Universally Composable Two-Party and Multi-Party Secure Computation*

Ran Canetti† Yehuda Lindell‡ Rafail Ostrovsky§ Amit Sahai¶

July 14, 2003

Abstract

We show how to securely realize any two-party and multi-party functionality in a *universally composable* way, regardless of the number of corrupted participants. That is, we consider an asynchronous multi-party network with open communication and an adversary that can adaptively corrupt as many parties as it wishes. In this setting, our protocols allow any subset of the parties (with pairs of parties being a special case) to securely realize any desired functionality of their local inputs, and be guaranteed that security is preserved regardless of the activity in the rest of the network. This implies that security is preserved under concurrent composition of an unbounded number of protocol executions, it implies non-malleability with respect to arbitrary protocols, and more. Our constructions are in the common reference string model and rely on standard intractability assumptions.

Keywords: Two-party and multi-party cryptographic protocols, secure composition of protocols, proofs of security.

---

*An extended abstract of this work appeared in the 34^{th} *STOC*, 2002.

†IBM T.J. Watson Research Center, email: `canetti@watson.ibm.com`.

‡IBM T.J. Watson Research Center, email: `lindell@us.ibm.com`. Most of this work was carried out while the author was at the Weizmann Institute of Science, Israel.

§Telcordia Technologies, email: `rafail@research.telcordia.com`.

¶Princeton University, email: `sahai@cs.princeton.edu`.

Contents

1 Introduction

Traditionally, cryptographic protocol problems were considered in a model where the only involved parties are the actual participants in the protocol, and only a single execution of the protocol takes place. This model allowed for relatively concise problem statements, and simplified the design and analysis of protocols. Indeed, this relatively simple model is a natural choice for the initial study of protocols.

However, this model of "stand-alone computation" does not really capture the security requirements from cryptographic protocols in modern computer networks. 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 [DDN00]. 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.

An alternative, and arguably preferable, approach is to design and analyze protocols as "stand alone", and then derive security in a multi-execution environment via a *secure composition theorem.* This is the approach taken by the recently proposed framework of universally composable security [C01]. 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 defining the protocol problem at hand. In addition, it has been 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. A protocol that is secure within the [C01] framework is called universally composable (UC), and we say that the protocol UC realizes the given functionality.

It has been shown that *any* ideal functionality can be UC realized using known constructions, as long as a majority of the participants are honest [C01] (building upon [BGW88, RB89, CFGN96]). 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 has been shown that in the plain model (i.e., where no trusted setup phase is assumed), the important two-party commitment and zero-knowledge functionalities cannot be UC realized by *any* two-party protocol [C01, CF01]. Nonetheless, protocols that UC realize the commitment and zero-knowledge functionalities in the common reference string (CRS) model were shown in [CF01, $\text{d}^{+}01$]. (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 [BFM88] and since then has proved useful in other cases as well.)

Our results. Loosely speaking, we show that any functionality can be UC realized 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 UC realize any functionality of their inputs. The functionality may be reactive, namely it may receive inputs and generate outputs multiple times throughout the computation. In addition to a common reference string, our protocols assume that the participants in each protocol execution have access to a broadcast channel among themselves.[1]

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 UC realize the ideal commitment functionality [CF01]. Existing constructions of UC commitments [CF01, DN02] rely on specific cryptographic assumptions.) Since UC zero-knowledge can be obtained given a UC commitment scheme [CF01], we can plug our new scheme into the UC zero-knowledge protocol of [CF01] 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. A high level outline of our construction appears in Section 2.2.

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 static adversarial case (both semi-honest and malicious) we assume the existence of enhanced trapdoor permutations[2] only (this is the same assumption used for known stand-alone constructions). For the adaptive case we also assume the existence of *augmented* non-committing encryption protocols [CFGN96]. The augmentation includes oblivious key generation and invertible samplability [DN00]. 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

[1] This broadcast channel is formally modeled by a universally composable broadcast functionality. In subsequent work to ours, it was shown that in the model where delivery of messages is not guaranteed, universally composable broadcast can be achieved in $O(1)$ rounds, for any number of corrupted parties, and without any setup assumptions [GL02]. Thus, in actuality, we only need to assume a common reference string here.

[2] Enhanced trapdoor permutations have the property that a random element generated by the domain sampler is hard to invert, even given the random coins used by the sampler. Note that any trapdoor permutation over $\{0,1\}^k$ is clearly enhanced, because this domain can be easily and directly sampled. See [G03, Appendix C] for a full discussion on enhanced trapdoor permutations and why they are needed.

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. We note that in both the static and adaptive cases, most of our constructions can be obtained assuming (plain) trapdoor permutations only. The additional assumption of enhanced trapdoor permutations is used for UC realizing the oblivious transfer functionality in the case of semi-honest, static adversaries, and for our construction of UC commitments for the case of malicious, adaptive adversaries. The additional assumption of augmented non-committing encryption is used for UC realizing the oblivious transfer functionality in the semi-honest, adaptive case.

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 [DP92].

Subsequent work. As we have mentioned, it has previously been shown the commitment and zero-knowledge functionalities cannot be UC realized in the plain model [C01, CF01]. Subsequently, broad impossibility results were shown, demonstrating that large classes of two-party functionalities cannot be UC realized in the plain model [CKL03]. Thus, some setup assumption, like that of a common reference string assumed here, is essential for obtaining UC security in the case of no honest majority. Another subsequent work [L03] has shown that the impossibility results of [CKL03] hold for *any* definition that implies security under the composition operation considered by the UC framework. Thus, in the plain model and with no honest majority, it is impossible to obtain security in a setting where protocols are run concurrently with arbitrary other protocols. Therefore, when this level of security is desired, some setup assumption is needed. We believe that this provides a strong justification for assuming a common reference string, as we do in this work.

Organization. In Section 2 we provide an overview of the model of [C01] and an outline of our construction of UC two-party and multi-party protocols. Section 3 contains a number of preliminaries: First, in Section 3.1, a more detailed description of the [C01] framework and of the composition theorem is presented. Then, in Section 3.2, the issue of universal composition with joint state is discussed (this is important when a common reference string is used, as is the case in our constructions). Finally, in Section 3.3, we describe the class of ideal functionalities for which we present UC secure protocols.

We then begin our constructions with the two-party case. First, in Section 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. Here we lead up to the general protocol compiler in a number of steps: In Section 5 we recall the commitment functionality $\mathcal{F}_{\text{MCOM}}$ and present our new UC commitment scheme. In Section 6, the ideal zero-knowledge functionality, $\mathcal{F}_{\text{ZK}}$, is described and known protocols for realizing it (either with ideal access to $\mathcal{F}_{\text{MCOM}}$ or directly in the common reference string model) are recalled. In Section 7 we define the two-party commit-and-prove functionality, $\mathcal{F}_{\text{CP}}$, and show how to realize it given ideal access to $\mathcal{F}_{\text{ZK}}$. This is then used in Section 8 to construct a two-party protocol compiler that transforms the protocol of Section 4 into a protocol that is secure against malicious adversaries.

Finally, in Section 9, we extend our two-party constructions to the multi-party case. We present the two-party case separately because it is simpler and most of the cryptographic ideas already arise in this setting.

2 Overview

This section provides a high-level overview of the model and our constructions. Section 2.1 contains an overview of the general framework of universal composability, the definition of security and the composition theorem. Then, in Section 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.

2.1 The Model

We begin by outlining the framework for universal composability; for more details see Section 3.1 and [C01]. 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., [GL90, MR91, B91, PW00, C00]), 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. Since the trusted party just runs the ideal functionality, we do not distinguish between them. Rather, we refer to interaction between the parties and the *functionality*. Informally, a protocol securely carries out a given task if no adversary can gain 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, it is required 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 UC 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 [C01] 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 [C01]: 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 UC 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 UC realizes some ideal functionality $\mathcal{G}$ in the $\mathcal{F}$-hybrid model, then π^ρ UC 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.

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 [GMW87, G98]. This construction (which we call the GMW construction) is comprised of two stages. First, they present a protocol for UC 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.

2.2.1 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 [GMW87, G98] 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 simple 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 [EGL85]). 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 of the output gates 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 UC 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 UC realizing $\mathcal{F}_{\text{OT}}$ itself. (Our overall 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 UC realize $\mathcal{F}_{\text{OT}}$ in the semi-honest case. In the static (i.e., non-adaptive) case, the protocol of [EGL85, G98] suffices. In the adaptive case, our protocol uses an augmented version of non-committing encryption [CFGN96]. 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 [DN00], 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 2.1 (semi-honest computation – informal): *Assume that enhanced*[3] *trapdoor permutations exist. Then, for any two-party ideal functionality $\mathcal{F}$, there exists a protocol Π that UC realizes $\mathcal{F}$ in the presence of* semi-honest, static *adversaries. Furthermore, if two-party augmented non-committing encryption protocols exist, then there exists a protocol Π that UC realizes $\mathcal{F}$ in the presence of* semi-honest, adaptive *adversaries.*

Proposition 2.1 as stated above is not precise. This is due to two technicalities regarding the model of computation as defined in [C01]. 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 3.3 for more discussion and an exact definition.

Another point where our results formally differ from Proposition 2.1 is due to the fact that, according to the definitions used here, protocols which do not generate any output are technically

[3] See Footnote 2.

secure (for any functionality). Thus, Proposition 2.1 as stated, can be easily (but un-interestingly) achieved. In contrast, we prove the existence of protocols which *do* generate output and UC realize any functionality (we call such a protocol non-trivial; for more details, see the discussion after Definition 3.2 in Section 3.1). Proposition 2.1 is formally restated in Section 4.2.

2.2.2 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. Further observe that each party holds a commitment to the input and random tape of the other party and 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 that was committed to. (Instead, the recipient receives only a formal "receipt" assuring it that a value was committed to. See Section 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. This notion is implicit in the work of [GMW87], and was also discussed by Kilian [K89]. We formulate this notion in a universally composable commit-and-prove functionality,[4] 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 UC 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 [N91] is sufficient as an instantiation of the scheme C. We thus obtain a protocol for UC 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 as-

[4]In a concurrent and independent work [DN02], 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.

suming the existence of enhanced trapdoor permutations, as is demonstrated by our construction of universally composable commitments in Section 5. In all we obtain:

Theorem 2.2 (two-party computation in the malicious model – informal): *Assume that enhanced trapdoor permutations exist. Then, for any two-party ideal functionality $\mathcal{F}$, there exists a protocol Π that UC 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 also exist, then there exists a protocol Π that UC realizes $\mathcal{F}$ in the $\mathcal{F}_{\text{ZK}}$-hybrid model in the presence of* malicious, adaptive *adversaries.*

As with Proposition 2.1, Theorem 2.2 is not stated exactly. It is formally restated in Section 8.2.

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 5, universally composable commitments can be UC realized in the $\mathcal{F}_{\text{CRS}}$-hybrid model, assuming the existence of enhanced trapdoor permutations. Furthermore, [CF01] showed that the $\mathcal{F}_{\text{ZK}}$ functionality can be UC realized given universally composable commitments. Combining these results together, we have that $\mathcal{F}_{\text{ZK}}$ can be UC realized in the $\mathcal{F}_{\text{CRS}}$-hybrid model, assuming the existence of enhanced trapdoor permutations. Using the composition theorem we obtain a similar result to Theorem 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).

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 enhanced 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* [DP92]. 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: enhanced trapdoor permutations, dense cryptosystems and, in the adaptive case, augmented two-party non-committing encryption protocols.

2.2.3 Extensions to multi-party computation

We now describe how the two-party construction of Theorem 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 $\mathcal{F}_{\text{CP}}^{\text{1:M}}$. (In the one-to-many extension, a *single* party commits and proves to *many* receivers/verifiers.) The extension of the protocol that UC realizes two-party $\mathcal{F}_{\text{CP}}$ to a protocol that UC realizes one-to-many $\mathcal{F}_{\text{CP}}^{\text{1:M}}$ 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 5 or from the scheme of [CF01] (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.[5] Nevertheless, a one-to-many zero-knowledge protocol can be constructed based on the universally-composable zero-knowledge protocol of [CF01] and the methodology of [G98] for obtaining a multi-party extension of zero-knowledge. Specifically, [CF01] 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, as in [G98], 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.)

It remains to describe how to realize $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ in the $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$-hybrid model. The basic idea is to generalize the $\mathcal{F}_{\text{CP}}$ protocol. As with zero-knowledge, this is not straightforward because in the protocol for adaptive adversaries, the $\mathcal{F}_{\text{CP}}$ 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) $\mathcal{F}_{\text{CP}}$ 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 $\mathcal{F}_{\text{CP}}^{1:\text{M}}$-hybrid model, with no further assumptions. Denoting the ideal broadcast functionality used by the parties by $\mathcal{F}_{\text{BC}}$, we have the following theorem:

Theorem 2.3 (multi-party computation in the malicious model – informal): *Assume that enhanced trapdoor permutations exist. Then, for any multi-party ideal functionality $\mathcal{F}$, there exists a protocol Π that UC realizes $\mathcal{F}$ in the $(\mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{CRS}})$-hybrid model in the presence of* malicious, static *adversaries, and for any number of corruptions. Furthermore, if augmented two-party non-committing encryption protocols also exist, then there exists a protocol Π that UC realizes $\mathcal{F}$ in the $(\mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{CRS}})$-hybrid model in the presence of* malicious, adaptive *adversaries, and for any number of corruptions.*

As with Proposition 2.1, Theorem 2.3 is not stated exactly. It is formally restated in Section 9.4.

