1:\mathrm{M}}_{\mathrm{MCOM}}$ is almost identical to the proof that Protocol UAHC realizes $\mathcal{F}_{\mathrm{MCOM}}$, and is omitted. We do, however, mention one important point. The commitment string is broadcast using the $\mathcal{F}_{\mathrm{BC}}$ functionality which ensures that only one message is broadcast using a given session identifier. This is important because otherwise the adversary could broadcast two different commitment strings c_1 and c_2, where it delivers c_1 to some of the honest parties and c_2 to the others. This is, of course, not allowed by the $\mathcal{F}^{1:\mathrm{M}}_{\mathrm{MCOM}}$ functionality that ensures that all parties receive the same commitment for the same identifier pair $(sid, ssid)$. We therefore have the following:

Proposition 4.9.4 *Assuming the existence of trapdoor permutations, there exists a (non-interactive) protocol that securely realizes $\mathcal{F}^{1:\mathrm{M}}_{\mathrm{MCOM}}$ in the $(\mathcal{F}_{\mathrm{CRS}}, \mathcal{F}_{\mathrm{BC}})$-hybrid model*[21]*, in the presence of malicious, adaptive adversaries. Furthermore, this protocol uses only a single copy of $\mathcal{F}_{\mathrm{CRS}}$.*

One-to-Many UC zero-knowledge. Similarly to the one-to-many extension of commitments, we define a one-to-many functionality where one party proves a statement to some set of parties. The definition of the (single-session) one-to-many zero-knowledge functionality, denoted $\mathcal{F}^{1:\mathrm{M}}_{\mathrm{ZK}}$, appears in Figure 4.14. (For simplicity, in the multi-party case we concentrate on single-session zero-knowledge, constructed using a single-session version of $\mathcal{F}^{1:\mathrm{M}}_{\mathrm{MCOM}}$. These protocols will later be composed, using universal composition with joint state, to obtain protocols that use only a single copy of the reference string when realizing all the copies of of $\mathcal{F}^{1:\mathrm{M}}_{\mathrm{ZK}}$.)

[21] In the $(\mathcal{F}_{\mathrm{CRS}}, \mathcal{F}_{\mathrm{BC}})$-hybrid model, all parties have ideal access to both the common reference string functionality $\mathcal{F}_{\mathrm{CRS}}$ and the ideal broadcast functionality $\mathcal{F}_{\mathrm{BC}}$.

Functionality $\mathcal{F}_{\rm ZK}^{\rm 1:M}$

$\mathcal{F}_{\rm ZK}^{\rm 1:M}$ proceeds as follows, running with parties $P_1, \ldots, P_n$ and an adversary $\mathcal{S}$, and parameterized with a relation R:

- Upon receiving a message $(\mathsf{ZK\text{-}prover}, sid, \mathcal{P}, x, w)$ from a party P_i where $\mathcal{P}$ is a set of parties: If $R(x,w) = 1$, then send $(\mathsf{ZK\text{-}proof}, sid, P_i, \mathcal{P}, x)$ to all parties in $\mathcal{P}$ and to $\mathcal{S}$ and halt. Otherwise, halt.

Figure 4.14: Single-session, One-to-Many zero-knowledge

As with the case of commitments, a non-interactive protocol that realizes the two-party zero-knowledge functionality $\mathcal{F}_{\rm ZK}$ could be directly used to realize $\mathcal{F}_{\rm ZK}^{\rm 1:M}$. For the case of static adversaries, the protocol of [26] can be used. However, for the case of adaptive adversaries, no non-interactive protocol is known. Rather, we base the one-to-many extension on the interactive UC zero-knowledge protocol of [16]. Their protocol is basically that of parallel Hamiltonicity (cf. [11]), except that the commitments used are universally composable. Our extension of this protocol to the one-to-many case follows the methodology of [44] and is presented in the proof of the following proposition:

Proposition 4.9.5 *There exists a protocol that realizes $\mathcal{F}_{\rm ZK}^{\rm 1:M}$ in the $(\mathcal{F}_{\rm COM}^{\rm 1:M}, \mathcal{F}_{\rm BC})$-hybrid model, in the presence of malicious, adaptive adversaries.*

Proof (sketch): The protocol for realizing $\mathcal{F}_{\rm ZK}^{\rm 1:M}$ works by having the prover separately prove the statement in question to all parties. The protocol used in each of these pairwise proofs is exactly the two-party protocol of [16], with the exception that the messages of each proof are broadcast to all parties. (This also means that all commitments and decommitments are run using $\mathcal{F}_{\rm COM}^{\rm 1:M}$, rather than the two-party $\mathcal{F}_{\rm COM}$. ($\mathcal{F}_{\rm COM}^{\rm 1:M}$ is the single-session parallel to $\mathcal{F}_{\rm MCOM}^{\rm 1:M}$.) Also, the protocol must make sure that each invocation of broadcast will have a unique session ID. This can be done in standard ways, given the unique session ID of the zero-knowledge protocol.) Then, a party accepts the proof, outputting (ZK-proof, $sid, P_i, \mathcal{P}, x$), if and only if all the pairwise proofs are accepting. Note that, other than the use of $\mathcal{F}_{\rm COM}^{\rm 1:M}$, no cryptographic primitives are used. Indeed, security of this protocol in the $\mathcal{F}_{\rm COM}^{\rm 1:M}$-hybrid model is unconditional.

Next, note that it is indeed possible for the parties to know whether all the pairwise proofs are accepting. This is because all the commitments and messages are seen by all the parties and the zero-knowledge proof of Hamiltonicity used by [16] is publicly verifiable (i.e., it is enough to see the transcript of prover/verifier messages to know whether or not the proof was accepted by the verifier).

Now, recall that in order to prove the universal composability of $\mathcal{F}_{\rm ZK}^{\rm 1:M}$, we must present an ideal-process adversary (i.e., a simulator) that simulates proofs for the case that the prover is not corrupted and verifiers are corrupted, and is also able to extract the witness from an adversarially generated proof (for the case that the prover is corrupted). When simulating a proof for a corrupted verifier, the simulator for $\mathcal{F}_{\rm ZK}^{\rm 1:M}$ works simply by running the simulator of the two-party protocol of [16] for every pairwise proof. On the other hand, in order to extract the witness from a corrupted prover, first note that it is possible to run the two-party extractor for any pairwise proof in which the verifier is not corrupted. Now, the scenario in which we need to run the extractor here is where the prover is corrupted and at least one verifier is not (otherwise, all parties are corrupted and simulation is straightforward). Therefore, there exists one pairwise proof in which the verifier is not corrupted. The simulator for $\mathcal{F}_{\rm ZK}^{\rm 1:M}$ thus runs the simulator for the protocol of [16] for this proof.

Finally, we note that the simulator delivers the output of $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$ to the verifiers if and only if all verifiers accept in the simulation. (Thus, the parties' outputs in the ideal process are the same as in a real execution.) This concludes the proof sketch. ■

One-to-Many UC commit-and-prove. The one-to-many extension of the commit-and-prove functionality, denoted $\mathcal{F}_{\text{CP}}^{1:\text{M}}$, is presented in Figure 4.15. The functionality handles a single session only, and requires that all commitments and proofs are to the *same* set $\mathcal{P}$. (This set is fixed the first time a commit is sent with a given *sid*.)

