Security and Composition of Cryptographic Protocols: A Tutorial[*]

Ran Canetti[†]

December 18, 2006

Abstract

What does it mean for a cryptographic protocol to be "secure"? Capturing the security requirements of cryptographic tasks in a meaningful way is a slippery business: On the one hand, we want security criteria that prevent "all feasible attacks" against a protocol. On the other hand, we want our criteria to not be overly restrictive; that is, we want them to accept those protocols that do not succumb to "feasible attacks".

This tutorial studies a general methodology for defining security of cryptographic protocols. The methodology, often dubbed the "trusted party paradigm", allows for defining the security requirements of practically any cryptographic task in a unified and natural way. We first review a basic formulation that captures security in isolation from other protocol instances. Next we address the *secure composition* problem, namely the vulnerabilities resulting from the often unexpected interactions among different protocol instances that run alongside each other in the same system. We demonstrate the limitations of the basic formulation and review a formulation that guarantees security of protocols even in general composite systems.

[*]This tutorial is based on a contribution to the distributed computing column of SIGACT News V. 37, No. 3 & 4.

[†]IBM T.J. Watson Research Center, canetti@csail.mit.edu. Supported by NSF CyberTrust Grant #0430450.

Contents

1 Introduction

Cryptographic protocols, namely distributed algorithms that aim to guarantee some "security properties" in face of adversarial behavior, have become an integral part of our society and everyday lives. Indeed, we have grown accustomed to relying on the ubiquity and functionality of computer systems, whereas these systems make crucial use of cryptographic protocols to guarantee their "expected functionality." Furthermore, the perceived security properties of cryptographic protocols and the functionality expected from applications that use them is being used by lawmakers to modify the ground rules of our society. It is thus crucial that we have sound understanding of how to specify, develop, and analyze cryptographic protocols.

The need for sound understanding is highlighted by the empirical fact that cryptographic protocols have been notoriously "hard to get right," with subtle flaws in protocols being discovered long after development, and in some cases even after deployment and standardization. In fact, even *specifying* the security properties required from protocols for a given task in a rigorous and meaningful way has proved to be elusive.

The goal of this tutorial is to introduce the reader to the problems associated with formulating and asserting security properties of protocols, and to present a general methodology for modeling protocols and asserting their security properties. The tutorial attempts to be accessible to non-cryptographers and cryptographers alike. In particular, for the most part it assumes very little prior knowledge in cryptography. Also, while the main focus is on the foundational aspects of specifying security, the text attempts to be accessible and useful to practitioners as well as theoreticians. Indeed, the considered security concerns are realistic ones, and the end goal is to enable analyzing the security of real-world protocols and systems.

Cryptographic tasks. In general, a cryptographic task, or a *protocol problem,* involves a set parties that wish to perform some joint computational task based on their respective local inputs, while guaranteeing some "security properties" in the face of various types of "adversarial behavior" by different components of the system and its users.

To get some feel for the range of issues and concerns involved, we briefly review some of the commonplace cryptographic tasks considered in the literature. Let us start with the very basic goal of guaranteeing secure communication between parties, in face of an external adversarial entity (or entities) that have some access to the communication network. When the adversarial access enables only recording of the transmissions, the most central concern that comes to mind is *secrecy,* namely guaranteeing that the actual transmissions leak as little as possible on the communicated information. When adversarial entities can obtain also *active* control over the network the secrecy concern becomes more intricate, and furthermore an even more basic concerns arises: how to guarantee the *authenticity* of messages, namely finding out whether a received message was indeed sent by its claimed sender. Additional concerns include anonymity, namely the ability to hide the identities of the communicating parties, and non-repudiation, namely the ability to prove to a third party that the communication took place. Central tasks that are typically needed to guarantee secure communication include encryption, digital signatures, and key-exchange, where two parties wish to agree on a random value (a key) that is known only to them.

Another set of tasks, often called two-party tasks, involve two parties who are *mutually distrustful* but still wish to perform some joint computation. Here the only adversarial behavior under consideration is typically that of the parties themselves, and the communication medium is treated as trusted. One such task is zero-knowledge (as in [GMRa89]), where one party wishes to convince the other in the correctness of some statement without disclosing any additional information on top of the mere fact that the statement is correct. Another example is commitment (as in [B82]), where a party C can *commit* to a secret value x, by providing some "commitment information" c that keeps x secret, while guaranteeing to a verifier that C can later come up with only one value x that's consistent with c. Another example is coin-tossing [B82], namely the task where two parties want to agree on a bit, or a sequence of bits, that are taken from some predefined distribution, say the uniform distribution. This should hold even if one of the parties is trying to bias the output towards some value. In addition to being natural tasks on their own, protocols for these tasks are often used as building blocks within more complex protocols.