[5]In the case of static adversaries, the non-interactive zero-knowledge protocol of [d$^+$01] suffices. Thus, here too, the prover message can simply be broadcast and one-to-many zero-knowledge is obtained.

3 Preliminaries

Section 3.1 reviews the framework of [C01] and the universal composition theorem. In Section 3.2 we discuss issues that arise regarding universal composition when some amount of joint state between protocols is desired. Finally, Section 3.3 presents the class of functionalities which we will show how to UC 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 3.1 *Two* binary *distribution ensembles X and Y are* indistinguishable *(written $X \stackrel{c}{\approx} Y$) 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*

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

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 UC 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 [C01]. For full details see there.

Protocol syntax. Following [GMR89, G01], 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).

3.1.1 The basic framework

As sketched in Section 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. This is called the *real-life model*. 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 UC 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.[6] 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 all 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 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[7] 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

[6] We remark that the basic model in [C01] 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 [C01].

[7] The environment 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.

execution is the output of the environment. We assume that this output consists of only a single bit.

In summary, the order of activations is as follows. The environment $\mathcal{Z}$ is always activated first. The environment then either activates the adversary $\mathcal{A}$ or some party P_i by writing on an input tape. If the adversary is activated, it may return control to the environment, or it may activate some party P_i by delivery a message to P_i. After P_i is activated, control is always returned to $\mathcal{Z}$. We stress that at any point, only a single party is activated. Furthermore, $\mathcal{Z}$ and $\mathcal{A}$ can only activate one other entity (thus only a single input is written by $\mathcal{Z}$ per activation and likewise $\mathcal{A}$ can deliver only message per activation).

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 the 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 (simulator) $\mathcal{S}$, an environment $\mathcal{Z}$ with input z, and a set of dummy parties $\tilde{P}_1, \ldots, \tilde{P}_n$.

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 completed 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 an input, it writes it on its outgoing communication tape for the ideal functionality $\mathcal{F}$. Furthermore, whenever a dummy party is activated due to delivery of some message (from $\mathcal{F}$), it copies this message to its output. At the conclusion of a dummy party's activation, the environment $\mathcal{Z}$ is activated. The communication by the dummy parties is with the ideal functionality $\mathcal{F}$ only. In principle, these messages sent between the dummy parties and $\mathcal{F}$ are secret and cannot be read by the adversary $\mathcal{S}$. However, these messages are actually comprised of two parts: a *header* and *contents*. The header is public and can be read by $\mathcal{S}$, whereas the contents is private and cannot be read by $\mathcal{S}$. The definition of the functionality states which information is in the "header" and which is in the "contents". Some information must clearly be in the public header; for example, the identity of the party to whom the functionality wishes to send output must be public so that it can be delivered. Beyond that, the functionality definition should specify what is in the "header" and what is in the "contents". However, for all the functionalities considered in this paper, there is a fixed format for the header. Specifically, the header contains the type of action being taken, the session identifier and the participating parties. For example, a commit message is of the following format: $(\mathsf{commit}, sid, P_i, P_j, b)$, where "commit" states that the party is committing to a new value, sid is the session identifier, P_i is the committing party, P_j is the receiving party and b is the value being committed to. In this case, the header consists of $(\mathsf{commit}, sid, P_i, P_j)$ and the contents consists of (b) only. We adopt this format by convention for all (but one of) the functionalities defined in this paper. We therefore omit specific reference to the header and contents in our definitions of functionalities (and explicitly describe the difference for the one functionality which does not have this format).

When the ideal functionality $\mathcal{F}$ 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 environment $\mathcal{Z}$ is activated next.

When the adversary $\mathcal{S}$ is activated, it may read its own input tape and in addition it can read the *public headers* of the messages on the outgoing communication tape of $\mathcal{F}$. In contrast, $\mathcal{S}$ cannot read the *private contents* of these messages (unless the recipient of the message is $\mathcal{S}$ or a corrupted party[8]). Likewise, $\mathcal{S}$ can read the public headers of the messages intended for $\mathcal{F}$ that appear on the outgoing communication tapes of the dummy parties. Then, $\mathcal{S}$ can execute one of the following four actions: It may either write a message from itself on $\mathcal{F}$'s incoming communication tape[9], deliver a message to $\mathcal{F}$ from some party P_i by copying it from P_i's outgoing communication tape to $\mathcal{Z}$'s incoming communication tape, deliver a message from $\mathcal{F}$ to P_i by copying the appropriate message from $\mathcal{Z}$'s outgoing communication tape to P_i's incoming communication tape, 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).[10] 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 P_i or to the functionality $\mathcal{F}$ in an activation, then this entity (i.e., P_i or $\mathcal{F}$) is activated once the activation of the adversary $\mathcal{S}$ is complete. Otherwise the environment $\mathcal{Z}$ 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 $\mathcal{Z}$.

In summary, the order of activations in the ideal model is as follows. As in the real model, the environment $\mathcal{Z}$ is always activated first, and then activates either the adversary $\mathcal{S}$ or some dummy party P_i by writing an input. If the adversary $\mathcal{S}$ is activated, then it either activates a dummy party P_i or the ideal functionality $\mathcal{F}$ by delivering the entity a message, or it returns control to the environment. After the activation of a dummy party or the functionality, the environment is always activated next.

Let $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k,z,\overline{r})$ 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 $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k,z)$ denote the random variable describing $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k,z,\overline{r})$ when $\overline{r}$ is uniformly chosen. Let $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ denote the ensemble $\{\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k,z)\}_{k\in\mathbf{N},z\in\{0,1\}^*}$.

[8]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.

[9]Many natural ideal functionalities indeed send messages to the adversary $\mathcal{S}$ (see the commitments and zero-knowledge functionalities of Sections 5 and 6 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 $\mathcal{F}$.

[10]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 3.3.

Remark. The above definition of the ideal model slightly differs from that of [C01]. Specifically, in [C01] messages between the dummy parties and ideal functionality are delivered *immediately.* In contrast, in our presentation, this message delivery is carried out by the adversary. Thus, in both the real and ideal models, *all* message delivery is the responsibility of the adversary alone. We note that our results can also be stated in the model of "immediate delivery" as defined in [C01]. However, in such a case, the functionality should always ask the adversary whether or not to accept an input and send an output. This has the same effect as when the adversary is responsible for the delivery of these messages. Therefore, the final result is the same.

UC realizing an ideal functionality. We say that a protocol π UC 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 π 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 π 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 processes. Here it is important that $\mathcal{Z}$ can provide the process in question with *adaptively chosen* inputs throughout the computation.) We have:

Definition 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* UC realizes *$\mathcal{F}$ if for any adversary $\mathcal{A}$ there exists an ideal-process adversary $\mathcal{S}$ such that for any environment $\mathcal{Z}$,*

$$\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} \stackrel{c}{\approx} \text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}. \tag{1}$$

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, UC 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 UC realizing any ideal functionality. All our protocols are in fact clearly non-trivial; therefore, we ignore this issue from here on.

Relaxations of Definition 3.2. We recall two standard relaxations of the definition:

- *Static* (*non-adaptive*) *adversaries.* Definition 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 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.[11] 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 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 UC 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. (1) holds for any environment $\mathcal{Z}$.

3.1.2 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, when the adversary delivers a message from a party to a copy of $\mathcal{F}$ with a particular SID, that copy of $\mathcal{F}$ is the next entity to be activated. (If no such copy of $\mathcal{F}$ exists then a new copy of $\mathcal{F}$ is created and activated to receive the message.) Furthermore, although the adversary in the hybrid model is responsible for delivering the messages between the copies of $\mathcal{F}$ and 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 UC 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 3.3 below.

Let $\text{EXEC}^{\mathcal{F}}_{\pi,\mathcal{A},\mathcal{Z}}(k, z)$ 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 and adversary $\mathcal{A}$, analogously to

[11] 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.

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 $\text{EXEC}^{\mathcal{F}}_{\pi,\mathcal{A},Z}$ denote the distribution ensemble $\{\text{EXEC}^{\mathcal{F}}_{\pi,\mathcal{A},Z}\}_{k\in\mathbf{N},z\in\{0,1\}^*}$.

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 UC 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 [C01] 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 UC 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 UC realizes some functionality $\mathcal{I}$ in the $\mathcal{F}$-hybrid model, and ρ UC realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model, then π^ρ UC realizes $\mathcal{I}$ in the $\mathcal{G}$-hybrid model. (Here one has to define what it means to UC realize functionality $\mathcal{I}$ in the $\mathcal{F}$-hybrid model. This is done in the natural way.) That is:

Theorem 3.3 ([C01]) *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 UC 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:*

$$\text{EXEC}^{\mathcal{G}}_{\pi^\rho,\mathcal{A},\mathcal{Z}} \stackrel{c}{\approx} \text{EXEC}^{\mathcal{F}}_{\pi,\mathcal{S},\mathcal{Z}}.$$

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

Consider the case that $\mathcal{G}$ is the empty functionality, and so the $\mathcal{G}$-hybrid model is actually the *real model*. Then, Theorem 3.3 states that ρ remains secure when run concurrently with *any* protocol π. We note that although π technically seems to be a "calling protocol", it can also represent arbitrary network activity. Thus, we obtain that ρ remains secure when run concurrently in an arbitrary network.

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 [LLR02] for more discussion on this point.

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 UC 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 UC 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 of 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 [CR02] 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 $\hat{\rho}$ be a protocol that UC realizes $\hat{\mathcal{F}}$. Then construct the composed protocol $\pi^{[\hat{\rho}]}$ by replacing *all copies* of $\mathcal{F}$ in π by a *single copy* of $\hat{\rho}$. (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 $\pi^{[\hat{\rho}]}$, 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 $\hat{\rho}$ that UC realizes $\hat{\mathcal{F}}$. Notice that π utilizes calls to $\mathcal{F}$, and not to $\hat{\mathcal{F}}$. Nevertheless, the JUC operation shows how to compose these together. The result is a composed protocol denoted $\pi^{[\hat{\rho}]}$ 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 $\pi^{[\hat{\rho}]}$ behaves like π with the following exceptions:[12]

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

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

3. Whenever protocol $\hat{\rho}$ instructs P_i to send a message m to some party P_j, P_i writes the message $(\hat{\rho}, sid, P_j, m)$ on its outgoing communication tape.

4. Whenever activated due to delivery of a message $(\hat{\rho}, sid, P_i, m)$ from P_j, P_i activates $\hat{\rho}$ with incoming message (sid, P_j, m).

5. Whenever (the single copy of) $\hat{\rho}$ generates an output value (sid_0, sid, v), proceed just as π proceeds with incoming message (sid, v) from $\mathcal{F}_{(sid)}$.

In short, whenever π makes an ideal call to $\mathcal{F}$ with a given identifier sid, protocol $\hat{\rho}$ is called with identifier sid_0 and *sub-session* identifier sid. This is consistent with the fact that $\hat{\rho}$ is a multi-session protocol. Conversely, whenever $\hat{\rho}$ returns a value associated with identifiers (sid_0, sid), this value is returned to π with (the single) identifier sid. This is consistent with the fact that π "thinks" that it is interacting with a single-session functionality $\mathcal{F}$.

Theorem statement. The JUC theorem asserts that if $\hat{\rho}$ UC 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 3.4 (universal composition with joint state [CR02]): *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 UC 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*

$$\text{EXEC}^{\mathcal{F}}_{\pi,\mathcal{S},\mathcal{Z}} \stackrel{\text{c}}{\approx} \text{EXEC}^{\mathcal{G}}_{\pi^{[\rho]},\mathcal{A},\mathcal{Z}}.$$

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

[12] For simplicity, we assume that $\hat{\rho}$ UC 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 $\hat{\rho}$ that UC realize $\hat{\mathcal{F}}$ in the $\mathcal{G}$-hybrid model for some ideal functionality $\mathcal{G}$.

Discussion. Jumping ahead, we sketch our use of the JUC theorem. Recall the commitment functionality, $\mathcal{F}_{\text{COM}}$, formalized in [CF01]. This functionality captures the process of commitment and decommitment to a single value, performed by two parties. In addition, [CF01] show how to realize $\hat{\mathcal{F}}_{\text{COM}}$ in the CRS model, using a single copy of the CRS for all commitments. (In [CF01] the multi-session extension functionality $\hat{\mathcal{F}}_{\text{COM}}$ is called $\mathcal{F}_{\text{MCOM}}$.) An alternative protocol that UC 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 $\hat{\rho}$ that UC realized $\hat{\mathcal{F}}_{\text{COM}}$ using a single copy of the reference string, to obtain a protocol $\pi^{[\hat{\rho}]}$ 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 6.)

3.3 Well-Formed Functionalities

In this section, we define the set of functionalities for which our feasibility results apply. Clearly, we would like to be able to state a theorem saying that *any* ideal functionality can be UC 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 UC 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.[13] (We stress that the fact that p and q are revealed in the real model does not cause any real security concern. This is because when all the 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 UC 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 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 3.4) holds even with respect to the modified definition of multi-session extensions.