Functionality $\mathcal{F}_{\text{CP}}^{1:\text{M}}$

$\mathcal{F}_{\text{CP}}^{1:\text{M}}$ proceeds as follows, running with parties $P_1, \ldots, P_n$ and an adversary $\mathcal{S}$, and parameterized by a value k and a relation R:

- **Commit Phase:** Upon receiving a message $(\mathsf{commit}, sid, \mathcal{P}, w)$ from P_i where $\mathcal{P}$ is a set of parties and $w \in \{0,1\}^k$, append the value w to the list $\overline{w}$, record $\mathcal{P}$, and send the message $(\mathsf{receipt}, sid, P_i, \mathcal{P})$ to the parties in $\mathcal{P}$ and $\mathcal{S}$. (Initially, the list $\overline{w}$ is empty. Also, if a commit message has already been received, then check that the recorded set of parties is $\mathcal{P}$. If it is a different set, then ignore this message.)
- **Prove Phase:** Upon receiving a message $(\mathsf{CP\text{-}prover}, sid, x)$ from P_i, where $x \in \{0,1\}^{\text{poly}(k)}$, compute $R(x, \overline{w})$: If $R(x, \overline{w}) = 1$, then send the message $(\mathsf{CP\text{-}proof}, sid, x)$ to the parties in $\mathcal{P}$ and to $\mathcal{S}$. Otherwise, ignore.

Figure 4.15: One-to-Many commit-and-prove

Our protocol for securely realizing the one-to-many commit-and-prove functionality $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ is constructed in the $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$-hybrid model. The protocol, denoted $\text{ACP}^{1:\text{M}}$, is very similar to Protocol ACP for the two-party case. Recall that Protocol ACP begins with the receiver choosing a pair (s,t), where $s = f(t)$ and f is a one-way function. The value s is then used by the committer who commits to w by sending $c = \mathsf{aHC}_s(w;r)$. This is generalized in the natural way by having *every* receiving party P_j choose a pair (s_j, t_j), and the committer then sending $c_j = \mathsf{aHC}_{s_j}(w; r_j)$ for all values of s_j. In addition, the committer proves that all these commitments are to the same w (this is done to prevent the committer from committing to different w's for different s_j's). We define a *compound* commitment scheme as follows. Let $\vec{s} = (s_1, \ldots, s_\ell)$ and $\vec{r} = (r_1, \ldots, r_\ell)$. Then, define $\mathsf{aHC}_{\vec{s}}(w; \vec{r}) = (\mathsf{aHC}_{s_1}(w; r_1), \ldots, \mathsf{aHC}_{s_\ell}(w; r_\ell))$. Restating the above, the commit phase consists of the committer committing to w using the compound scheme $\mathsf{aHC}_{\vec{s}}$ and proving that the commitment was generated correctly.

The multi-party protocol $\text{ACP}^{1:\text{M}}$ uses three different copies of $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$, where each copy is parameterized by a different relation. The copies are denoted $\mathcal{F}_{\text{ZK,T}}^{1:\text{M}}$ (for the initialization phase), $\mathcal{F}_{\text{ZK,C}}^{1:\text{M}}$ (for the commit stage) and $\mathcal{F}_{\text{ZK,P}}^{1:\text{M}}$ (for the prove stage). These functionalities are differentiated by session identifiers sid_T, sid_C and sid_P, respectively. These identifiers should be unique, as long as the session ID of the current instance of $\text{ACP}^{1:\text{M}}$ is unique. One way to guarantee this is setting $sid_T = sid \circ T$, $sid_C = sid \circ C$ and $sid_P = sid \circ P$, where sid is the session ID of the current instance of $\text{ACP}^{1:\text{M}}$. The protocol is presented in Figure 4.16.

Proposition 4.9.6 *Assuming the existence of one-way functions, Protocol* $\text{ACP}^{1:\text{M}}$ *of Figure* 4.16 *securely realizes* $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ *in the* $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$*-hybrid model, in the presence of adaptive adversaries.*

Protocol $\mathrm{ACP}^{1:\mathrm{M}}$

- **Auxiliary Input:** A security parameter k, and a session identifier sid.
- **Initialization phase:**
  The first time that the committer P_i wishes to commit to a value to the set of parties $\mathcal{P}$ using the identifier sid, the parties in $\mathcal{P}$ execute the following initialization phase. (To simplify notation, assume that $\mathcal{P} = \{P_1, \ldots, P_\ell\}$ for some ℓ, and that the sender belongs to $\mathcal{P}$. Also, the parties ignore incoming messages that are addressed to a set $\mathcal{P}'$ that is different than the set $\mathcal{P}$ specified in the first message.)
  1. P_i sends a $(\mathsf{broadcast}, sid, \mathcal{P}, \text{begin-commit})$ message to $\mathcal{F}_{\text{BC}}$ to indicate that it wishes to initiate a commit activation.
  2. Upon receiving $(\mathsf{broadcast}, sid, P_i, \mathcal{P}, \text{begin-commit})$, each party $P_j \in \mathcal{P}$ records the triple $(sid, P_i, \mathcal{P})$. From here on, the parties only relate to messages with identifier sid if they are associated with the committer/prover P_i and set of parties $\mathcal{P}$.
     Then, every P_j chooses $t_j \in_R \{0,1\}^k$, computes $s_j = f(t_j)$ (where f is a one-way function), and sends $(\mathsf{ZK\text{-}prover}, sid_T, \mathcal{P}, s_j, t_j)$ to $\mathcal{F}_{\text{ZK,T}}^{1:\mathrm{M}}$, where $\mathcal{F}_{\text{ZK,T}}^{1:\mathrm{M}}$ is parameterized by the relation R_T defined by:
     $$R_T \stackrel{\text{def}}{=} \{(s,t) \mid s = f(t)\}$$
  3. Upon receiving $(\mathsf{ZK\text{-}proof}, sid_T, P_j, \mathcal{P}, s_j)$ from $\mathcal{F}_{\text{ZK,T}}^{1:\mathrm{M}}$, all the parties in $\mathcal{P}$ (including the committer P_i) record the value s_j. This phase concludes when all parties in $\mathcal{P}$ have sent the appropriate $\mathsf{ZK\text{-}proof}$ message, and thus when all the parties hold the vector $\vec{s} = (s_1, \ldots, s_\ell)$.

  The parties now proceed to the commit phase.
- **Commit phase:** (P_i's input is $(\mathsf{commit}, sid, \mathcal{P}, w)$, where $w \in \{0,1\}^k$.)
  1. P_i computes the compound commitment $c = \mathsf{aHC}_{\vec{s}}(w; \vec{r})$ where the vector $\vec{s}$ is the one obtained in the initialization phase, and the r_j's in $\vec{r}$ are uniformly chosen.
     P_i then sends $(\mathsf{ZK\text{-}prover}, sid_C, \mathcal{P}, (\vec{s}, c), (w, \vec{r}))$ to $\mathcal{F}_{\text{ZK,C}}^{1:\mathrm{M}}$, where $\mathcal{F}_{\text{ZK,C}}^{1:\mathrm{M}}$ is parameterized by the relation R_C defined by:
     $$R_C \stackrel{\text{def}}{=} \{(\vec{s}, c), (w, \vec{r}) \mid c = \mathsf{aHC}_{\vec{s}}(w; \vec{r})\}$$
     (That is, R_C verifies that c is a valid compound commitment to the value w, using $\vec{s}$.)
     In addition, P_i stores in a list $\overline{w}$ all the values w that were sent, and in lists $\overline{c}$ and $\overline{r}$ the corresponding commitment values c and random strings $\vec{r} = (r_1, \ldots, r_\ell)$.
  2. Upon receiving $(\mathsf{ZK\text{-}proof}, sid_C, P_i, \mathcal{P}, (\vec{s}', c))$ from $\mathcal{F}_{\text{ZK,C}}^{1:\mathrm{M}}$, every party $P_j \in \mathcal{P}$ verifies that $\vec{s}' = \vec{s}$ (where $\vec{s}$ equals the list of strings that it recorded in the initialization phase). If yes, then P_j outputs $(\mathsf{receipt}, sid)$ and adds the commitment c to its list $\overline{c}$. (Initially, $\overline{c}$ is empty.) Otherwise, the parties in $\mathcal{P}$ ignore the message.
- **Prove phase:** (P_i's input is $(\mathsf{CP\text{-}prover}, sid, x)$.)
  1. P_i sends $(\mathsf{ZK\text{-}prover}, sid_P, P_i, \mathcal{P}, (x, \vec{s}, \overline{c}), (\overline{w}, \overline{r}))$ to $\mathcal{F}_{\text{ZK,P}}^{1:\mathrm{M}}$, where $\overline{c}$, $\overline{w}$ and $\overline{r}$ are the lists described above. Let $\overline{w} = (w_1, \ldots, w_m)$, $\overline{c} = (c_1, \ldots, c_m)$ and $\overline{r} = (\vec{r}_1, \ldots, \vec{r}_m)$. Then, $\mathcal{F}_{\text{ZK,P}}^{1:\mathrm{M}}$ is parameterized by the relation R_P defined by:
     $$R_P \stackrel{\text{def}}{=} \{((x, \vec{s}, \overline{c}), (\overline{w}, \overline{r})) \mid R(x, \overline{w}) = 1 \ \&\ \forall j \ c_j = \mathsf{aHC}_{\vec{s}}(w_j; \vec{r}_j)\}$$
     That is, R_P verifies that $R(x, \overline{w}) = 1$ and that $\overline{c}$ contains commitments to the previously committed values $\overline{w}$.
  2. Upon receiving $(\mathsf{ZK\text{-}proof}, sid_P, P_i, \mathcal{P}, (x, \vec{s}', \overline{c}))$ from $\mathcal{F}_{\text{ZK,P}}^{1:\mathrm{M}}$, every party in $\mathcal{P}$ verifies that $\vec{s}' = \vec{s}$ and that its list of stored commitments equals $\overline{c}$. If yes, then it outputs $(\mathsf{CP\text{-}proof}, sid, x)$. Otherwise, it ignores the message.