More generally, cryptographic tasks may involve multiple parties with intricate trust relationships, and exhibit a wide variety of secrecy and correctness requirements. Furthermore, in addition to plain correctness and secrecy, there are typically other task-specific concerns. We briefly mention some examples: Electronic voting, in a number of contexts, require careful balancing among correctness, public accountability, privacy and deniability. Electronic-commerce applications such as on-line auctions, on line trading and stock markets, and plain on-line shopping require *fairness* in completion of the transaction, as well as the ability to resolve disputes in an acceptable way. On-line gambling tasks require, in addition, the ability to guarantee fair distribution of the outcomes. Privacy-preserving computations on databases introduce a host of additional concerns and goals, such as providing statistical information while preserving the privacy of individual entries, obtaining data while hiding from the database which data was obtained, and answering queries that depend on several databases without leaking additional information in the process. Secure distributed depositories, either via a centrally-managed distributed system or in an open, peer-to-peer fashion, involve a host of additional secrecy, anonymity, availability and integrity concerns.

Cryptographic protocols. There is vast literature describing protocols aimed at solving the problems mentioned above, and many others, in a variety of settings. Out of this literature, let us mention only the works of Yao [Y86], and Goldreich, Micali and Wigderson [GMW87, G04], which give a mechanical way to generate protocols for solving practically *any* multi-party cryptographic protocol problem "in a secure way", assuming authenticated communication. (These constructions do not cover all tasks; for instance, they do not address the important problem of *obtaining* authenticated communication. Still, they are very general.)

Defining security of protocols. But, what does it mean for a cryptographic protocol to solve a given protocol problem, or a cryptographic task, "in a secure way"? How can we formalize the relevant security requirements in a way that makes mathematical sense, matches our informal intention, and at the same time can also be met by actual protocols? This turns out to be a tricky business.

Initially, definitions of security were problem-specific. That is, researchers came up with ad-hoc models of protocols and sets of requirements that seem to match the intuitive perception of the problem at hand. In addition, definitions were often tailored to capture the properties of specific solutions or protocols. However, as pointed out already in [Y82A], we would like to have a general framework for specifying the security properties of different tasks. A general framework allows for uniform and methodological specification of security properties. Such a specification methodology may provide better understanding of requirements and their formalization. It is also likely to result in fewer flaws in formulating the security requirements of tasks. In fact, it can be argued that having a general definitional framework is *essential* for understanding the notion of security of protocols.

Yet there is another, more concrete argument in favor of having a general analytical framework. Traditionally, notions of security tend to be very sensitive to the specific "execution environment" in which the protocol is running, and in particular to the other protocols running in the system at the same time. Thus, a set of requirements that seem appropriate in one setting may easily become insufficient when the setting is changed only slightly. This is a serious drawback when trying to build secure systems that make use of cryptographic protocols. Here a general analytical framework with a uniform methodology of specifying security requirements can be very useful: It allows formulating statements such as "A protocol that realizes some task can be used in conjunction with any protocol that makes use of this task, without bad interactions," or "Protocols that realize this task continue to realize it in any execution environment, regardless of what other protocols run in the system." Such *security-preserving composition* theorems are essential for building security protocols in a systematic way. They can be meaningful only in the context of a general framework for representing cryptographic protocols.

Several general frameworks for representing cryptographic protocols and specifying the security requirements of tasks were developed over the years, e.g. [GL90, MR91, B91, BCG93, PW94, LMMS98, C00, HM00, DM00, PW00, PW01, C01, MRST06, MMS03, K06]. All of these frameworks follow in one way or another the same underlying definitional approach, called the *trusted-party paradigm.* Still, these frameworks differ greatly in their expressibility (i.e., the range of security concerns and tasks that can be captured), in the models addressed, and in many significant technical details. They also support different types of security-preserving composition theorems.

This tutorial. This tutorial concentrates on the trusted-party definitional paradigm and the security it provides. Special attention is given to the importance of security-preserving composition in cryptographic protocols, and to the composability properties of this paradigm in particular. We also briefly discuss the relations with other (non-cryptographic) general approaches for modeling distributed protocols and analyzing their properties. For sake of concreteness, we base the current treatment on two specific formalizations of the trusted-party paradigm; other formulations are surveyed in the Appendix. The first formalization [C00] is somewhat simpler and provides a rather basic notion of security, with limited expressibility and a limited form of security-preserving composition. The second one [C01] is more general in terms of expressibility of concerns and situations, and also enables a very general form of security-preserving composition. The presentation here tries to "de-couple" the expressibility aspect from the aspect of the security level; indeed, we believe that the two aspects are very different from each other.