4 Two-Party Secure Computation for Semi-Honest Adversaries

This section presents general constructions for UC realizing any two-party ideal functionality in the presence of semi-honest adversaries. The high-level construction is basically that of Goldreich *et. al.* (GMW) [GMW87, G98]. However, there are two differences. First, [GMW87] consider static adversaries whereas we consider adaptive adversaries. This actually only makes a difference in the oblivious transfer protocol; the rest of the protocol for circuit evaluation remains unchanged. We note that although the protocol constructions are very similar, our proof of security is significantly different. This is due to the fact that we deal with adaptive adversaries and in addition show universal composability (in contrast with static adversaries and the standard, stand-alone definitions of security). The second difference between the GMW construction and ours is that while GMW considered function evaluation, we consider more general reactive functionalities. In this case, parties may receive new inputs during the protocol execution and 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 despite the additional generality, this makes only a small difference to the construction.

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

[13] 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.

4.1 Universally Composable Oblivious Transfer

Oblivious transfer [R81, EGL85] 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 $\mathcal{F}_{\mathrm{OT}}^{\ell}$, appears in Figure 1. (Using standard terminology, $\mathcal{F}_{\mathrm{OT}}^{\ell}$ captures 1-out-of-ℓ OT.)

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

$\mathcal{F}_{\mathrm{OT}}^{\ell}$ 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 1: The oblivious transfer functionality, $\mathcal{F}_{\mathrm{OT}}^{\ell}$

Section 4.1.1 presents a protocol that UC realizes $\mathcal{F}_{\mathrm{OT}}$ for static adversaries. Section 4.1.2 presents our protocol for UC realizing $\mathcal{F}_{\mathrm{OT}}$ for adaptive adversaries.

4.1.1 Static UC Oblivious Transfer

The oblivious transfer protocol of [GMW87, G98], denoted SOT (for Static Oblivious Transfer) is presented in Figure 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[14] 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 2: The static, semi-honest oblivious transfer protocol

Claim 4.1 *Assuming that f is an enhanced trapdoor permutation, Protocol* SOT *UC realizes $\mathcal{F}_{\mathrm{OT}}^{\ell}$ 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 $\mathcal{F}_{\mathrm{OT}}^{\ell}$ 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 $\mathcal{F}_{\mathrm{OT}}^{\ell}$. Recall that $\mathcal{S}$ interacts with the ideal functionality $\mathcal{F}_{\mathrm{OT}}^{\ell}$ 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 $\mathcal{F}_{\mathrm{OT}}^{\ell}$ and delivers the output from $\mathcal{F}_{\mathrm{OT}}^{\ell}$ 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 $\mathcal{F}_{\mathrm{OT}}^{\ell}$ 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) expects to receive from T in a real execution. That is, $\mathcal{S}$ chooses a random trapdoor permutation f over $\{0,1\}^k$ and its inverse f^{-1}, and sends (sid, f) 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 $\mathcal{F}_{\mathrm{OT}}^{\ell}$ 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 $\mathcal{F}_{\mathrm{OT}}^{\ell}$.)

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 $\mathcal{F}_{\mathrm{OT}}$. $\mathcal{S}$ 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.

[14] In actuality, it suffices to use any enhanced trapdoor permutation (see Footnote 2) and not necessarily one whose domain equals $\{0,1\}^k$. For simplicity, we present the protocol for this special case.

We demonstrate that $\text{IDEAL}_{\mathcal{F}_{\text{OT}},\mathcal{S},\mathcal{Z}} \stackrel{c}{\approx} \text{REAL}_{\text{SOT},\mathcal{A},\mathcal{Z}}$. 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 i^{th} 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, in the ideal simulation, 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.

4.1.2 Adaptive UC Oblivious Transfer

Due to the above-described problem, we use a different protocol for UC realizing $\mathcal{F}_{\text{OT}}^{\ell}$ for the case of adaptive, semi-honest adversaries. A main ingredient in this protocol are non-committing encryptions as defined in [CFGN96] and constructed in [CFGN96, B97, DN00]. 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.[15]

Augmented two-party non-committing encryption protocols exist under either one of the RSA or the DDH assumptions. The requirements are also fulfilled in any case that public keys are uniformly distributed over some public domain and secret keys are defined by the random coins input to G. See more details in [CFGN96, DN00].

The protocol for UC realizing $\mathcal{F}^{\ell}_{\text{OT}}$, denoted AOT (for Adaptive Oblivious Transfer) is presented in Figure 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 (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 (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 3: The adaptive, semi-honest oblivious transfer protocol

[15] 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 S 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.

Claim 4.2 *Assume that $(G, \hat{G}, E, D, S)$ is an augmented non-committing encryption scheme. Then, Protocol* AOT *UC realizes $\mathcal{F}_{\rm OT}^{\ell}$ in the presence of semi-honest, adaptive adversaries.*

Proof: The main difference between this proof and the proof of Claim 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 $\mathcal{F}_{\rm OT}^{\ell}$ such that no environment $\mathcal{Z}$ can tell with non-negligible advantage whether it is interacting with $\mathcal{A}$ and the above protocol or with $\mathcal{S}$ in the ideal process for $\mathcal{F}_{\rm OT}^{\ell}$. In describing the simulation here, it is helpful to distinguish between the ideal (dummy) parties, denoted $\tilde{T}$ and $\tilde{R}$, and the ITMs representing the real model parties in the internal interaction, denoted T and R. $\mathcal{S}$ works as follows: $\mathcal{S}$ 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 $\tilde{R}$ is not corrupted:* In this case, $\mathcal{S}$ needs to simulate the real 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 R's message to be $(sid, e_1, \ldots, e_\ell)$ and simulates R writing it on its outgoing communication tape for T.
2. *Simulation when the receiver $\tilde{R}$ is corrupted:* In this case, $\mathcal{S}$ holds the input $(\mathsf{receiver}, sid, i)$ of the ideal receiver $\tilde{R}$ and constructs a virtual real-model receiver R as follows. The contents of the input tape of R is set to $(\mathsf{receiver}, sid, i)$. In order to set the contents of 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 $j \neq i$, $\mathcal{S}$ 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. Furthermore, $\mathcal{S}$ uses the invertible sampling algorithm in order to find r_i so that $G(r_i) = (e_i, d_{x_i}^i)$. (Recall that x_i is the ideal receiver $\tilde{R}$'s output.) Then, $\mathcal{S}$ sets the contents of R's random tape to equal $r_1, \ldots, r_\ell$.

   $\mathcal{S}$ passes the simulated internal state of R (including the contents of its input and random tapes) to $\mathcal{A}$ and waits for $\mathcal{A}$ to activate R in the simulation. When this occurs, $\mathcal{S}$ internally obtains the message $(sid, e_1, \ldots, e_\ell)$ that R writes on its outgoing message tape. Then, when $\mathcal{A}$ delivers this message to T in the internal simulation, $\mathcal{S}$ externally delivers the $(\mathsf{receiver}, sid, i)$ message from $\tilde{R}$ to $\mathcal{F}_{\rm OT}^{\ell}$. (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 is the encryption key output by $G(r_i)$, 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 $\tilde{T}$ is not corrupted:* After $\mathcal{A}$ delivers the message from R to T in the internal simulation, $\mathcal{S}$ simulates the real-model T writing $(sid, c_1, \ldots, c_\ell)$ on its outgoing communication tape, where the c_i's were generated from $S(1^k)$ when simulating the receiver message above. When $\mathcal{A}$ delivers this message from T to R in the internal simulation, then $\mathcal{S}$ delivers the output from $\mathcal{F}^\ell_{\mathrm{OT}}$ to $\tilde{R}$ in the ideal process. This is the same whether or not $\tilde{R}$ is corrupted.

2. *Simulation when the sender $\tilde{T}$ is corrupted:* The simulator $\mathcal{S}$ holds the ideal $\tilde{T}$'s input $(\mathsf{sender}, sid, x_1, \ldots, x_\ell)$ and constructs a virtual real-model sender 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 simulated internal state of T (consisting of the contents of its input and random tapes). When $\mathcal{A}$ activates T and delivers the outgoing message from T to R, then simulator $\mathcal{S}$ externally delivers the message $(\mathsf{sender}, sid, x_1, \ldots, x_\ell)$ from $\tilde{T}$ to $\mathcal{F}^\ell_{\mathrm{OT}}$.

Dealing with "corrupt" commands: We assume that any corruption of a party occurs after the adversary has delivered the party's 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 R, then it corrupts the ideal model $\tilde{R}$ and obtains its input i and its output x_i. Given i and x_i, simulator $\mathcal{S}$ passes $\mathcal{A}$ the decryption-key $d^i_{x_i}$ (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 $G(r_i) = (e_i, d^i_{x_i})$. Notice that these two invertible sampling algorithms are different. $\mathcal{S}$ supplies $\mathcal{A}$ with the random tape $r_1, \ldots, r_\ell$ for R.

If $\mathcal{S}$ receives a command from $\mathcal{A}$ to corrupt real-model T, then it first corrupts the ideal-model $\tilde{T}$ 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 T's random tape.

As argued in the proof of Claim 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.) ■

4.2 The General Construction

We are now ready to show how to UC realize any (adaptively) well-formed two-party functionality in the $\mathcal{F}_{\mathrm{OT}}$-hybrid model, in the semi-honest case. (Adaptively well-formed functionalities are defined in Section 3.3. Two-party functionalities are functionalities that interact with the adversary, plus at most two parties.) The construction is essentially that of [GMW87, G98], 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 2.1:

Proposition 4.3 (Proposition 2.1 – formally restated): *Assume that enhanced trapdoor permutations exist. Then, for any two-party well-formed ideal functionality $\mathcal{F}$, there exists a non-trivial protocol that UC 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 UC 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 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 UC realizes $\mathcal{F}$ is to represent (the core 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 [GMW87, G98], 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.[16] 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 UC 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

[16] 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.

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^m.)

   - *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 $\mathcal{F}_{\mathrm{OT}}^4$ 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, 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	$o_1 = \gamma_1 + (\alpha_1 + 0) \cdot (\beta_1 + 0)$
(0,1)	2	$o_2 = \gamma_1 + (\alpha_1 + 0) \cdot (\beta_1 + 1)$
(1,0)	3	$o_3 = \gamma_1 + (\alpha_1 + 1) \cdot (\beta_1 + 0)$
(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 $(\textsf{receiver}, sid \circ j, i)$ 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, and writes the result on its output tape. (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 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 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 $\langle sid \circ \ell \circ \textsf{input}\rangle$. 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 UC realizes any adaptively well-formed functionality. (We stress that for the case of static adversaries, $\Pi_{\mathcal{F}}$ UC 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 also receive notification of the parties that are corrupted. The protocol does not (and cannot) deal with such messages and therefore does not UC realize functionalities that are not well-formed.

Claim 4.4 *Let $\mathcal{F}$ be a two-party adaptively well-formed functionality. Then, protocol $\Pi_{\mathcal{F}}$ UC realizes $\mathcal{F}$ in the $\mathcal{F}_{\rm OT}$-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 UC 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 [G98, 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 $\Pi_{\mathcal{F}}$ UC 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 $\mathcal{F}_{\rm OT}$-*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 $\mathcal{F}_{\rm OT}$-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 by the specified uniform random tape of P_1, 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^{th} gate is the $(sid \circ j)$ message from the corresponding copy of $\mathcal{F}_{\text{OT}}$. Thus, $\mathcal{S}$ simulates this stage by merely simulating for $\mathcal{A}$ an $(sid \circ j)$ message sent from $\mathcal{F}_{\text{OT}}$ 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 $\mathcal{F}_{\text{OT}}$ 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 $\mathcal{F}_{\text{OT}}$. Therefore, $\mathcal{S}$ merely chooses $\gamma_2 \in_R \{0,1\}$ and simulates P_2 receiving γ_2 from $\mathcal{F}_{\text{OT}}$.
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 here 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 and output value x. $\mathcal{S}$ 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^m 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 of 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^m.) 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 $\gamma_1 = \gamma_2 \oplus \gamma$ (notice that γ_1 and P_1's input values to the gate fully define the table). $\mathcal{S}$ 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 the uncorrupted 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 $\mathcal{F}_{\text{OT}}$-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 a *real* (or, more precisely, hybrid) interaction with adversary $\mathcal{A}$ and parties running $\Pi_{\mathcal{F}}$ in the $\mathcal{F}_{\text{OT}}$-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 $(\zeta_i^{\text{R}}, \alpha_i^{\text{R}})$ and $(\zeta_i^{\text{I}}, \alpha_i^{\text{I}})$ are identically distributed. More precisely, Let $i > 0$. Then, for any values a_1, a_2, b_1, b_2 we show:

$$\Pr\left[(\zeta_{i+1}^{\text{R}}, \alpha_{i+1}^{\text{R}}) = (b_1, b_2) \,\middle|\, (\zeta_i^{\text{R}}, \alpha_i^{\text{R}}) = (a_1, a_2)\right] = \Pr\left[(\zeta_{i+1}^{\text{I}}, \alpha_{i+1}^{\text{I}}) = (b_1, b_2) \,\middle|\, (\zeta_i^{\text{I}}, \alpha_i^{\text{I}}) = (a_1, a_2)\right] \quad (2)$$

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 real 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. (2), 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 $\zeta_i^{\text{R}} = \zeta_i^{\text{I}}$.) Next, $\mathcal{Z}$ either activates some uncorrupted party with some input v, or activates the adversary with input v. We treat these cases separately:

$\mathcal{Z}$ activates an uncorrupted party with some input value v. 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 v. 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 has written a message on its outgoing communication tape for $\mathcal{F}$) generates the message that $\mathcal{A}$ would expect to see; recall that this message is just a uniformly distributed string.