Figure 4.16: A protocol for realizing $\mathcal{F}_{\text{CP}}^{1:\mathrm{M}}$ for adaptive adversaries

Proof (sketch): The proof of this proposition is very similar to the proof of Proposition 4.7.2 for the two-party case. Let $\mathcal{A}$ be an adaptive adversary who operates against Protocol ACP$^{1:\mathrm{M}}$ in the $\mathcal{F}_{\mathrm{ZK}}^{1:\mathrm{M}}$-hybrid model. We construct a simulator $\mathcal{S}$ such that no environment $\mathcal{Z}$ can tell with non-negligible probability whether it is interacting with $\mathcal{A}$ and parties running Protocol ACP$^{1:\mathrm{M}}$ in the $\mathcal{F}_{\mathrm{ZK}}^{1:\mathrm{M}}$-hybrid model or with $\mathcal{S}$ in the ideal process for $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$. Simulator $\mathcal{S}$ operates by running a simulated copy of $\mathcal{A}$ and using $\mathcal{A}$ in order to interact with $\mathcal{Z}$ and $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$. $\mathcal{S}$ works as follows.

Simulating the initialization phase: $\mathcal{S}$ records the pairs $(s_1,t_1),\ldots,(s_\ell,t_\ell)$ from the initialization phase of an execution, and defines $\vec{s}=(s_1,\ldots,s_\ell)$. For every uncorrupted receiving party P_j, simulator $\mathcal{S}$ chooses the pair (s_j,t_j) by itself. For corrupted receiving parties, the pairs are chosen by the simulated $\mathcal{A}$ and $\mathcal{S}$ obtains the t_j's from $\mathcal{A}$'s messages to $\mathcal{F}_{\mathrm{ZK,T}}^{1:\mathrm{M}}$.

Simulating the case where the committer is corrupted: We first describe how to simulate the commit phase. Whenever $\mathcal{A}$ (controlling P_i) wishes to commit to a value, $\mathcal{S}$ obtains the message $(\mathsf{ZK\text{-}prover}, sid_C, \mathcal{P}, (\vec{s},c),(w,\vec{r}))$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathrm{ZK,C}}^{1:\mathrm{M}}$. $\mathcal{S}$ checks that $\vec{s}$ is as generated in the initialization phase and that $c=\mathsf{aHC}_{\vec{s}}(w;\vec{r})$. If yes, then $\mathcal{S}$ internally passes $\mathcal{A}$ the message $(\mathsf{ZK\text{-}proof}, sid_C, \mathcal{P}, (\vec{s},c))$ and externally sends $(\mathsf{commit}, sid, \mathcal{P}, w)$ to $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$. Furthermore, $\mathcal{S}$ adds the commitment c to its list of commitments $\overline{c}$.

We now describe the simulation of the prove phase. Whenever $\mathcal{A}$ wishes to prove a statement, $\mathcal{S}$ receives the message $(\mathsf{ZK\text{-}prover}, sid_P, (x,\vec{s},\overline{c}),(\overline{w},\overline{r}))$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathrm{ZK,P}}^{1:\mathrm{M}}$. $\mathcal{S}$ then checks that the list $\overline{c}$ is as stored above and that $R(x,\overline{w})=1$. If yes, then $\mathcal{S}$ internally passes $(\mathsf{ZK\text{-}proof}, sid_P, (x,\vec{s},\overline{c}))$ to $\mathcal{A}$ and externally sends $(\mathsf{CP\text{-}prover}, sid, x)$ to $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$. Otherwise, it ignores the message.

Simulating the case where the committer is not corrupted: Whenever an uncorrupted party P_i commits to an unknown value w, simulator $\mathcal{S}$ hands $\mathcal{A}$ a commitment to 0^k as the commitment value. More precisely, whenever $\mathcal{S}$ receives from $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$ a message $(\mathsf{receipt}, sid, P_i, \mathcal{P})$ where P_i is uncorrupted, simulator $\mathcal{S}$ computes $c=\mathsf{aHC}_{\vec{s}}(0^k;\vec{r})$ and hands $\mathcal{A}$ the message $(\mathsf{ZK\text{-}proof}, sid_C, P_i, \mathcal{P}, (\vec{s},c))$, as if coming from $\mathcal{F}_{\mathrm{ZK}}^{\mathrm{C}}$. (Recall that by the aHC scheme, given the trapdoor information $\vec{t}=(t_1,\ldots,t_\ell)$, a commitment to 0 with $\vec{s}$ can be opened as either 0 or 1; see Section 4.5.)

The simulation of the prove phase is carried out as follows. Whenever $\mathcal{S}$ receives a message $(\mathsf{CP\text{-}proof}, sid, x)$ from $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$, it internally passes $\mathcal{A}$ the message $(\text{ZK-proof}, sid_P, (x,\vec{s},\overline{c}))$, where $\overline{c}$ is the list of simulated commitments generated above.

Dealing with the corruption of parties: The only private information held by a receiving party P_j is the trapdoor information t_j that it chooses in the initialization phase. As we have seen in the simulation of the initialization phase above, $\mathcal{S}$ knows all of the trapdoors in the simulated execution. Therefore, when $\mathcal{A}$ corrupts a receiving party P_j, simulator $\mathcal{S}$ internally passes t_j to $\mathcal{A}$.

The committing party P_i's private state in an execution of Protocol ACP$^{1:\mathrm{M}}$ consists of the list of committed values $\overline{w}$ and the list of vectors of random strings $\overline{r}$ (that contain the decommitment information of the list $\overline{c}$). Therefore, when $\mathcal{A}$ corrupts the committer P_i, simulator $\mathcal{S}$ first externally corrupts P_i in the ideal process and obtains the list $\overline{w}$. Next, $\mathcal{S}$ generates the list $\overline{r}$ so that the simulated list of commitments $\overline{c}$ is "explained" as a list of commitments to $\overline{w}$. $\mathcal{S}$ can do this because it has all of the trapdoor information $t_1,\ldots,t_\ell$ (this case is identical to in the proof of Proposition 4.7.2).