$\mathcal{Z}$ activates the adversary or a corrupted party with some input value v. Recall that in the interaction with $\Pi_{\mathcal{F}}$ adversary $\mathcal{A}$ is now activated, reads v, and in addition has access to the messages sent by the parties and by the various copies of $\mathcal{F}_{\text{OT}}$ 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 v 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 UC composition theorem, Proposition 4.3 follows from Claims 4.1, 4.2, and 4.4.

5 Universally Composable Commitments

We describe our new universally composable non-interactive commitment scheme that is secure against adaptive adversaries. Our construction is in the common reference string model, and assumes only the existence of enhanced trapdoor permutations. (If the common reference string must come from the uniform distribution, then we actually require enhanced trapdoor permutations with dense public descriptions [DP92].) UC commitment schemes are protocols that UC realize the multi-session ideal commitment functionality $\mathcal{F}_{\text{MCOM}}$ that is presented in Figure 4. Note that $\mathcal{F}_{\text{MCOM}}$ is in fact a re-formulation of $\hat{\mathcal{F}}_{\text{COM}}$, the multi-session extension of the single-session ideal commitment functionality, $\mathcal{F}_{\text{COM}}$, presented in [CF01].

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: The ideal commitment functionality

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

- *Polynomial equivocality:*[17] the simulator (i.e., the adversary in the ideal process) should be able to produce polynomially 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).

[17]Commitments with this property have mistakenly been called "equivocable"; we fix this error and call them *equivocal*.

- *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 equivocal commitments.

We remark that in the equivocal commitment protocols of [DIO98, DKOS01] each copy of the reference string can be used for only a single commitment. Furthermore, they are not extractable. In contrast, [CF01] 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 [FS89], which is at the heart of the construction. Then we show how to transform this scheme into one that is universally composable.

Underlying standard commitment. Our UC commitment scheme uses 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 [FS89], which is based on the zero-knowledge proof for Hamiltonicity of Blum [B86]. (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 (Cook-Levin) NP-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 scheme of [FS89] that we use here is equivocality (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 0 and 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. In order to explain elements of our construction that are important in obtaining security against adaptive adversaries, we begin by describing a general problem that can occur in such a setting. In the adaptive setting, the adversary can corrupt parties at any point in the computation. Specifically, for the case of commitments, this means that the adversary can corrupt a party *after* the commit stage has been completed. In such a case, the simulator must generate some "simulated commitment" string c before the committing party is corrupted, and therefore without knowing the value b that is being committed to. Then, after the corruption takes place and the committed value b becomes known to the simulator and adversary, it must be possible to "explain" the string c as a valid commitment to b. This is because in a real execution, c is indeed a commitment to b and $\mathcal{A}$ will see this upon corrupting the committer. However, in the ideal process, c is generated by $\mathcal{S}$ who knows nothing about the value b. A valid "explanation" of the string c is a series of coins r_b such that upon input b and random coins r_b, the *honest committer* would output the commitment string c. Consider the [FS89] commitment scheme described above, and assume that the ideal-model simulator $\mathcal{S}$ knows a Hamiltonian cycle in G. Then, as we have described, it is possible for $\mathcal{S}$ to generate a commitment string c that can be decommitted to both 0 and 1. However, it is *not* possible for $\mathcal{S}$ to later provide two sets of coins r_0 and r_1 so that c is the result of an honest commitment to b with coins r_b (for $b = 0$ and $b = 1$). This is because the coins used in all of the *Com* commitments demonstrate whether a permutation of G was committed to or just a simple cycle. This demonstrates that the trapdoor or equivocality property is not enough for obtaining adaptive security.

In order to obtain adaptive security, we modify the way that the *Com* commitments are generated in the [FS89] scheme. Specifically, commitments which are not opened upon decommitment (i.e., commitments outside of the simple cycle in a commitment to 1) are generated by just choosing random strings. Since *Com* has a pseudorandom range and these commitments are never opened, this makes no difference. Below we will show that this actually suffices for obtaining adaptive security. More precisely, the adaptively secure scheme, denoted aHC (for *adaptive Hamiltonian Commitment*), is defined as follows:

- 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 [FS89] 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 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 [FS89]. That is, in [FS89] 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 [FS89], 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 [FS89], 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 equivocality, where the same reference string can be reused polynomially-many times.

Achieving simulation extractability. As discussed above, the commitment scheme aHC has the equivocality 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 equivocality. Simulation-extractability alone could be achieved by including a public-key for an encryption scheme secure against adaptive chosen-ciphertext attacks (CCA2) [DDN00] 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 equivocality of the overall scheme, since such an encryption is binding even to a simulator. In order to regain equivocality, we use encryption schemes with pseudorandom ciphertexts. This is used in the following way. Given any equivocal 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, equivocality 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 enhanced 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 enhanced trapdoor permutations[18] 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 [DDN00]. For the second encryption scheme, denoted E, we use the standard encryption scheme based on trapdoor-permutations and hard-core predicates [GL89], 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 UAHC (for UC Adaptive Hamiltonicity Commitment), is presented in Figure 5.

[18]The fact that enhanced trapdoor permutations are used (and not any trapdoor permutation) is due to the non-interactive zero-knowledge (NIZK) proofs which are used in all known CCA2-secure schemes that are based on general assumptions. The additional "enhanced" feature of trapdoor permutations is used in the construction of these NIZK proofs.

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 $|C_b|$.[19] 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 5: The commitment protocol UAHC

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 5). Then, we have:

Proposition 5.1 *Assuming the existence of enhanced trapdoor permutations, Protocol* UAHC *of Figure* 5 *UC 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 $\mathcal{A}$.

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^{cca} 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):

[19] 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.

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: In the ideal model, when the honest committer P_i receives an input $(\mathsf{commit}, sid, ssid, P_i, P_j, b)$ from the environment, it writes this message on its outgoing communication tape for $\mathcal{F}_{\text{MCOM}}$. Recall that by convention, the $(\mathsf{commit}, sid, sssid, P_i, P_j)$ part of the message (i.e., the header) is public and can be read by $\mathcal{S}$, whereas the actual input value b cannot be read by $\mathcal{S}$ (see Section 3.1 – "the ideal process"). Now, upon seeing that P_i writes a "commit" message for $\mathcal{F}_{\text{MCOM}}$, $\mathcal{S}$ simulates a real party P_i writing the commit message of Protocol UAHC on its outgoing communication tape for P_j. That is, $\mathcal{S}$ 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, $\mathcal{S}$ stores (c, r_0, s_0, r_1, s_1) and simulates P_i writing $c = (sid, ssid, P_i, z, C_0, C_1)$ on its outgoing communication tape for P_j. When $\mathcal{A}$ delivers c from P_i to P_j in the internal simulation, then $\mathcal{S}$ delivers the message from the ideal process P_i's outgoing communication tape to $\mathcal{F}_{\text{MCOM}}$. Furthermore, $\mathcal{S}$ also delivers the $(\mathsf{receipt}, \ldots)$ message from $\mathcal{F}_{\text{MCOM}}$ to P_j. If $\mathcal{A}$ corrupts P_i before delivering c and then *modifies* c before delivering it, then $\mathcal{S}$ proceeds by following the instructions for a corrupted committer. If $\mathcal{A}$ corrupts P_i but does not modify c, then $\mathcal{S}$ carries out the simulation as described here.

Simulating "reveal" activations where the committer is uncorrupted: When an honest P_i receives a $(\mathsf{reveal}, sid, ssid)$ input from $\mathcal{Z}$, it writes this on its outgoing communication tape for $\mathcal{F}_{\text{MCOM}}$ (this entire message is a "header" and is therefore public). $\mathcal{S}$ then delivers this message to $\mathcal{F}_{\text{MCOM}}$ and obtains the message $(\mathsf{reveal}, sid, ssid, P_i, P_j, b)$ from $\mathcal{F}_{\text{MCOM}}$. Given the value b, $\mathcal{S}$ 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 internally simulates for $\mathcal{A}$ the event where P_i writes this message on its outgoing communication tape for P_j. When $\mathcal{A}$ delivers this message from P_i to P_j in the internal interaction, then $\mathcal{S}$ delivers the $(\mathsf{reveal}, sid, ssid, P_i, P_j, b)$ message from $\mathcal{F}_{\text{MCOM}}$ to P_j.

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 b is the commitment value associated with $(sid, ssid)$ in P_i's internal state, and 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, delivers 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^{cca} and C_1^{cca} and then decrypting each of C_0^{cca} and C_1^{cca}. There are three cases here:

1. *Case 1:* For some $b \in \{0,1\}$, C_b^{cca} 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}^{cca} does *not* decrypt to a decommitment to $1-b$. Then, $\mathcal{S}$ sends $(\mathsf{commit}, sid, ssid, P_i, P_j, b)$ to $\mathcal{F}_{\text{MCOM}}$, delivers $\mathcal{F}_{\text{MCOM}}$'s $\mathsf{receipt}$ response to P_j, and stores the commitment string.
2. *Case 2:* Neither C_0^{cca} or C_1^{cca} 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 $\mathcal{F}_{\text{MCOM}}$ and delivers $\mathcal{F}_{\text{MCOM}}$'s $\mathsf{receipt}$ response to P_j. (The commitment string is not stored, since it will never be opened correctly.)
3. *Case 3:* C_0^{cca} decrypts to $(P_i, P_j, sid, ssid, r_0)$ and C_1^{cca} 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, delivers 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 $\mathcal{F}_{\text{MCOM}}$ and delivers the reveal message from $\mathcal{F}_{\text{MCOM}}$ to P_j. Otherwise, $\mathcal{S}$ ignores the message.

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) 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 [BBM00]. 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). This is the case because decryption is only needed for decommitment, and the only ciphertext to be decrypted upon decommitment is C_b (and not C_{1-b}). 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. That is, if $\mathcal{S}$ never outputs $\mathsf{failure}$, then experiments (III) and (IV) are *identical*. This is due to the fact that if $\mathcal{S}$ never outputs $\mathsf{failure}$, then for every commitment c generated by the real-model adversary $\mathcal{A}$, there is at most one possible decommitment. There is therefore no need for $\mathcal{S}$ to carry out these checks. We 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} (i.e., in (IV) we have that C_{1-b} contains the actual random coins r_{1-b}, whereas in (V) this is replaced by $0^{|r_{1-b}|}$). 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 plaintexts $(P_i, P_j, sid, ssid, r_{1-b})$ and $(P_i, P_j, sid, ssid, 0^{|r_{1-b}|})$. When the LR-oracle responds with a ciphertext C^{cca}, the attacker sets $C_{1-b} \leftarrow E(C^{cca})$.
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^{cca} and C_1^{cca}. There are two cases:

- *Case 1* – a ciphertext C_b^{cca} came from a commitment previously generated for an honest party by $\mathcal{S}$: If this generated commitment was not from P_i to P_j, then C_b^{cca} cannot constitute a valid decommitment because the encryption does not contain the pair (P_i, P_j) in this order. Likewise, if the previous commitment was from P_i to P_j but the sub-session identifiers are different, then it still cannot be a valid decommitment. (Recall that P_j will ignore a second commitment from P_i with the same identifiers.) In the above cases, 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.)

  One other possible case is that the previous commitment was from P_i to P_j and the sub-session identifiers are correct. This can only happen if P_i was corrupted after the commitment message containing C_b^{cca} was written by $\mathcal{S}$ on P_i's outgoing communication tape, but before $\mathcal{A}$ delivered it. However, in this case, the attacker knows the decryption of the ciphertext without querying the decryption oracle, and therefore uses this value in the simulation.
- *Case 2* – a ciphertext C_b^{cca} 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.

We note that the above simulation can be carried out without knowing the decryption key for E_{cca}. This is because the attacker knows the value b that an honest party is committing to before it needs to generate the simulated commitment value. Therefore, it never needs to decrypt C_{1-b}.

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^{cca} 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^{cca} 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^{cca} 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. That is, in these experiments C_{1-b} does not contain the random coins r_{1-b} which would constitute a decommitment to $\mathsf{aHC}(1–b)$. Now, for commitments to 0 (i.e., when $b = 0$), experiments (V) and (IV) are identical. However, for commitments to 1 (i.e., when $b = 1$), they are different. Nevertheless, it is hard to distinguish the case that $z = \mathsf{aHC}(0)$ from the case that $z = \mathsf{aHC}(1)$, even given the random coins r_1 that constitute a decommitment of z to 1.