The analysis of the correctness of the simulation is analogous to in the two-party case and is omitted. ■

4.9.4 Multi-party Secure Computation for Malicious Adversaries

As in the two-party case, multi-party secure computation in the presence of malicious adversaries is obtained by constructing a protocol compiler that transforms protocols for the semi-honest model into protocols for the malicious model. This compiler is then applied to Protocol $\Pi_{\mathcal{F}}$ of Section 4.9.1. The compiler is constructed in the $\mathcal{F}_{\text{CP}}^{1:\text{M}}$-hybrid model and in a very similar way to the two-party compiler. The compiler itself is described in Figure 4.17. We note that each party has to commit and prove statements to all other parties during the protocol execution. In order to do this, each party P_i uses a separate invocation of $\mathcal{F}_{\text{CP}}^{1:\text{M}}$, with session ID sid_i. (Also here, the protocol should make sure that these session ID's are unique as long as the session ID of the current copy of $\mathsf{Comp}(\Pi)$ is unique. This can be done by setting $sid_i = sid \circ i$ where sid is the session ID of the current copy of $\mathsf{Comp}(\Pi)$. The relations parameterizing these functionalities are natural extensions of the relations parameterizing the relations $\mathcal{F}_{\text{CP}}^{1}$ and $\mathcal{F}_{\text{CP}}^{2}$ in the two-party compiler of Figure 4.11. The multi-party compiler here also uses "standard" commitments (rather than UC commitments) for coin-tossing. In order to remain in the $\mathcal{F}_{\text{CP}}^{1:\text{M}}$-hybrid model, these commitments are implemented by $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ where the relation used is the identity relation (and so a proof is just a decommitment). Once again, the different invocations of $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ are distinguished by unique identifiers (in the coin-tossing used for generating P_j's random tape, P_i uses $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ with session ID $sid_{i,j} = sid \circ i \circ j$). Notice that protocol $\mathsf{Comp}(\Pi)$ essentially broadcasts (via $\mathcal{F}_{\text{CP}}^{1:\text{M}}$) each message that was sent in Π, even if this message was originally sent only to a single party. This is done to provide all parties with consistent views of the execution; it clearly has no negative effect on the security of the protocol (since the adversary anyway sees all messages).

Implicit in the above protocol specification is the fact that all parties only consider messages that are associated with the specified session identifiers and referring to the same set of parties $\mathcal{P}$. All other messages are ignored. As in the two-party case, we assume that Π is such that the parties copy their input tape onto an internal work tape when first activated.

We now prove that the compiler achieves the desired result:

Proposition 4.9.7 (multi-party protocol compiler): *Let Π be a multi-party protocol and let $\mathsf{Comp}(\Pi)$ be the protocol obtained by applying the compiler of Figure 4.17 to Π. Then, for every* malicious *adversary $\mathcal{A}$ that interacts with $\mathsf{Comp}(\Pi)$ in the $\mathcal{F}_{\text{CP}}^{1:\text{M}}$-hybrid model there exists a* semi-honest *adversary $\mathcal{A}'$ that interacts with Π in the plain real-life model, such that for every environment $\mathcal{Z}$,*

$$\text{REAL}_{\Pi,\mathcal{A}',\mathcal{Z}} \equiv \text{HYBRID}_{\mathsf{Comp}(\Pi),\mathcal{A},\mathcal{Z}}^{\mathcal{F}_{\text{CP}}^{1:\text{M}}}$$

Proof (sketch): The proof sketch is very similar to the proof of Proposition 4.8.1 for the two-party case. We construct a semi-honest adversary $\mathcal{A}'$ from the malicious adversary $\mathcal{A}$. Adversary $\mathcal{A}'$ runs the protocol Π while internally simulating an execution of $\mathsf{Comp}(\Pi)$ for $\mathcal{A}$. The key point in the simulation is that $\mathcal{A}'$ is able to complete the simulation in spite of the fact that, being semi-honest, it cannot diverge from the protocol specification. This is so since $\mathcal{A}$ is forced to send all messages via $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ that verifies their correctness. Thus, essentially, $\mathcal{A}$ must behave in a semi-honest way and can be simulated by a truly semi-honest party $\mathcal{A}'$. (Of course, $\mathcal{A}$ is not semi-honest and can send arbitrary messages. However, since all invalid messages are ignored by $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ in $\mathsf{Comp}(\Pi)$, they do not cause any problem.) $\mathcal{A}'$ runs a simulated copy of $\mathcal{A}$, and proceeds as follows:

$\mathsf{Comp}(\Pi)$

Party P_i proceeds as follows (the code for all other parties is analogous):

1. *Random tape generation:* When activating $\mathsf{Comp}(\Pi)$ for the first time with session identifier sid and set $\mathcal{P}$ or parties, party P_i proceeds as follows. For every party P_j, the parties run the following procedure in order to choose a random tape for P_j:

   (a) P_i chooses $r_i^j \in_R \{0,1\}^k$ and sends $(\mathsf{commit}, sid_{i,j}, \mathcal{P}, r_i^j)$ to $\mathcal{F}_{\text{CP}}^{1:\text{M}}$.

   (b) P_i receives $(\mathsf{receipt}, sid_{k,j}, P_k, \mathcal{P})$ for every $P_k \in \mathcal{P}$. P_i also receives $(\mathsf{receipt}, sid_j, P_j, \mathcal{P})$, where P_j is the party for whom the random tape is being chosen. P_i then uses $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ to decommit to its value r_i^j. That is, P_i sends $(\mathsf{CP\text{-}prover}, sid_{i,j}, r_i^j)$ to $\mathcal{F}_{\text{CP}}^{1:\text{M}}$, where the relation parameterizing the $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ functionality with identifier $sid_{i,j}$ is the identity relation (i.e., $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ sends $(\text{CP-proof}, sid_{i,j}, r_i^j)$ if r_i^j was the value previously committed to; it thus serves as a regular commitment functionality).

   (c) P_i receives $(\mathsf{CP\text{-}proof}, sid_{k,j}, r_k^j)$ messages for every $k \neq j$ and defines the string $s_j = \bigoplus_{k \neq j} r_k^j$. (The random tape for P_j is defined by $r_j = r_j^j \oplus s_j$.)

   When choosing a random tape for P_i, the only difference for P_i is that it sends its random string r_i^i to $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ indexed by session-identifier sid_i and it does not decommit (as is understood from P_j's behavior above).

2. *Activation due to new input:* When activated with input (sid, x), party P_i proceeds as follows.

   (a) *Input commitment:* P_i sends $(\mathsf{commit}, sid_i, \mathcal{P}, x)$ to $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ and adds x to the list of inputs $\overline{x}_i$ (this list is initially empty and contains P_i's inputs from all the previous activations of Π). (At this point all other parties P_j receive the message $(\mathsf{receipt}, sid_i, P_i, \mathcal{P})$ from $\mathcal{F}_{\text{CP}}^{1:\text{M}}$. P_i then proceeds to the next step.)

   (b) *Protocol computation:* Let $\overline{m}$ be the series of Π-messages that were broadcast in all the activations of Π until now ($\overline{m}$ is initially empty). P_i runs the code of Π on its input list $\overline{x}_i$, messages $\overline{m}$, and random tape r_i (as generated above). If Π instructs P_i to broadcast a message, P_i proceeds to the next step (Step 2c).

   (c) *Outgoing message transmission:* For each outgoing message m that P_i sends in Π, P_i sends $(\mathsf{CP\text{-}prover}, sid_i, (m, s_i, \overline{m}))$ to $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ with a relation R_Π defined as follows:

   $$R_\Pi = \left\{((m, s_i, \overline{m}), (\overline{x}_i, r_i^i)) \mid m = \Pi(\overline{x}_i, r_i^i \oplus s_i, \overline{m})\right\}$$

   In other words, P_i proves that m is the correct next message generated by Π when the input sequence is $\overline{x}_i$, the random tape is $r_i = r_i^i \oplus s_i$ and the series of broadcast Π-messages equals $\overline{m}$. (Recall that r_i^i and all the elements of $\overline{x}_i$ were committed to by P_i in the past using commit activations of $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ with identifier sid_i, and that s_i is the random-string derived in the random tape generation for P_i above.)