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. ■

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

Figure 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.)[20]

[20]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 UC realize $\hat{\mathcal{F}}_{\text{ZK}}$ using a single short instance of the common string. We then use the JUC theorem (Theorem 3.4) to compose protocols in the $\mathcal{F}_{\text{ZK}}$-hybrid model with protocols that UC 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 3.2). Nonetheless, for the sake of clarity we explicitly present functionality $\hat{\mathcal{F}}_{\text{ZK}}$ in Figure 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 $P_1,\dots,P_n$ 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 7: The multi-session zero-knowledge functionality

In the case of static adversaries, there exists a protocol that UC realizes $\hat{\mathcal{F}}_{\text{ZK}}$ for any NP relation, in the common reference string (CRS) model [d+01]. The protocol of [d+01] assumes the existence of enhanced trapdoor 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, [CF01] show a three-round protocol that UC 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 5 below). The protocol uses a single copy of $\mathcal{F}_{\text{MCOM}}$.[21] We conclude that also in the adaptive case $\hat{\mathcal{F}}_{\text{ZK}}$ can be UC realized in the $\mathcal{F}_{\text{CRS}}$-hybrid model, assuming the existence of enhanced trapdoor permutations. (This is obtained by plugging our construction of $\mathcal{F}_{\text{MCOM}}^{1:\text{M}}$ into the $\hat{\mathcal{F}}_{\text{ZK}}$ protocol of [CF01].)

[21] Actually, the zero-knowledge functionality in [CF01] 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 its input x and w is such that $(x,w) \in R$, their protocol UC realizes $\hat{\mathcal{F}}_{\text{ZK}}$.

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 UC realizing it. As discussed in Section 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 than the above description, 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 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 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.

7.1 UC Realizing $\mathcal{F}_{\text{CP}}$ for Static Adversaries

We present protocols for UC 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 Com be a perfectly binding commitment scheme, and denote by $Com(w;r)$ 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 [G01]. (Alternatively, we could use the Naor scheme [N91] 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 7.2.) Loosely speaking, the protocol begins by the committer C sending $c = Com(w;r)$ to V, and then proving knowledge of the pair (w,r). In our terminology, this consists of C sending $(\mathsf{ZK\text{-}prover}, sid_C, c = Com(w;r), (w,r))$ to $\mathcal{F}_{\text{ZK}}$, which is parameterized by the following relation R_C:

$$R_C = \{(c,(w,r)) \mid c = Com(w;r)\} \tag{3}$$

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:

$$R_P \stackrel{\text{def}}{=} \{((x,\overline{c}),(\overline{w},\overline{r})) \mid \forall i,\ c_i = Com(w_i;r_i) \ \&\ R(x,\overline{w}) = 1\} \tag{4}$$

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 $R \in \text{NP}$, then so too is R_P.

We denote by $\mathcal{F}_{\text{ZK}}^{\text{C}}$ and $\mathcal{F}_{\text{ZK}}^{\text{P}}$ the copies of $\mathcal{F}_{\text{ZK}}$ from the commit phase and prove phase respectively (i.e., $\mathcal{F}_{\text{ZK}}^{\text{C}}$ is parameterized by R_C and $\mathcal{F}_{\text{ZK}}^{\text{P}}$ 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 Com, is presented in Figure 9.

Proposition 7.1 *Assuming that Com is a secure* (*perfectly binding*) *commitment scheme,*[22] *Protocol* SCP *of Figure* 9 *UC 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

[22]When we refer to "secure perfectly binding commitments" here, we mean secure according to standard definitions (see [G01, Section 4.4.1] for a formal definition).

Protocol SCP

- **Auxiliary Input:** A security parameter k.
- **Commit phase:**
  1. On input (commit, sid, w), where $w \in \{0,1\}^k$, C chooses a random string r of length sufficient for committing to w in scheme Com, and sends ($\mathsf{ZK\text{-}prover}, sid_C, Com(w;r), (w,r)$) to $\mathcal{F}_{\text{ZK}}^{\text{C}}$, where $\mathcal{F}_{\text{ZK}}^{\text{C}}$ is parameterized by the relation R_C defined in Eq. (3). In addition, C stores in a vector $\overline{w}$ the list of all the values w that it has sent to $\mathcal{F}_{\text{ZK}}^{\text{C}}$, 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 $\mathcal{F}_{\text{ZK}}^{\text{C}}$, the receiver 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 $\mathcal{F}_{\text{ZK}}^{\text{P}}$, where $\overline{w},\overline{r},\overline{c}$ are the above-define vectors and $\mathcal{F}_{\text{ZK}}^{\text{P}}$ is parameterized by the relation R_P defined in Eq. (4).
  2. When receiving ($\mathsf{ZK\text{-}proof}, sid_P, (x,\overline{c})$) from $\mathcal{F}_{\text{ZK}}^{\text{P}}$, 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 9: A protocol for realizing $\mathcal{F}_{\text{CP}}$ for static adversaries

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 $\mathcal{A}$" 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 $\mathcal{F}_{\text{CP}}$. We use the term *external communication* to refer to $\mathcal{S}$'s communication with $\mathcal{Z}$ and $\mathcal{F}_{\text{CP}}$. 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 $\mathcal{F}_{\text{CP}}$) 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 that the committer is uncorrupted: In this case, $\mathcal{A}$ expects to see the messages sent by $\mathcal{F}_{\text{ZK}}^{\text{C}}$ and $\mathcal{F}_{\text{ZK}}^{\text{P}}$ to V. (Notice that the only messages sent in the protocol are to and from $\mathcal{F}_{\text{ZK}}^{\text{C}}$ and $\mathcal{F}_{\text{ZK}}^{\text{P}}$; 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 $\mathcal{F}_{\text{CP}}$. It constructs the appropriate $\mathcal{F}_{\text{ZK}}^{\text{C}}$ messages given the $\mathsf{receipt}$ messages from $\mathcal{F}_{\text{CP}}$, and the appropriate $\mathcal{F}_{\text{ZK}}^{\text{P}}$ messages given the $\mathsf{CP\text{-}proof}$ messages from $\mathcal{F}_{\text{CP}}$. This is done as follows:

- Whenever $\mathcal{S}$ receives a message ($\mathsf{receipt}, sid$) from $\mathcal{F}_{\text{CP}}$ where C is uncorrupted, $\mathcal{S}$ computes

$c = Com(0^k; r)$ for a random r and (internally) passes $\mathcal{A}$ the message $(\mathsf{ZK\text{-}proof}, sid_C, c)$, as $\mathcal{A}$ would receive from $\mathcal{F}_{\rm ZK}^{\rm C}$ 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 $\mathcal{F}_{\rm CP}$ 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 $\mathcal{F}_{\rm ZK}^{\rm P}$ 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 that the committer is corrupted: Here, $\mathcal{A}$ controls C and generates the messages that C sends during an execution of Protocol SCP.[23] 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 $\mathcal{F}_{\rm CP}$ (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 $\mathcal{F}_{\rm ZK}$-hybrid model, and any message sent by $\mathcal{A}$ to $\mathcal{F}_{\rm ZK}$ is seen by $\mathcal{S}$ during the simulation. In particular, $\mathcal{S}$ obtains the ZK-proof message sent by $\mathcal{A}$ to $\mathcal{F}_{\rm ZK}^{\rm C}$, 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 $\mathcal{F}_{\rm ZK}^{\rm C}$, simulator $\mathcal{S}$ checks that $c = Com(w; r)$. If this holds, then $\mathcal{S}$ externally sends $(\mathsf{commit}, sid, w)$ to $\mathcal{F}_{\rm CP}$ and internally passes $(\mathsf{ZK\text{-}proof}, sid_C, c)$ to $\mathcal{A}$ (as if coming from $\mathcal{F}_{\rm ZK}^{\rm C}$). 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 $\mathcal{F}_{\rm ZK}^{\rm P}$, 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 $\mathcal{F}_{\rm ZK}^{\rm P}$) and externally sends the prover message $(\mathsf{CP\text{-}prover}, sid, x)$ to $\mathcal{F}_{\rm CP}$. If no, then $\mathcal{S}$ does nothing.

Message delivery: It remains to describe when (if at all) $\mathcal{S}$ delivers the messages between the dummy parties C and V and the functionality $\mathcal{F}_{\rm CP}$. Simulator $\mathcal{S}$ delivers commit and CP-prover messages from C to $\mathcal{F}_{\rm CP}$ when $\mathcal{A}$ delivers the corresponding ZK-prover messages from C to $\mathcal{F}_{\rm ZK}^{\rm C}$ and $\mathcal{F}_{\rm ZK}^{\rm P}$ in the internal simulation. Likewise, $\mathcal{S}$ delivers the receipt and CP-proof messages from $\mathcal{F}_{\rm CP}$ to V in the ideal process, when $\mathcal{A}$ delivers the corresponding ZK-proof messages from $\mathcal{F}_{\rm ZK}^{\rm C}$ and $\mathcal{F}_{\rm ZK}^{\rm P}$ to V in the simulation.

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

$$\text{IDEAL}_{\mathcal{F}_{\rm CP},\mathcal{S},\mathcal{Z}} \stackrel{\rm c}{\approx} \text{EXEC}^{\mathcal{F}_{\rm ZK}}_{\rm SCP,\mathcal{A},\mathcal{Z}} \tag{5}$$

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 $\mathcal{F}_{\rm CP}$. 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

[23] 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 $\mathcal{Z}$.

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 V. Likewise, in a protocol execution, if $((x, \overline{c}), (\overline{w}, \overline{r})) \notin R_P$, then V receives no CP-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 CP-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. (5). 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 $c = Com(0^k; r)$. 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 $c = Com(w; r)$, where w is C's actual input. Intuitively, by the hiding property of the commitment scheme Com, 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 EXEC distributions can be distinguished. Then, we construct a distinguisher D for the commitment scheme Com. 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 $\{1, \ldots, t\}$, 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 $c = Com(0^k; r)$. 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 [N91] (which in turn can use *any* one-way function). In this case, V begins by sending the receiver message of the [N91] 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 UC realized in the $\mathcal{F}_{\text{ZK}}$-hybrid model, assuming the existence of *any* one-way function.

7.2 UC Realizing $\mathcal{F}_{\text{CP}}$ for Adaptive Adversaries

We now present a protocol for UC 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 Com 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.[24] 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 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 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 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 [N91] (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 [N91] 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

[24] 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_w such that $c = Com(w; r_w)$. This is needed for the adaptive corruption of the committing party. See Section 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:

$$R_T \stackrel{\text{def}}{=} \{(s,t) \mid s = f(t)\} \tag{6}$$

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

$$R_C \stackrel{\text{def}}{=} \{((s,c),(w,r)) \mid c = \mathsf{aHC}_s(w;r)\} \tag{7}$$

$$R_P \stackrel{\text{def}}{=} \{((x,s,\overline{c}),(\overline{w},\overline{r})) \mid \forall i, c_i = \mathsf{aHC}_s(w_i;r_i) \ \&\ R(x,\overline{w}) = 1\} \tag{8}$$

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 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 7.2 *Assuming the existence of one-way functions, Protocol* ACP *of Figure* 10 *UC 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 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 the underlying commitment scheme aHC is only computational. Thus, the validity of the simulation will be demonstrated by a reduction to the binding property of aHC.)

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 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 $\mathcal{F}_{\text{ZK}}^{\text{T}}$.

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. (6). 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. (7). 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 $(\mathsf{CP\text{-}prover}, sid, x)$, the committer/prover 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. (8).
  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 same string s 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 10: A protocol for realizing $\mathcal{F}_{\text{CP}}$ for adaptive adversaries

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 an uncorrupted C writes a commit message on its outgoing transcript for $\mathcal{F}_{\text{CP}}$, simulator $\mathcal{S}$ simulates the hybrid-model C writing a $\mathsf{ZK\text{-}prover}$ message on its outgoing communication tape for $\mathcal{F}_{\text{ZK}}^{\text{C}}$ (recall that only the headers in these messages are public). Then, when $\mathcal{A}$ delivers this message to $\mathcal{F}_{\text{ZK}}^{\text{C}}$ in the simulated interaction, $\mathcal{S}$ computes $c = \mathsf{aHC}_s(0^k; r)$ for a random r, and simulates $\mathcal{A}$ receiving the message $(\mathsf{ZK\text{-}proof}, sid_C, (s, c))$ from $\mathcal{F}_{\text{ZK}}^{\text{C}}$. Likewise, $\mathcal{S}$ delivers the commit message from C to $\mathcal{F}_{\text{CP}}$, and receives back $(\mathsf{receipt}, sid)$ from $\mathcal{F}_{\text{CP}}$. Then, when $\mathcal{A}$ delivers the $\mathsf{ZK\text{-}proof}$ from $\mathcal{F}_{\text{ZK}}^{\text{C}}$ to V, simulator $\mathcal{S}$ delivers the $\mathsf{receipt}$ message from $\mathcal{F}_{\text{CP}}$ to V. The "prove phase" is simulated in an analogous way.

   (Recall that by the aHC scheme, the commitment c to 0^k that is generated by $\mathcal{S}$ can be later opened as any string in $\{0,1\}^k$, given the trapdoor information t; see Section 5. This will be needed below.)

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 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 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 $(\mathsf{ZK\text{-}prover}, sid_P, (x, \overline{c}), (\overline{w}', \overline{r}'))$ to send to $\mathcal{F}_{\text{ZK}}^{\text{P}}$, where $\overline{w}' \neq \overline{w}$ and yet for every i, it holds that $c_i = \mathsf{aHC}_s(w_i', r_i')$. 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 $(\mathsf{CP\text{-}proof}, sid, x)$ 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 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 aHC commitment scheme. It suffices to show that $\mathcal{A}$ cannot generate a message $(\mathsf{ZK\text{-}prover}, sid_P, (x, \overline{c}), (\overline{w}', \overline{r}'))$ where $\overline{w}' \neq \overline{w}$ and yet for every i, it holds that $c_i = \mathsf{aHC}_s(w_i', r_i')$. Intuitively, this follows from the binding property of aHC (see Section 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 $c = \mathsf{aHC}_s(w_1; r_1) = \mathsf{aHC}_s(w_2; r_2)$ 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 $c = \mathsf{aHC}_s(0^k; r)$, machine M provides $\mathcal{S}$ with the input w being committed to and then $\mathcal{S}$ provides $\mathcal{A}$ with $c = \mathsf{aHC}_s(w; r)$. 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.[25]

[25]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 unknown).

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* 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. ∎

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 8.1. Then, Section 8.2 ties together all the components of the construction, from Sections 4, 6, 7 and 8.1.

8.1 The Protocol Compiler

As discussed in Section 2.2, the $\mathcal{F}_{\text{CP}}$ functionality is used to construct a protocol compiler that transforms any non-trivial protocol that UC realizes some two-party functionality $\mathcal{F}$ in the presence of semi-honest adversaries (e.g., the protocol of Section 4), into a non-trivial protocol that UC 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 Sections 3.1 and 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 11 below. It uses two copies of $\mathcal{F}_{\text{CP}}$: one for when P_1 is the committer and one for when P_2 is the committer. These copies of $\mathcal{F}_{\text{CP}}$ are denoted $\mathcal{F}^1_{\text{CP}}$ and $\mathcal{F}^2_{\text{CP}}$, 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 $\mathcal{F}_{\text{CP}}$-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

$\mathsf{Comp}(\Pi)$

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 $\mathcal{F}_{\rm CP}^1$. (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 $r_1 \stackrel{\rm def}{=} r_1^1 \oplus r_1^2$ (r_1 serves 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 $\mathcal{F}_{\rm CP}^2$ (this occurs after P_2 sends a commit message $(\mathsf{commit}, sid_2, r_2^2)$ to $\mathcal{F}_{\rm CP}^2$). 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 $\mathcal{F}_{\rm CP}^1$ 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 $\mathcal{F}_{\rm CP}^1$.
   (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 $\mathcal{F}_{\rm CP}^1$ where the relation R_Π for $\mathcal{F}_{\rm CP}^1$ 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 $\mathcal{F}_{\rm CP}^1$, 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 $\mathcal{F}_{\rm CP}^2$, P_1 first verifies that the following conditions hold (we note that $\mathcal{F}_{\rm CP}^2$ is parameterized by the same relation R_Π as $\mathcal{F}_{\rm CP}^1$):
   (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 11: The compiled protocol $\mathsf{Comp}(\Pi)$

note that the compiler does not use any additional cryptographic construct other than access to $\mathcal{F}_{\text{CP}}$. Consequently, the following proposition holds unconditionally, and even if the adversary and environment are computationally unbounded.

Proposition 8.1 *Let* Π *be a two-party protocol and let* $\mathsf{Comp}(\Pi)$ *be the protocol obtained by applying the compiler of Figure* 11 *to* Π. *Then, for every* malicious *adversary* $\mathcal{A}$ *that interacts with* $\mathsf{Comp}(\Pi)$ *in the* $\mathcal{F}_{\text{CP}}$*-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* $\mathcal{Z}$*,*

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

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

Corollary 8.2 *Let* $\mathcal{F}$ *be a two-party functionality and let* Π *be a non-trivial protocol that UC 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 UC 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 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 $\mathcal{A}$ (in the name of 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 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 for P_2 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: Recall that the input commitment phase consists only of P_1 sending a commit message to $\mathcal{F}^1_{\text{CP}}$. We deal with the case that P_1 is not corrupted separately from the case that P_1 is corrupted. First, in the case that 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 3.1.1 for an exact description of how values are written to the parties' input tapes in the semi-honest model.)

Dealing with Π 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 analogous message that it expects to see in $\mathsf{Comp}(\Pi)$: $(\mathsf{CP\text{-}proof}, sid_1, (m, r_1^2, \overline{m}_1))$, 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 $(\mathsf{CP\text{-}proof}, sid_2, (m, r_2^1, \overline{m}_2))$, where r_2^1 and $\overline{m}_2$ are the analogous values to the previous case.

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

Dealing with $\mathsf{Comp}(\Pi)$ messages sent internally by corrupted parties: Assume that P_1 is corrupted. If $\mathcal{A}$, controlling P_1, sends a message $(\mathsf{CP\text{-}prover}, sid_1, (m, {r'}_1^2, \overline{m}'_1))$, 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 $(\mathsf{CP\text{-}proof}, sid_1, (m, {r'}_1^2, \overline{m}'_1))$, as $\mathcal{A}$ expects to receive from $\mathcal{F}_{\mathrm{CP}}^1$. Then, when $\mathcal{A}$ delivers this $(\mathsf{CP\text{-}proof}, \ldots)$ message from $\mathcal{F}_{\mathrm{CP}}^1$ 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.[26] 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 $\mathsf{CP\text{-}proof}$ message in the name of a corrupt P_2 is analogous.

Dealing with corruption of parties:[27] 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 $\mathsf{Comp}(\Pi)$. This is done as follows. The only additional internal state that P_1 keeps in $\mathsf{Comp}(\Pi)$ is the random string r_1^1 (this is the string that P_1 commits to in the random tape generation phase of $\mathsf{Comp}(\Pi)$). Therefore, $\mathcal{A}'$ 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 $\mathsf{Comp}(\Pi)$. In this way, $\mathcal{A}'$ prepares a simulated internal state of P_1 in $\mathsf{Comp}(\Pi)$ 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 $\mathsf{Comp}(\Pi)$ in the $\mathcal{F}_{\mathrm{CP}}$-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 $\mathsf{Comp}(\Pi)$ in the $\mathcal{F}_{\mathrm{CP}}$-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

[26] 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 $\mathsf{Comp}(\Pi)$ both have the same inputs, random tapes and series of incoming messages. Therefore, their outgoing messages are also the same.

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

uncorrupted party sends a message are immediate. To see that this holds also in the case that $\mathcal{A}'$ delivers messages sent by corrupted parties, observe the following facts:

1. *Random tape:* $\mathcal{A}'$ forces the random tape of a corrupted P_1 in the internal execution of $\mathsf{Comp}(\Pi)$ with $\mathcal{A}$ to be the random tape of the semi-honest party P_1 externally executing Π.

2. *Input:* $\mathcal{A}'$ 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 the malicious $\mathcal{A}$ committed to for the internal P_1 are exactly the same as the input and random tapes used by the external, semi-honest P_1.

3. *Message generation and delivery:* In the simulation with malicious $\mathcal{A}$, semi-honest $\mathcal{A}'$ obtains all the inputs committed to by a corrupted P_1. Consequently, $\mathcal{A}'$ is able to verify at every step if the message m sent by $\mathcal{A}$, in the name of corrupted P_1, is according to the protocol specification. If yes, then it is 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, it is 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 $\mathcal{F}_{\text{CP}}$.

4. *Corruptions:* 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 EXEC are identical. ■

8.2 Conclusions

Combining the semi-honest protocol of Proposition 4.3 with the compilation obtained in Corollary 8.2, we have that for any two-party ideal functionality $\mathcal{F}$, there exists a protocol that UC realizes $\mathcal{F}$ in the $\mathcal{F}_{\text{CP}}$-hybrid model (in the presence of malicious adversaries). Combining this with the fact that assuming the existence of one-way functions, $\mathcal{F}_{\text{CP}}$ can be UC realized in the $\mathcal{F}_{\text{ZK}}$-hybrid model (Proposition 7.1), and using the UC composition theorem (Theorem 3.3), we obtain universally composable general two-party computation in the $\mathcal{F}_{\text{ZK}}$-hybrid model. That is,

Theorem 8.3 (Theorem 2.2 – formally restated): *Assume that enhanced trapdoor permutations exist. Then, for any well-formed two-party ideal functionality $\mathcal{F}$, there exists a non-trivial protocol that UC realizes $\mathcal{F}$ in the $\mathcal{F}_{\text{ZK}}$-hybrid model in the presence of* malicious, static *adversaries. Furthermore, if one-way functions and 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 UC realizes $\mathcal{F}$ in the $\mathcal{F}_{\text{ZK}}$-hybrid model in the presence of* malicious, adaptive *adversaries.*

Recall that, under the assumption that enhanced trapdoor permutations exist, functionality $\hat{\mathcal{F}}_{\text{ZK}}$ (the multi-session extension of $\mathcal{F}_{\text{ZK}}$) can be UC 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 3.4) to obtain the following corollary:

Corollary 8.4 *Assume that enhanced trapdoor permutations exist. Then, for any well-formed two-party ideal functionality $\mathcal{F}$, there exists a non-trivial protocol that UC 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 also exist, then for any adaptively well-formed two-party functionality $\mathcal{F}$, there exists a non-trivial protocol that UC 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 $\mathcal{F}_{\text{CRS}}$.*

9 Multi-Party Secure Computation

This section extends the two-party constructions of Sections 5–8 to the multi-party setting, thereby proving Theorem 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 used in all 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.

9.1 Multi-Party Secure Computation for Semi-Honest Adversaries

In this section, we sketch the construction of non-trivial protocols that UC realize any adaptively well-formed functionality $\mathcal{F}$ for semi-honest adversaries. (Recall the definition of adaptively well-formed functionalities in Section 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 α 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 9.1 *Assume that enhanced trapdoor permutations exist. Then, for any well-formed (multi-party) ideal functionality $\mathcal{F}$, there exists a non-trivial protocol that UC 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 UC 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 UC realize the two-party functionality $\mathcal{F}^4_{\text{OT}}$, which plays a central role even in the multi-party case.

We begin our proof of Proposition 9.1 by presenting a non-trivial multi-party protocol $\Pi_{\mathcal{F}}$ that UC 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 4.2 (the extensions to the multi-party case are straightforward).

Protocol $\Pi_{\mathcal{F}}$ (for UC 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 [G98, 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 4.2. 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 $\Pi_{\mathcal{F}}$ UC realizes any adaptively well-formed multi-party functionality $\mathcal{F}$:

Claim 9.2 *Let $\mathcal{F}$ be an adaptively well-formed multi-party functionality. Then, protocol $\Pi_{\mathcal{F}}$ UC realizes $\mathcal{F}$ in the $\mathcal{F}_{\mathrm{OT}}$-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). 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 pairwise oblivious transfer calls to $\mathcal{F}_{\mathrm{OT}}$. 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 $\mathcal{F}_{\mathrm{OT}}$ is the session-identifier used. This is therefore easily simulated by $\mathcal{S}$. (If the sender is corrupted, then its input table to $\mathcal{F}_{\mathrm{OT}}$ 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 output lines are already defined for any party P_j that is already corrupted. (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 (when referring to corrupted and uncorrupted P_j below, we mean the current corruption status and not the status at the time that the given oblivious transfer was executed):

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 above 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.[28] Upon the corruption of P_i, simulator $\mathcal{S}$ receives P_i's output

[28] 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.

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; $\mathcal{S}$ can do this because it now holds $\mathcal{F}$'s random tape and the inputs of all parties in all the activations of $\mathcal{F}$. Let c be the computed state string and let r equal $\mathcal{F}$'s length-m random tape. Now, apart from P_i, all the other parties have already been corrupted. Therefore, the shares of the random tape r_j are already 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 $\mathcal{F}_{\mathrm{OT}}$-hybrid model. The analysis is similar to the one for the two-party case and is omitted. ■

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 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 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*. In subsequent work to ours, Goldwasser and Lindell [GL02] show that in our model where message delivery is not guaranteed, functionality $\mathcal{F}_{\text{BC}}$ can be UC realized by a non-trivial protocol, for malicious adversaries, for any number of corrupted parties and without any setup assumptions. (In fact, the broadcast functionality defined in [GL02] is different to the one here in that it requires that *all* the parties $P_1, \ldots, P_n$ receive every broadcasted message. In the functionality here, however, only a subset of the parties $\mathcal{P}$ receive the message. Furthermore, all the parties receiving the message know the identities of the parties in the set $\mathcal{P}$. This gap can be easily bridged by having the broadcasting party include the set $\mathcal{P}$ along with the broadcasted message x. Then, all parties receive $(\mathcal{P}, x)$ and the parties not in $\mathcal{P}$ simply discard the message.)

We remark that in contrast to all the other functionalities defined in this paper, the *entire* $(\mathsf{broadcast}, sid, \mathcal{P}, x)$ message is included in the *public header* (i.e., x is not private and can be read by the adversary). This is consistent with the fact that the adversary can read any message x sent by one party to another party, using its outgoing communication tape. (That is, we view the broadcast to be like regular communication between parties and not like private communication that takes place between parties and ideal functionalities.)