3. *Activation due to incoming message:* Upon receiving a message $(\mathsf{CP\text{-}proof}, sid_j, (m, s_j, \overline{m}))$ that is sent by P_j, party P_i first verifies that the following conditions hold (note that $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ with sid_j is parameterized by the same relation R_Π as $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ with sid_i above):

   - s_j is the random string that is derived in the random tape generation for P_j above.
   - $\overline{m}$ equals the series of Π-messages that were broadcast in all the activations until now. (P_i knows these messages because all parties see all messages sent.)

   If any of these conditions fail, then P_i ignores the messages. Otherwise, P_i appends m to $\overline{m}$ and proceeds as in Steps 2b and send-step-multi above.

4. *Output:* Whenever Π generates an output value, $\mathsf{Comp}(\Pi)$ generates the same output value.

Figure 4.17: The compiled protocol $\mathsf{Comp}(\Pi)$

Simulating the communication with $\mathcal{Z}$: The input values received by $\mathcal{A}'$ from $\mathcal{Z}$ are written on $\mathcal{A}$'s input tape, and the output values of $\mathcal{A}$ are copied to $\mathcal{A}'$'s own output tape.

Simulating the "random tape generation" phase: When the first activation of Π takes place, $\mathcal{A}'$ simulates the random tape generation phase of $\mathsf{Comp}(\Pi)$ for $\mathcal{A}$. We describe $\mathcal{A}'$'s simulation of the random tape generation for a party P_j (this simulation is repeated for every j). We differentiate between the case that P_j is honest and P_j is corrupted:

1. *Party P_j is not corrupted:* $\mathcal{A}'$ hands $\mathcal{A}$ the message $(\mathsf{receipt}, sid_j, P_j, \mathcal{P})$ from $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$ (this refers to P_j's commitment). In addition, $\mathcal{A}'$ simulates all the $(\mathsf{receipt}, sid_{k,j}, P_k, \mathcal{P})$ messages that $\mathcal{A}'$ expects to receive from $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$. $\mathcal{A}'$ completes the simulation by passing $\mathcal{A}$ the "decommit" messages $(\mathsf{CP\text{-}proof}, sid_{k,j}, r_k^j)$ for every uncorrupted party P_k. $\mathcal{A}'$ also obtains $(\mathsf{commit}, sid_{k,j}, r_k^j)$ messages from $\mathcal{A}$ for the corrupted parties P_k as well as the respective decommit messages $(\mathsf{CP\text{-}prover}, sid_{k,j}, r_k^j)$. $\mathcal{A}'$ computes s_j as in the protocol and records this value.

2. *Party P_j is corrupted:* Let r_j be the random tape of the semi-honest party P_j in protocol Π. Now, as above, for every uncorrupted party P_k, adversary $\mathcal{A}'$ passes $\mathcal{A}$ the $(\mathsf{receipt}, sid_{k,j}, P_k, \mathcal{P})$ messages that $\mathcal{A}$ expects to receive from $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$. $\mathcal{A}'$ also obtains $(\mathsf{commit}, sid_{k,j}, r_k^j)$ messages from $\mathcal{A}$ (for every corrupt P_k) and corrupted P_j's commitment $(\mathsf{commit}, sid_j, r_j^j)$. Notice that at this point, $\mathcal{A}$ is bound to all the r_k^j values of the corrupted parties, whereas $\mathcal{A}'$ is still free to choose the analogous values for the uncorrupted parties. Therefore, in the "decommitment" part of the phase, $\mathcal{A}'$ chooses the uncorrupted parties' values so that $\bigoplus_{k=1}^{\ell} r_k^j = r_j$ where r_j is the random tape of the external P_j in Π. (Thus, $\mathcal{A}'$ forces $\mathcal{A}$ into using r_j for the malicious P_j in $\mathsf{Comp}(\Pi)$ as well.)

Simulating an activation due to a new input: When the first message of an activation of Π is sent, $\mathcal{A}'$ internally simulates for $\mathcal{A}$ the appropriate stage in $\mathsf{Comp}(\Pi)$. This is done as follows. Let P_i be the activated party with a new input. If P_i is not corrupted, then $\mathcal{A}'$ internally passes $\mathcal{A}$ the message $(\mathsf{receipt}, sid_i, P_i, \mathcal{P})$ that $\mathcal{A}$ expects to receive from $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$. If P_i is corrupted, then $\mathcal{A}'$ receives a message $(\mathsf{commit}, sid_i, \mathcal{P}, x)$ from $\mathcal{A}$ (who controls P_i). $\mathcal{A}'$ adds x to its list $\overline{x}_i$ of inputs received from P_i and passes $\mathcal{A}$ the string $(\mathsf{receipt}, sid_i i, P_i, \mathcal{P})$. Furthermore, $\mathcal{A}'$ sets P_i's input tape to equal x. (Recall that in the semi-honest model, $\mathcal{A}'$ can modify the input that the environment writes on a corrupted party's input tape.)

Dealing with messages sent by honest parties: If an uncorrupted party P_i sends a message m in Π to a corrupted party (controlled by $\mathcal{A}'$), then $\mathcal{A}'$ prepares a simulated message of $\mathsf{Comp}(\Pi)$ to give to $\mathcal{A}$. Specifically, $\mathcal{A}'$ passes $\mathcal{A}$ the message $(\mathsf{CP\text{-}proof}, sid_i, (m, s_i, \overline{m}))$ as expected from $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$.

Dealing with messages sent by corrupted parties: When $\mathcal{A}$ sends a $\mathsf{Comp}(\Pi)$-message from a corrupted party, $\mathcal{A}'$ translates this to the appropriate message in Π. That is, $\mathcal{A}'$ obtains a message $(\mathsf{CP\text{-}prover}, sid_i, (m, s_i', \overline{m}))$ from $\mathcal{A}$ controlling a corrupted party P_i. $\mathcal{A}'$ checks that the series of broadcasted Π-messages is indeed $\overline{m}$. $\mathcal{A}'$ also checks that $s_i' = s_i$, where s_i is the value defined in the random tape generation phase. Finally, $\mathcal{A}'$ checks that $m = \Pi(\overline{x}_i, s_i \oplus r_i^i, \overline{m})$. If yes, then it delivers the message written on semi-honest party P_i's outgoing communication tape in Π. Otherwise, $\mathcal{A}'$ does nothing.

Dealing with corruption of parties: When the simulated $\mathcal{A}$ internally corrupts a party P_i, adversary $\mathcal{A}'$ externally corrupts P_i and obtains all of its past inputs, outputs and random tapes in Π. Then, $\mathcal{A}'$ prepares a simulated internal state of P_i in $\mathsf{Comp}(\Pi)$. The only additional state that P_i has in $\mathsf{Comp}(\Pi)$ is the random string r_i^i for the random tape generation phase. Since the string s_i is public and fixed, $\mathcal{A}'$ sets r_i^i so that $r_i = s_i \oplus r_i^i$, where r_i is P_i's random tape in Π.

We now claim that $\mathcal{Z}$'s view of an interaction with $\mathcal{A}'$ and Π is distributed identically to its view of an interaction with $\mathcal{A}$ and $\mathsf{Comp}(\Pi)$. This follows from the same observations as in the two-party case. The key points are as follows. For every corrupted P_i, the semi-honest adversary $\mathcal{A}'$ can force $\mathcal{A}$ into using the exact random tape of P_i in Π. Furthermore, any modification of the inputs made by $\mathcal{A}$ can also be carried out by $\mathcal{A}'$. We therefore have that if $\mathcal{A}$ follows the protocol specification with respect to these inputs and random tapes, then the semi-honest parties in Π will send exactly the same messages as the malicious parties in $\mathsf{Comp}(\Pi)$. The proof is concluded by observing that $\mathcal{A}$ must follow the protocol specification because the $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ functionality enforces this. Thus, $\mathcal{A}'$'s checks of correctness in the simulation perfectly simulate the behavior of $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ in a hybrid execution. Finally, the internal state revealed to $\mathcal{A}$ in the case of a corruption is exactly as it expects to see. This completes the proof. ■

Conclusions

By combining Propositions 4.9.5 and 4.9.6, and using the universal composition theorem (Theorem 4.3.3), we obtain that $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ can be securely realized in the $(\mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{MCOM}}^{1:\text{M}})$-hybrid model, assuming the existence of trapdoor permutations. Then, combining this with Proposition 4.9.7 we obtain a protocol compiler in the same hybrid model and under the same assumptions. Applying this compiler to Proposition 4.9.1, we derive universally composable multi-party computation in the presence of malicious adversaries. That is:

Theorem 4.9.8 *Assume that trapdoor permutations exist. Then, for any well-formed multi-party ideal functionality $\mathcal{F}$, there exists a non-trivial protocol that securely realizes $\mathcal{F}$ in the $(\mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{MCOM}}^{1:\text{M}})$-hybrid model in the presence of* malicious, static *adversaries. Furthermore, if two-party augmented non-committing encryption protocols exist, then for any adaptively well-formed multi-party ideal functionality $\mathcal{F}$, there exists a non-trivial protocol that securely realizes $\mathcal{F}$ in the $(\mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{MCOM}}^{1:\text{M}})$-hybrid model in the presence of* malicious, adaptive *adversaries.*

By combining Theorem 4.9.8 and Proposition 4.9.4, and using the universal composition with joint state theorem (Theorem 4.3.4), we obtain:

Theorem 4.9.9 (Theorem 4.2.3 – restated): *Assume that trapdoor permutations exist. Then, for any well-formed multi-party ideal functionality $\mathcal{F}$, there exists a non-trivial protocol that securely realizes $\mathcal{F}$ in the $(\mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{CRS}})$-hybrid model in the presence of* malicious, static *adversaries. Furthermore, if two-party augmented non-committing encryption protocols exist, then for any adaptively well-formed multi-party ideal functionality $\mathcal{F}$, there exists a non-trivial protocol that securely realizes $\mathcal{F}$ in the $(\mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{CRS}})$-hybrid model in the presence of* malicious, adaptive *adversaries. In both cases, the protocol uses only a single copy of $\mathcal{F}_{\text{CRS}}$.*

Appendix A

Abstracts of Additional Results

In this appendix we present abstracts of additional work done by the author during graduate studies at the Weizmann Institute of Science.

A.1 Secure Protocols

Session-Key Generation using Human Passwords Only [50]: In this paper, we present session-key generation protocols in a model where the legitimate parties share *only* a human-memorizable password. The security guarantee holds with respect to probabilistic polynomial-time adversaries that control the communication channel (between the parties), and may omit, insert and modify messages at their choice. Loosely speaking, the effect of such an adversary that attacks an execution of our protocol is comparable to an attack in which an adversary is only allowed to make a constant number of queries of the form "is w the password of Party A". We stress that the result holds also in case the passwords are selected at random from a small dictionary so that it is feasible (for the adversary) to scan the entire directory. We note that prior to our result, it was not clear whether or not such protocols were attainable without the use of random oracles or additional setup assumptions.

Joint work with Oded Goldreich.

Parallel Coin-Tossing and Constant-Round Secure Two-Party Computation [64]: In this paper we show that any *two-party* functionality can be securely computed in a *constant number of rounds*, where security is obtained against (polynomial-time) malicious adversaries that may arbitrarily deviate from the protocol specification. This is in contrast to Yao's constant-round protocol that ensures security only in the face of semi-honest adversaries, and to its malicious adversary version that requires a polynomial number of rounds.

In order to obtain our result, we present a constant-round protocol for secure coin-tossing of polynomially many coins (in parallel). We then show how this protocol can be used in conjunction with other existing constructions in order to obtain a constant-round protocol for securely computing any two-party functionality. On the subject of coin-tossing, we also present a constant-round *almost perfect* coin-tossing protocol, where by "almost perfect" we mean that the resulting coins are guaranteed to be statistically close to uniform (and not just pseudorandom).

Privacy Preserving Data Mining [68]: This paper addresses the issue of privacy preserving data mining. Specifically, we consider a scenario in which two parties owning confidential databases wish to run a data mining algorithm on the union of their databases, without revealing any unnecessary information. Our work is motivated by the need to both protect privileged information and enable its use for research or other purposes.

The above problem is a specific example of secure multi-party computation and as such, can be solved using known generic protocols. However, data mining algorithms are typically complex and, furthermore, the input usually consists of massive data sets. The generic protocols in such a case are of no practical use and therefore more efficient protocols are required. We focus on the problem of decision tree learning with the popular ID3 algorithm. Our protocol is considerably more efficient than generic solutions and demands both very few rounds of communication and reasonable bandwidth.

Joint work with Benny Pinkas.

Sequential Composition of Protocols without Simultaneous Termination [67]: The question of the composition of protocols is an important and heavily researched one. In this paper we consider the problem of sequential composition of synchronous protocols that do not have simultaneous termination; i.e., the parties do not necessarily conclude a protocol execution in the same round. A problem arises becauses such protocols must begin in synchrony; therefore a second execution cannot follow from the first in a straightforward manner. An important example of a protocol with this property is that of randomized Byzantine Agreement with an expected constant number of rounds (such as the one due to Feldman and Micali). We note that expected constant-round Byzantine Agreement *cannot* have simultaneous termination and thus this (problematic) property is inherent.

Given that the termination of the parties is not simultaneous, a natural question to consider is how to synchronize the parties so that such protocols can be sequentially composed. Furthermore, such a composition should preserve the original running-time of the protocol, i.e. running the protocol ℓ times sequentially should take in the order of ℓ times the running-time of the protocol. In this paper, we present a method for sequentially composing any protocol in which the players do not terminate in the same round, while preserving the original round complexity. An important application of this result is the sequential composition of parallel Byzantine Agreement. Such a composition can be used by parties connected in a point-to-point network to run protocols designed for the broadcast model, while maintaining the original round complexity.

Joint work with Anna Lysyanskaya and Tal Rabin.

A.2 Zero-Knowledge

Resettably-Sound Zero-Knowledge and Its Applications [2]: Resettably-sound proofs and arguments maintain soundness even when the prover can reset the verifier to use the same random coins in repeated executions of the protocol. We show that resettably-sound zero-knowledge arguments for $\mathcal{NP}$ exist if collision-free hash functions exist. In contrast, resettably-sound zero-knowledge proofs are possible only for languages in $\mathcal{P}/\text{poly}$.

We present two applications of resettably-sound zero-knowledge arguments. First, we construct resettable zero-knowledge arguments of knowledge for $\mathcal{NP}$, using a natural relaxation of the definition of arguments (and proofs) of knowledge. We note that, under the standard definition of proof of knowledge, it is impossible to obtain resettable zero-knowledge arguments of knowledge for languages outside $\mathcal{BPP}$. Second, we construct a constant-round resettable zero-knowledge argument for $\mathcal{NP}$ in the public-key model, under the assumption that collision-free hash functions exist. This improves upon the sub-exponential hardness assumption required by previous constructions.

We emphasize that our results use non-black-box zero-knowledge simulations. Indeed, we show that some of the results are *impossible* to achieve using black-box simulations. In particular, only languages in $\mathcal{BPP}$ have resettably-sound arguments that are zero-knowledge with respect to black-box simulation.

Joint work with Boaz Barak, Oded Goldreich and Shafi Goldwasser.