9.3 One-to-Many Commitment, Zero-Knowledge and Commit-and-Prove

In this section, we present *one-to-many* extensions of the commitment, zero-knowledge and commit-and-prove functionalities. The extensions are called "one-to-many" because in all of them, a *single* party commits/proves to *many* receivers/verifiers.

One-to-many UC commitments. We begin by defining a one-to-many extension of the UC commitment functionality, denoted $\mathcal{F}_{\text{MCOM}}^{\text{1:M}}$. In this functionality, the committing party commits to a value to many receivers. The formal definition appears in Figure 13. Similarly to the two-party case, the commitment functionality is presented as a *multi-session functionality.* From here on, the JUC theorem of [CR02] is applied and we consider only single-session functionalities (see Section 3.2 for more explanation). We denote the single session analog to $\mathcal{F}_{\text{MCOM}}^{\text{1:M}}$ by $\mathcal{F}_{\text{COM}}^{\text{1:M}}$.

The key observation in realizing the $\mathcal{F}_{\text{MCOM}}^{\text{1:M}}$ functionality is that Protocol UAHC (of Section 5) that UC realizes the two-party commitment functionality $\mathcal{F}_{\text{MCOM}}$ is *non-interactive*. Therefore, the

Functionality $\mathcal{F}_{\text{MCOM}}^{\text{1:M}}$

$\mathcal{F}_{\text{MCOM}}^{\text{1:M}}$ 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 13: One-to-Many multi-session commitment

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 UC realizes $\mathcal{F}_{\text{MCOM}}^{\text{1:M}}$ is almost identical to the proof that Protocol UAHC UC realizes $\mathcal{F}_{\text{MCOM}}$, and is omitted. We do, however, mention one important point. The commitment string is broadcast using the $\mathcal{F}_{\text{BC}}$ 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 $\mathcal{F}_{\text{MCOM}}^{\text{1:M}}$ functionality that ensures that all parties receive the same commitment for the same identifier pair $(sid, ssid)$. We therefore have the following:

Proposition 9.3 *Assuming the existence of enhanced trapdoor permutations, there exists a (non-interactive) protocol that UC realizes $\mathcal{F}_{\text{MCOM}}^{\text{1:M}}$ in the $(\mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BC}})$-hybrid model[29], in the presence of malicious, adaptive adversaries. Furthermore, this protocol uses only a single copy of $\mathcal{F}_{\text{CRS}}$.*

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 $\mathcal{F}_{\text{ZK}}^{\text{1:M}}$, appears in Figure 14. (For simplicity, in the multi-party case we concentrate on single-session zero-knowledge, constructed using a single-session version of $\mathcal{F}_{\text{MCOM}}^{\text{1:M}}$. 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 $\mathcal{F}_{\text{ZK}}^{\text{1:M}}$.)

Functionality $\mathcal{F}_{\text{ZK}}^{\text{1:M}}$

$\mathcal{F}_{\text{ZK}}^{\text{1:M}}$ 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 14: Single-session, One-to-Many zero-knowledge

[29] In the $(\mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BC}})$-hybrid model, all parties have ideal access to both the common reference string functionality $\mathcal{F}_{\text{CRS}}$ and the ideal broadcast functionality $\mathcal{F}_{\text{BC}}$.

As with the case of commitments, a non-interactive protocol that UC realizes the two-party zero-knowledge functionality $\mathcal{F}_{\text{ZK}}$ could be directly used to realize $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$. For the case of static adversaries, the protocol of [d^{+}01] 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 [CF01]. Their protocol is basically that of parallel Hamiltonicity (cf. [B86]), except that the commitments used are universally composable. Our extension of this protocol to the one-to-many case follows the methodology of [G98] and is presented in the proof of the following proposition:

Proposition 9.4 *There exists a protocol that UC realizes $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$ in the $(\mathcal{F}_{\text{COM}}^{1:\text{M}}, \mathcal{F}_{\text{BC}})$-hybrid model, in the presence of malicious, adaptive adversaries.*

Proof (sketch): The protocol for realizing $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$ 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 [CF01], 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 $\mathcal{F}_{\text{COM}}^{1:\text{M}}$, rather than the two-party $\mathcal{F}_{\text{COM}}$. ($\mathcal{F}_{\text{COM}}^{1:\text{M}}$ is the single-session parallel to $\mathcal{F}_{\text{MCOM}}^{1:\text{M}}$.) 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 $(\mathsf{ZK\text{-}proof}, sid, P_i, \mathcal{P}, x)$, if and only if all the pairwise proofs are accepting. Note that other than the use of $\mathcal{F}_{\text{COM}}^{1:\text{M}}$, no cryptographic primitives are used. Indeed, security of this protocol in the $\mathcal{F}_{\text{COM}}^{1:\text{M}}$-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 [CF01] 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 $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$, 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 $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$ works simply by running the simulator of the two-party protocol of [CF01] 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 $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$ thus runs the extractor for the protocol of [CF01] 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 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*.)

Our protocol for UC 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 $\mathsf{ACP}^{1:\text{M}}$, is very similar to Protocol ACP

Functionality $\mathcal{F}_{\text{CP}}^{\text{1:M}}$

$\mathcal{F}_{\text{CP}}^{\text{1:M}}$ 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 15: One-to-Many commit-and-prove

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 $\vec{c} = \mathsf{aHC}_{\vec{s}}(w;\vec{r}) = (\mathsf{aHC}_{s_1}(w;r_1), \ldots, \mathsf{aHC}_{s_\ell}(w;r_\ell))$. 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 $\mathsf{ACP}^{\text{1:M}}$ uses three different copies of $\mathcal{F}_{\text{ZK}}^{\text{1:M}}$, where each copy is parameterized by a different relation. The copies are denoted $\mathcal{F}_{\text{ZK,T}}^{\text{1:M}}$ (for the initialization phase), $\mathcal{F}_{\text{ZK,C}}^{\text{1:M}}$ (for the commit stage) and $\mathcal{F}_{\text{ZK,P}}^{\text{1:M}}$ (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 $\mathsf{ACP}^{\text{1:M}}$ 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 $\mathsf{ACP}^{\text{1:M}}$. The protocol is presented in Figure 16.

Proposition 9.5 *Assuming the existence of one-way functions, Protocol* $\mathsf{ACP}^{\text{1:M}}$ *of Figure* 16 *UC realizes* $\mathcal{F}_{\text{CP}}^{\text{1:M}}$ *in the* $(\mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{ZK}}^{\text{1:M}})$*-hybrid model, in the presence of adaptive adversaries.*

Proof (sketch): The proof of this proposition is very similar to the proof of Proposition 7.2 for the two-party case. Let $\mathcal{A}$ be an adaptive adversary who operates against Protocol $\mathsf{ACP}^{\text{1:M}}$ in the $\mathcal{F}_{\text{ZK}}^{\text{1:M}}$-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 $\mathsf{ACP}^{\text{1:M}}$ in the $\mathcal{F}_{\text{ZK}}^{\text{1:M}}$-hybrid model or with $\mathcal{S}$ in the ideal process for $\mathcal{F}_{\text{CP}}^{\text{1:M}}$. Simulator $\mathcal{S}$ operates by running a simulated copy of $\mathcal{A}$ and using $\mathcal{A}$ in order to interact with $\mathcal{Z}$ and $\mathcal{F}_{\text{CP}}^{\text{1:M}}$. $\mathcal{S}$ works as follows.

Simulating the initialization phase: $\mathcal{S}$ records all 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 $\mathcal{F}_{\text{ZK,T}}^{\text{1:M}}$.

Protocol $\mathsf{ACP}^{1:\mathrm{M}}$

- **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 $\mathcal{F}_{\mathrm{BC}}$ 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 $\mathcal{F}^{1:\mathrm{M}}_{\mathrm{ZK,T}}$, where $\mathcal{F}^{1:\mathrm{M}}_{\mathrm{ZK,T}}$ is parameterized by the relation R_T defined by:
    $$R_T \stackrel{\mathrm{def}}{=} \{(s,t) \mid s = f(t)\}$$
    3. Upon receiving $(\mathsf{ZK\text{-}proof}, sid_T, P_j, \mathcal{P}, s_j)$ from $\mathcal{F}^{1:\mathrm{M}}_{\mathrm{ZK,T}}$, 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 $\vec{c} = \mathsf{aHC}_{\vec{s}}(w; \vec{r})$ 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 $(\mathsf{ZK\text{-}prover}, sid_C, \mathcal{P}, (\vec{s}, \vec{c}), (w, \vec{r}))$ to $\mathcal{F}^{1:\mathrm{M}}_{\mathrm{ZK,C}}$, where $\mathcal{F}^{1:\mathrm{M}}_{\mathrm{ZK,C}}$ is parameterized by the relation R_C defined by:
    $$R_C \stackrel{\mathrm{def}}{=} \{(\vec{s}, \vec{c}), (w, \vec{r}) \mid \vec{c} = \mathsf{aHC}_{\vec{s}}(w; \vec{r})\}$$
    (That is, R_C verifies that $\vec{c}$ 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 $\vec{c}$ and random strings $\vec{r} = (r_1, \ldots, r_\ell)$.
    2. Upon receiving $(\mathsf{ZK\text{-}proof}, sid_C, P_i, \mathcal{P}, (\vec{s}', \vec{c}))$ from $\mathcal{F}^{1:\mathrm{M}}_{\mathrm{ZK,C}}$, 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 $\vec{c}$ 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 $\mathcal{F}^{1:\mathrm{M}}_{\mathrm{ZK,P}}$, where $\overline{c}$, $\overline{w}$ and $\overline{r}$ are the lists described above. Let $\overline{w} = (w_1, \ldots, w_m)$, $\overline{c} = (\vec{c}_1, \ldots, \vec{c}_m)$ and $\overline{r} = (\vec{r}_1, \ldots, \vec{r}_m)$. Then, $\mathcal{F}^{1:\mathrm{M}}_{\mathrm{ZK,P}}$ is parameterized by the relation R_P defined by:
    $$R_P \stackrel{\mathrm{def}}{=} \{((x, \vec{s}, \overline{c}), (\overline{w}, \overline{r})) \mid R(x, \overline{w}) = 1 \ \&\ \forall j \ \vec{c}_j = \mathsf{aHC}_{\vec{s}}(w_j; \vec{r}_j)\}$$
    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 $\mathcal{F}^{1:\mathrm{M}}_{\mathrm{ZK,P}}$, 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 16: A protocol for realizing $\mathcal{F}^{1:\mathrm{M}}_{\mathrm{CP}}$ for adaptive adversaries

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 $(\mathsf{ZK\text{-}prover}, sid_C, \mathcal{P}, (\vec{s}, \vec{c}), (w, \vec{r}))$ that $\mathcal{A}$ sends to $\mathcal{F}_{\text{ZK,C}}^{1:\text{M}}$. $\mathcal{S}$ checks that $\vec{s}$ is as generated in the initialization phase and that $\vec{c} = \mathsf{aHC}_{\vec{s}}(w; \vec{r})$. If yes, then $\mathcal{S}$ internally passes $\mathcal{A}$ the message $(\mathsf{ZK\text{-}proof}, sid_C, \mathcal{P}, (\vec{s}, \vec{c}))$ and externally sends $(\mathsf{commit}, sid, \mathcal{P}, w)$ to $\mathcal{F}_{\text{CP}}^{1:\text{M}}$. Furthermore, $\mathcal{S}$ adds the commitment $\vec{c}$ 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 a message $(\mathsf{ZK\text{-}prover}, sid_P, (x, \vec{s}, \overline{c}), (\overline{w}, \overline{r}))$ that $\mathcal{A}$ sends to $\mathcal{F}_{\text{ZK,P}}^{1:\text{M}}$. $\mathcal{S}$ then checks that $\vec{s}$ is as generated in the initialization phase, 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 $\mathcal{F}_{\text{CP}}^{1:\text{M}}$. 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 P_i writes a commit message on its outgoing communication tape for $\mathcal{F}_{\text{CP}}$, simulator $\mathcal{S}$ internally simulates P_i writing an appropriate $\mathsf{ZK\text{-}prover}$ message on its outgoing communication tape (recall that only the commit header is viewed by $\mathcal{S}$, and that it only needs to write the $\mathsf{ZK\text{-}prover}$ header). Then, when $\mathcal{A}$ delivers the $\mathsf{ZK\text{-}prover}$ message to $\mathcal{F}_{\text{ZK,C}}^{1:\text{M}}$, simulator $\mathcal{S}$ delivers the commit message from P_i to $\mathcal{F}_{\text{CP}}^{1:\text{M}}$. Furthermore, $\mathcal{S}$ computes $\vec{c} = \mathsf{aHC}_{\vec{s}}(0^k; \vec{r})$ and hands $\mathcal{A}$ the message $(\mathsf{ZK\text{-}proof}, sid_C, P_i, \mathcal{P}, (\vec{s}, \vec{c}))$, as if coming from $\mathcal{F}_{\text{ZK,C}}^{1:\text{M}}$. Now, when $\mathcal{A}$ delivers this $\mathsf{ZK\text{-}proof}$ message from $\mathcal{F}_{\text{ZK,C}}^{1:\text{M}}$ to a party $P_j \in \mathcal{P}$ in the internal simulation,, then $\mathcal{S}$ delivers the $\mathsf{receipt}$ message from $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ to P_j. (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 5.)