Strict Polynomial-Time in Simulation and Extraction [3]: The notion of efficient computation is usually identified in cryptography and complexity with probabilistic polynomial time. However, until recently, in order to obtain *constant-round* zero-knowledge proofs and proofs of knowledge, one had to allow simulators and knowledge-extractors to run in time which is only polynomial *on the average* (i.e., *expected* polynomial time). Whether or not allowing expected polynomial-time is *necessary* for obtaining constant-round zero-knowledge proofs and proofs of knowledge, has been posed as an important open question. This question is interesting not only for its theoretical ramifications, but also because expected polynomial time simulation is not closed under composition. Therefore, in some cases security may not be maintained when a protocol that utilizes expected polynomial time simulation (or extraction) is used as a part of a larger protocol.

A partial answer to the question of the necessity (or non-necessity) of expected polynomial-time was provided recently by Barak, who gave the first constant-round zero-knowledge argument with a *strict* (in contrast to expected) polynomial-time simulator. His was also the first protocol that is *not* black-box zero-knowledge. That is, the simulator in his protocol makes inherent use of the description of the *code* of the verifier.

In this paper, we completely resolve the question of expected polynomial-time in constant-round zero-knowledge arguments and arguments of knowledge. First, we show that there exist constant-round zero-knowledge arguments of knowledge with strict polynomial-time *extractors*. As in the simulator of Barak's zero-knowledge protocol, the extractor for our proof of knowledge is not black-box and makes inherent use of the code of the prover. On the negative side, we show that non-black-box techniques are *essential* for both strict polynomial-time simulation and extraction. That is, we show that no constant-round zero-knowledge argument (or proof) can have a strict polynomial-time *black-box* simulator. Similarly, we show that no constant-round zero-knowledge argument of knowledge can have a strict polynomial-time *black-box* knowledge extractor. Thus, for constant-round black-box zero-knowledge arguments (resp., arguments of knowledge), it is imperative that the simulator (resp., extractor) be allowed to run in expected polynomial-time.

Joint work with Boaz Barak.

A.3 Miscellaneous

A Simpler Construction of CCA2-Secure Public-Key Encryption Under General Assumptions [65]: In this paper we present a simpler construction of an encryption scheme that achieves adaptive chosen ciphertext security (CCA2), assuming the existence of trapdoor permutations. We build on previous works of Sahai and De Santis et al. and construct a scheme that we believe is the easiest to understand to date. In particular, it is only slightly more involved than the Naor-Yung encryption scheme that is secure against passive chosen-ciphertext attacks (CCA1). We stress that the focus of this paper is on *simplicity* only.

Bibliography

[1] B. Barak. How to go beyond the black-box simulation barrier. In *42nd FOCS*, pages 106–115, 2001.

[2] B. Barak, O. Goldreich, S. Goldwasser and Y. Lindell. Resettably-Sound Zero-Knowledge and its Applications. In *42nd FOCS*, pages 116–125, 2001.

[3] B. Barak and Y. Lindell. Strict Polynomial-Time in Simulation and Extraction. In *34th STOC*, pages 484–493, 2002.

[4] D. Beaver. Foundations of Secure Interactive Computing. In *CRYPTO'91*, Springer-Verlag (LNCS 576), pages 377–391, 1991.

[5] D. Beaver. Secure Multi-party Protocols and Zero-Knowledge Proof Systems Tolerating a Faulty Minority. *Journal of Cryptology,* Springer-Verlag, 4:75–122, 1991.

[6] D. Beaver. Plug and play encryption. In *CRYPTO'97,* Springer-Verlag (LNCS 1294), pages 75–89, 1997.

[7] M. Bellare, A. Boldyreva, and S. Micali. Public-Key Encryption in a Multi-user Setting: Security Proofs and Improvements. *Eurocrypt'00,* Springer-Verlag (LNCS 1807), pages 259–274, 2000.

[8] D. Beaver and S. Goldwasser. Multiparty Computation with Fault Majority. In *CRYPTO'89*, Springer-Verlag (LNCS 435), 1989.

[9] D. Beaver, S. Micali and P. Rogaway. The Round Complexity of Secure Protocols. In *22nd STOC*, pages 503–513, 1990.

[10] M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. *20th STOC,* pages 1–10, 1988.

[11] M. Blum How to Prove a Theorem So No One Else Can Claim It. *Proceedings of the International Congress of Mathematicians*, Berkeley, California, USA, 1986, pp. 1444-1451.

[12] M. Blum, P. Feldman and S. Micali. Non-interactive zero-knowledge and its applications. In *20th STOC,* pages 103–112, 1988.

[13] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[14] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In 42nd *FOCS*, pages 136–145. 2001. Full version available at `http://eprint.iacr.org/2000/067`.

[15] R. Canetti, U. Feige, O. Goldreich and M. Naor. Adaptively Secure Multi-Party Computation. In 28th *STOC*, pages 639–648, 1996.

[16] R. Canetti and M. Fischlin. Universally Composable Commitments. In *CRYPTO'01*, Springer-Verlag (LNCS 2139), pages 19–40, 2001.

[17] R. Canetti, J. Kilian, E. Petrank, and A. Rosen. Black-Box Concurrent Zero-Knowledge Requires $\tilde{\Omega}(\log n)$ Rounds. In 33th *STOC*, pages 570–579. 2001.

[18] R. Canetti and H. Krawczyk. Universally Composable Notions of Key-Exchange and Secure Channels. In *Eurocrypt'02*, Springer-Verlag (LNCS 2332), pages 337–351, 2002.

[19] R. Canetti, E. Kushilevitz and Y. Lindell. On the Limitations of Universal Composition in the Standard Model. Work in progress, 2002.

[20] R. Canetti, Y. Lindell, R. Ostrovsky and A. Sahai. Universally Composable Two-Party and Multi-Party Computation. In 34th *STOC*, pages 494–503, 2002.

[21] R. Canetti and T. Rabin. Universal Composition with Joint State. *Cryptology ePrint Archive,* Report 2002/047, `http://eprint.iacr.org/`, 2002.

[22] D. Chaum, C. Crepeau and I. Damgard. Multi-party Unconditionally Secure Protocols. In 20th *STOC*, pages 11–19, 1988.

[23] R. Cleve. Limits on the Security of Coin Flips when Half the Processors are Faulty. In 18th *STOC,* pages 364–369, 1986.

[24] I. Damgard and J.B. Nielsen. Improved non-committing encryption schemes based on general complexity assumptions. In *CRYPTO'00*, Springer-Verlag (LNCS 1880), pages 432–450.

[25] I. Damgard and J. Nielsen. Perfect Hiding or Perfect Binding Universally Composable Commitment Schemes with Constant Expansion Factor. To appear in *CRYPTO'02,* 2002.

[26] A. De Santis, G. Di Crescenzo, R. Ostrovsky, G. Persiano, A. Sahai. Robust Non-interactive Zero-Knowledge. In *CRYPTO'01,* Springer-Verlag (LNCS 2139), pages 566–598, 2001.

[27] A. De Santis and G. Persiano. Zero-Knowledge Proofs of Knowledge Without Interaction. In 33rd *FOCS,* pages 427–436, 1992.

[28] G. Di Crescenzo, Y. Ishai, and R. Ostrovsky. Non-Interactive and Non-Malleable Commitment. In 30th *STOC,* pages 141–150, 1998.

[29] G. Di Crescenzo, J. Katz, R. Ostrovsky and A. Smith. Efficient and Non-interactive Non-malleable Commitment. In *Eurocrypt'01*, Springer-Verlag (LNCS 2045), pages 40–59, 2001.

[30] Y. Dodis and S. Micali. Parallel Reducibility for Information-Theoretically Secure Computation. In *Crypto'00*, Springer-Verlag (LNCS 1880), pages 74–92, 2000.

[31] D. Dolev. The Byzantine Generals Strike Again. *Journal of Algorithms,* 3(1):14–30, 1982.

[32] D. Dolev, C. Dwork and M. Naor. Non-malleable cryptography. *SIAM Journal of Computing,* 30(2):391–437, 2000.

[33] C. Dwork, M. Naor, and A. Sahai. Concurrent Zero-Knowledge. In 30*th STOC*, pages 409–418, 1998.

[34] D. Dolev and H.R. Strong. Authenticated Algorithms for Byzantine Agreement. *SIAM Journal of Computing*, 12(4):656–665, 1983.

[35] S. Even, O. Goldreich and A. Lempel. A randomized protocol for signing contracts. In *Communications of the ACM,* 28(6):637–647, 1985.

[36] U. Feige and A. Shamir. Zero-Knowledge Proofs of Knowledge in Two Rounds. In *CRYPTO'89*, Springer-Verlag (LNCS 435), pages 526–544, 1989.

[37] P. Feldman and S. Micali. An Optimal Algorithm for Synchronous Byzantine Agreement. *SIAM Journal of Computing*, 26(2):873–933, 1997.

[38] M. Fischer, N. Lynch, and M. Merritt. Easy Impossibility Proofs for Distributed Consensus Problems. *Distributed Computing*, 1(1):26–39, 1986.

[39] M. Fitzi, N. Gisin, U. Maurer and O. Von Rotz. Unconditional Byzantine Agreement and Multi-Party Computation Secure Against Dishonest Minorities from Scratch. In *Eurocrypt'02,* Springer-Verlag (LNCS 2332), pages 482–501, 2002.

[40] M. Fitzi, D. Gottesman, M. Hirt, T. Holenstein and A. Smith. Byzantine Agreement Secure Against Faulty Majorities From Scratch. To appear in the 21*st PODC,* 2002.

[41] M. Fitzi and U. Maurer. From partial consistency to global broadcast. In 32*th STOC*, pages 494–503. 2000.

[42] Z. Galil, S. Haber and M. Yung. Cryptographic Computation: Secure Fault Tolerant Protocols and the Public Key Model. In *CRYPTO'87,* Springer-Verlag (LNCS 293), pages 135–155, 1987.

[43] J. Garay and P. Mackenzie. Concurrent Oblivious Transfer. In 41*st FOCS*, pages 314–324, 2000.

[44] O. Goldreich. *Secure Multi-Party Computation.* Manuscript. Preliminary version, 1998. Available from `http://www.wisdom.weizmann.ac.il/~oded/pp.html`.

[45] O. Goldreich. *Foundations of Cryptography: Volume 1 – Basic Tools.* Cambridge University Press, 2001.

[46] O. Goldreich. Concurrent Zero-Knowledge With Timing Revisited. In 34*th STOC*, pages 332–340, 2002.

[47] O. Goldreich and A. Kahan. How To Construct Constant-Round Zero-Knowledge Proof Systems for NP. *Journal of Cryptology*, 9(3):167–190, 1996.

[48] O. Goldreich and H. Krawczyk. On the composition of zero-knowledge proof systems. *SIAM Journal of Computing*, 25(1):169–192, 1996.

[49] O. Goldreich and L. Levin. A Hard Predicate for All One-way Functions. In 21*st STOC,* pages 25–32, 1989.

[50] O. Goldreich and Y. Lindell. Session-Key Generation using Human Passwords Only. In *CRYPTO'01,* Springer-Verlag (LNCS 2139), pages 408–432, 2001.

[51] O. Goldreich and Y. Oren. Definitions and Properties of Zero-Knowledge Proof Systems. *Journal of Cryptology,* 7(1):1–32, 1994.

[52] O. Goldreich, S. Micali and A. Wigderson. How to Play any Mental Game – A Completeness Theorem for Protocols with Honest Majority. In 19*th STOC,* pages 218–229, 1987. For details see [44].

[53] S. Goldwasser and L. Levin. Fair Computation of General Functions in Presence of Immoral Majority. In *CRYPTO'90,* Springer-Verlag (LNCS 537), pages 77–93, 1990.

[54] S. Goldwasser and Y. Lindell. Secure Computation Without Agreement. To appear in the 16*th DISC,* 2002.

[55] S. Goldwasser, S. Micali and C. Rackoff The Knowledge Complexity of Interactive Proof Systems. *SIAM Journal of Computing,* 18(1):186–208, 1989.

[56] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing*, 17(2):281–308, 1988.

[57] L. Gong, P. Lincoln, and J. Rushby. Byzantine Agreement with Authentication: Observations and Applications in Tolerating Hybrid and Link Faults. In *Dependable Computing for Critical Applications*, pages 139–157, 1995.

[58] J.N. Gray. Notes on data base operating systems. In *Operating Systems: An Advanced Course,* Springer-Verlag (LNCS 60), chapter 3.F, page 465, 1978.

[59] J. Kilian. Uses of Randomness in Algorithms and Protocols. The ACM Distinguished Dissertation 1989, MIT press.

[60] J. Kilian and E. Petrank. Concurrent and resettable zero-knowledge in poly-logartihmic rounds. In 33*rd STOC,* pages 560–569, 2001.

[61] J. Kilian, E. Petrank and C. Rackoff. Lower Bounds for Zero Knowledge on the Internet. In 39*th FOCS,* pages 484–492, 1998.

[62] L. Lamport, R. Shostack, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[63] L. Lamport. The weak byzantine generals problem. In *Journal of the ACM*, 30(3):668–676, 1983.

[64] Y. Lindell. Parallel Coin-Tossing and Constant-Round Secure Two-Party Computation. In *CRYPTO'01,* Springer-Verlag (LNCS 2139), pages 171–189, 2001.

[65] Y. Lindell. A Simpler Construction of CCA2-Secure Public-Key Encryption Under General Assumptions. Technical Report MCS02-10, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, May 2002.

[66] Y. Lindell, A. Lysysanskaya and T. Rabin. On the Composition of Authenticated Byzantine Agreement. In 34*th STOC*, pages 514–523, 2002.

[67] Y. Lindell, A. Lysyanskaya and T. Rabin. Sequential Composition of Protocols without Simultaneous Termination. To appear in 21*st PODC*, 2002.

[68] Y. Lindell and B. Pinkas. Privacy Preserving Data Mining. *Journal of Cryptology,* 15(3):177–206, 2002.

[69] S. Micali and P. Rogaway. Secure computation. Unpublished manuscript, 1992. Preliminary version in *CRYPTO'91*, Springer-Verlag (LNCS 576), pages 392–404, 1991.

[70] M. Naor. Bit Commitment using Pseudorandom Generators. *Journal of Cryptology,* 4(2):151–158, 1991.

[71] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, 1980.

[72] B. Pfitzmann and M. Waidner. Information-Theoretic Pseudosignatures and Byzantine Agreement for $t >= n/3$. Technical Report RZ 2882 (#90830), IBM Research, 1996.

[73] B. Pfitzmann and M. Waidner. Composition and integrity preservation of secure reactive systems. *7th ACM Conference on Computer and Communication Security*, 2000, pp. 245-254.

[74] M. Prabhakaran, A. Rosen and A. Sahai. Concurrent Zero Knowledge With Logarithmic Round Complexity. To appear in the 43*rd FOCS,* 2002.

[75] M. Rabin. How to exchange secrets by oblivious transfer. Tech. Memo TR-81, Aiken Computation Laboratory, Harvard U., 1981.

[76] T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multi-party Protocols with Honest Majority. 21*st STOC*, pages 73–85, 1989.

[77] R. Richardson and J. Kilian. On the Concurrent Composition of Zero-Knowledge Proofs. In *Eurocrypt'99*, Springer-Verlag (LNCS 1592), pages 415–413, 1999.

[78] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[79] A. Rosen. A Note on the Round-Complexity of Concurrent Zero-Knowledge. In *CRYPTO'00,* Springer-Verlag (LNCS 1880), pages 451–468, 2000.

[80] A. Sahai. Non-Malleable Non-Interactive Zero-Knowledge and Adaptive Chosen-Ciphertext Security. In 40*th FOCS*, pages 543–553, 1999.

[81] A. Yao. How to Generate and Exchange Secrets. In 27*th FOCS*, pages 162–167, 1986.