The simulation of the prove phase is carried out as follows. Whenever an uncorrupted P_i writes a $\mathsf{CP\text{-}prover}$ message on its outgoing communication tape for $\mathcal{F}_{\text{CP}}^{1:\text{M}}$, simulator $\mathcal{S}$ internally simulates P_i writing the appropriate $\mathsf{ZK\text{-}prover}$ message on its outgoing communication tape for $\mathcal{F}_{\text{ZK,P}}^{1:\text{M}}$. When $\mathcal{A}$ delivers this $\mathsf{ZK\text{-}prover}$ message in the internal simulation, then $\mathcal{S}$ delivers P_i's $\mathsf{CP\text{-}prover}$ message to $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ and receives back the message $(\mathsf{CP\text{-}proof}, sid, x)$. Then, $\mathcal{S}$ internally passes $\mathcal{A}$ the message $(\mathsf{ZK\text{-}proof}, sid_P, (x, \vec{s}, \overline{c}))$ as if it came from $\mathcal{F}_{\text{ZK,P}}^{1:\text{M}}$, where $\overline{c}$ is the list of simulated commitments generated above. When $\mathcal{A}$ delivers this $\mathsf{ZK\text{-}proof}$ message from $\mathcal{F}_{\text{ZK,P}}^{1:\text{M}}$ to a party $P_j \in \mathcal{P}$ in the internal simulation, then $\mathcal{S}$ delivers the $\mathsf{CP\text{-}proof}$ message from $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ to P_j.

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 $\mathsf{ACP}^{1:\text{M}}$ 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 7.2).

The analysis of the correctness of the simulation is analogous to in the two-party case and is omitted. ■

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 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 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 11. In addition to commit-and-prove, the multi-party compiler here uses "standard" UC commitments for coin-tossing. However, in order to remain in the $\mathcal{F}_{\text{CP}}^{1:\text{M}}$-hybrid model (and not the $(\mathcal{F}_{\text{CP}}^{1:\text{M}}, \mathcal{F}_{\text{MCOM}}^{1:\text{M}})$-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 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 of Figure 17 achieves the desired result:

Proposition 9.6 (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 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 $\mathcal{Z}$,*

$$\text{REAL}_{\Pi,\mathcal{A}',\mathcal{Z}} \equiv \text{EXEC}^{\mathcal{F}_{\text{CP}}^{1:\text{M}}}_{\mathsf{Comp}(\Pi),\mathcal{A},\mathcal{Z}}$$

Proof (sketch): The proof sketch is very similar to the proof of Proposition 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:

$\mathsf{Comp}(\mathbf{\Pi})$

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 $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$.
   (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 $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$ to decommit to its value r_i^j. That is, P_i sends $(\mathsf{CP\text{-}prover}, sid_{i,j}, r_i^j)$ to $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$, where the relation parameterizing the $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$ functionality with identifier $sid_{i,j}$ is the identity relation (i.e., $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$ sends (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 $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$ 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 $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$ 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 $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$. 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 $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$ 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 $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$ 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 $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$ with sid_j is parameterized by the same relation R_Π as $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$ 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 2c above.
4. **Output:** Whenever Π generates an output value, $\mathsf{Comp}(\Pi)$ generates the same output value.

Figure 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:* The semi-honest adversary $\mathcal{A}'$ internally hands the malicious $\mathcal{A}$ the message $(\mathsf{receipt}, sid_j, P_j, \mathcal{P})$ from $\mathcal{F}_{\rm CP}^{\rm 1:M}$ (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 $\mathcal{F}_{\rm CP}^{\rm 1:M}$. $\mathcal{A}'$ 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 $\mathcal{F}_{\rm CP}^{\rm 1:M}$. $\mathcal{A}'$ 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 $\mathcal{F}_{\rm CP}^{\rm 1:M}$. 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 $\mathcal{F}_{\rm CP}^{\rm 1:M}$.

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 $\mathcal{A}$, in the name of a corrupted party P_i. Then, $\mathcal{A}'$ 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. ■

9.4.1 Conclusions

Combining the semi-honest protocol of Proposition 9.1 with the compilation obtained in Proposition 9.6, we have that for any multi-party ideal functionality $\mathcal{F}$, there exists a protocol that UC realizes $\mathcal{F}$ in the $\mathcal{F}_{\text{CP}}^{1:\text{M}}$-hybrid model (in the presence of malicious adversaries). Combining this with the fact that assuming the existence of one-way functions, $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ can be UC realized in the $(\mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{ZK}}^{1:\text{M}})$-hybrid model (Proposition 9.5), and using the UC composition theorem (Theorem 3.3), we obtain universally composable general two-party computation in the $(\mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{ZK}}^{1:\text{M}})$-hybrid model. That is:

Theorem 9.7 *Assume that enhanced trapdoor permutations exist. Then, for any well-formed multi-party ideal functionality $\mathcal{F}$, there exists a non-trivial protocol that UC realizes $\mathcal{F}$ in the $(\mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{ZK}}^{1:\text{M}})$-hybrid model in the presence of* malicious, static *adversaries. Furthermore, if one-way functions and 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 UC realizes $\mathcal{F}$ in the $(\mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{ZK}}^{1:\text{M}})$-hybrid model in the presence of* malicious, adaptive *adversaries.*

By Proposition 9.4, the zero-knowledge functionality $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$ can be UC realized in the $(\mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{COM}}^{1:\text{M}})$-hybrid model. In addition, by Proposition 9.3, under the assumption that enhanced trapdoor permutations exist, the multi-session $\mathcal{F}_{\text{MCOM}}^{1:\text{M}}$ can be UC realized in the $(\mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BC}})$-hybrid model. Applying the universal composition with joint state theorem (Theorem 3.4), we obtain that $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$ can be UC realized in the $(\mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BC}})$-hybrid model (with a single copy of the CRS). We therefore obtain that Theorem 9.7 can be restated in the $(\mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BC}})$-hybrid model only. Recalling that $\mathcal{F}_{\text{BC}}$ can be UC realized in the plain model [GL02], we obtain the following theorem:

Theorem 9.8 (Theorem 2.3 – restated): *Assume that enhanced trapdoor permutations exist. Then, for any well-formed multi-party ideal functionality $\mathcal{F}$, there exists a non-trivial protocol that UC 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 also exist, then for any adaptively* *well-formed multi-party ideal functionality $\mathcal{F}$, there exists a non-trivial protocol that UC realizes $\mathcal{F}$ in the $\mathcal{F}_{\text{CRS}}$-hybrid model in the presence of* malicious, adaptive *adversaries. In both cases, the protocol uses only a single copy of $\mathcal{F}_{\text{CRS}}$.*

Acknowledgements

The authors would like to thank Oded Goldreich for helpful discussions (and in particular, for suggesting the generalization of the commit-and-prove functionality to allow for many commitments).

References

[B91] D. Beaver. Secure Multi-party Protocols and Zero-Knowledge Proof Systems Tolerating a Faulty Minority. *Journal of Cryptology,* Springer-Verlag, 4:75–122, 1991.

[B97] D. Beaver. Plug and play encryption. In *CRYPTO'97,* Springer-Verlag (LNCS 1294), pages 75–89, 1997.

[BBM00] 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.

[BGW88] M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. 20*th STOC,* pages 1–10, 1988.

[B86] 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.

[BFM88] M. Blum, P. Feldman and S. Micali. Non-interactive zero-knowledge and its applications. In 20*th STOC,* pages 103–112, 1988.

[C00] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[C01] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In 42*nd FOCS*, pages 136–145. 2001. Full version available at `http://eprint.iacr.org/2000/067`.

[CFGN96] R. Canetti, U. Feige, O. Goldreich and M. Naor. Adaptively Secure Multi-Party Computation. In 28*th STOC*, pages 639–648, 1996.

[CF01] R. Canetti and M. Fischlin. Universally Composable Commitments. In *CRYPTO'01*, Springer-Verlag (LNCS 2139), pages 19–40, 2001.

[CKL03] R. Canetti, E. Kushilevitz and Y. Lindell. On the Limitations of Universal Composition Without Set-Up Assumptions. In *EUROCRYPT'03,* Springer-Verlag (LNCS 2656), pages 68–86, 2003.

[CR02] R. Canetti and T. Rabin. Universal Composition with Joint State. *Cryptology ePrint Archive,* Report 2002/047, `http://eprint.iacr.org/`, 2002.

[DN00] 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.

[DN02] I. Damgard and J.B. Nielsen. Perfect Hiding or Perfect Binding Universally Composable Commitment Schemes with Constant Expansion Factor. In *CRYPTO'02,* Springer-Verlag (LNCS 2442), pages 581–596, 2002.

[d$^+$01] A. De Santis, G. Di Crescenzo, R. Ostrovsky, G. Persiano and A. Sahai. Robust Non-interactive Zero-Knowledge. In *CRYPTO'01,* Springer-Verlag (LNCS 2139), pages 566–598, 2001.

[DP92] A. De Santis and G. Persiano. Zero-Knowledge Proofs of Knowledge Without Interaction. In 33*rd FOCS,* pages 427–436, 1992.

[DIO98] G. Di Crescenzo, Y. Ishai and R. Ostrovsky. Non-Interactive and Non-Malleable Commitment. In 30*th STOC,* pages 141–150, 1998.

[DKOS01] 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.

[DDN00] D. Dolev, C. Dwork and M. Naor. Non-Malleable Cryptography. *SIAM Journal on Computing,* 30(2):391–437, 2000.

[DNS98] C. Dwork, M. Naor, and A. Sahai. Concurrent Zero-Knowledge. In 30*th STOC*, pages 409–418, 1998.

[EGL85] S. Even, O. Goldreich and A. Lempel. A randomized protocol for signing contracts. In *Communications of the ACM,* 28(6):637–647, 1985.

[FS89] U. Feige and A. Shamir. Zero-Knowledge Proofs of Knowledge in Two Rounds. In *CRYPTO'89*, Springer-Verlag (LNCS 435), pages 526–544, 1989.

[GM00] J. Garay and P. Mackenzie. Concurrent Oblivious Transfer. In 41*st FOCS*, pages 314–324, 2000.

[G98] O. Goldreich. *Secure Multi-Party Computation*. Manuscript. Preliminary version, 1998. Available from `http://www.wisdom.weizmann.ac.il/~oded/pp.html`.

[G01] O. Goldreich. *Foundations of Cryptography: Volume 1 – Basic Tools.* Cambridge University Press, 2001.

[G03] O. Goldreich. *Foundations of Cryptography: Volume 2 – Basic Applications.* To be published by the Cambridge University Press. Available from `http://www.wisdom.weizmann.ac.il/~oded/PSBookFrag/v2.ps`.

[GL89] O. Goldreich and L. Levin. A Hard Predicate for All One-Way Functions. In 21*st STOC,* pages 25–32, 1989.

[GMW87] O. Goldreich, S. Micali and A. Wigderson. How to Play any Mental Game – A Completeness Theorem for Protocols with Honest Majority. In *19th STOC,* pages 218–229, 1987. For details see [G98].

[GL90] 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.

[GL02] S. Goldwasser and Y. Lindell. Secure Computation Without Agreement. In 16*th DISC,* Springer-Verlag (LNCS 2508), pages 17–32 2002.

[GMR89] S. Goldwasser, S. Micali and C. Rackoff The Knowledge Complexity of Interactive Proof Systems. *SIAM Journal on Computing,* 18(1):186–208, 1989.

[K89] J. Kilian. *Uses of Randomness in Algorithms and Protocols.* The ACM Distinguished Dissertation 1989, MIT press.

[L03] Y. Lindell. General Composition and Universal Composability in Secure Multi-Party Computation. In 44*th FOCS*, 2003.

[LLR02] Y. Lindell, A. Lysysanskaya and T. Rabin. On the Composition of Authenticated Byzantine Agreement. In 34$^{\text{th}}$ *STOC*, pages 514–523, 2002.

[MR91] S. Micali and P. Rogaway. Secure computation. Unpublished manuscript, 1992. Preliminary version in *CRYPTO'91*, Springer-Verlag (LNCS 576), pages 392–404, 1991.

[N91] M. Naor. Bit Commitment using Pseudorandom Generators. *Journal of Cryptology*, 4(2):151–158, 1991.

[PW00] 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.

[R81] M. Rabin. How to Exchange Secrets by Oblivious Transfer. Tech. Memo TR-81, Aiken Computation Laboratory, Harvard U., 1981.

[RB89] T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multi-party Protocols with Honest Majority. In 21*st STOC*, pages 73–85, 1989.

[RK99] R. Richardson and J. Kilian. On the Concurrent Composition of Zero-Knowledge Proofs. In *EUROCRYPT'99*, Springer-Verlag (LNCS 1592), pages 415–413, 1999.

[S99] A. Sahai. Non-Malleable Non-Interactive Zero-Knowledge and Adaptive Chosen-Ciphertext Security. In 40*th FOCS*, pages 543–553, 1999.