We start (in Section 2) with very high-level motivation and exposition of the trusted-party paradigm. We first demonstrate the failure of some naive approaches for defining security. We then present the paradigm in an abstract form and argue why it could allow overcoming the same pitfalls. We also try to demonstrate the intuitive appeal of this paradigm.

We then continue to develop (in Section 3) a highly simplified formalization of the general paradigm, that deals only with two parties that wish to compute a pre-specified function of their inputs, once, in isolation. While considerably restricted in its expressibility, this formulation (which is a restricted case of [C00]) allows concentrating on the main ideas without much of the complexity of the general case.

Section 4 presents a generalization of the model from Section 3 that allows capturing general cryptographic tasks, including multi-party tasks, reactive tasks (i.e. tasks where parties provide multiple inputs and receive multiple outputs), as well as fine-tuning of the security requirements.

This model can be seen as a *generalization* of [c00]. Still, this model concentrates on protocols that are executed once, in isolation. At the end of this section we briefly review some basic feasibility results for this "stand-alone security" model.

Section 5 introduces the notion of security-preserving protocol composition. We start by demonstrating, via some examples, the security pitfalls associated with protocol composition. We then survey some salient composition operations and scenarios. Finally, we define what it means for a notion of security to provide "composable security" (with respect to some type of composition). The presentation in this section is, for the most part, independent of the material in the previous sections.

Section 6 reviews the composability properties of the "stand-alone" notion of security from Section 4. In a nutshell, it is demonstrated that security is preserved as long as no two protocol instances run concurrently with each other. However, no security is guaranteed as soon as even *two* protocols instances run concurrently.

Section 7 presents and discusses the notion of Universally Composable (UC) security [c01]. The salient feature of this notion is that it guarantees that security is preserved in any composite system, and for any set of protocols running concurrently. After presenting the notion and its relation to the stand-alone notion from Section 4, we briefly review the known feasibility results, as well as some relaxations of this notion that were recently studied in the literature.

Section 8 provides a brief and subjective view of notions of security for cryptographic protocols. The Appendix contains a mini survey of definitional works that follow the trusted-party paradigm.

2 The trusted-party paradigm

This section motivates and sketches the trusted-party definitional paradigm, and highlights some of its main advantages. More detailed descriptions of actual definitions are left to subsequent sections.

Let us consider, as a generic example, the task of two-party secure function evaluation. Here two mutually distrustful parties P_0 and P_1 want to "securely evaluate" some known function f, in the sense that P_i has value x_i and the parties wish to jointly compute $f(x_0, x_1)$ "in a secure way." Which protocols should we consider adequate for this task?

First attempts. Two basic types of requirements come to mind. The first is correctness: the parties that follow the protocol (often called the "good parties" or "honest parties") should output the correct value of the function evaluated at the inputs of all parties. Here the "correct function value" may capture multiple concerns, including authenticity of the identities of the participants, integrity of the input values, correct choice of random values, etc. The second requirement is secrecy, or hiding the local information held by the parties as much as possible.

For instance, consider two parties (say, two databases), each having a list of items, that wish to find out which items appear in both lists. Here, correctness means that the parties output all the entries which appear in both lists, and only those entries. Secrecy means that no party learns anything from the interaction *other than* the joint entries.

However, in general, formalizing these requirements in a meaningful way seems problematic. Let us briefly mention some of the issues. First, defining correctness is complicated by the fact that it is not clear how to define the "input value" that an arbitrarily-behaving party contributes to the computation. In particular, it is of course impossible to "force" such parties to use some value given from above. So, what would be a "legitimate", or "acceptable" process for choosing inputs by parties who do not necessarily follow the protocol?

Another question is how to formulate the secrecy requirement. Here it seems reasonable to require that parties should be able to learn from participating in the protocol nothing more than their prescribed outputs of the computation, namely the "correct" function value. But, even before getting into the more technical question of how to formulate such a "learn nothing more" requirement, we run into the problem that the "correct function value" in itself depends on the inputs contributed by parties who may not follow the protocol.

Let us exemplify some of these issues via the following toy protocol (taken from [MR91]): Assume that $x_0, x_1 \in \{0,1\}$, and that f is the exclusive or function, namely $f(x_0, x_1) = x_0 \oplus x_1$. That is, each party contributes an (a priori secret) input value, and obtains the exclusive or of the two inputs. The protocol instructs P_0 to send its input to P_1; then P_1 announces the result. Intuitively, this protocol is insecure since P_1 can unilaterally determine the output, *after learning P_0's input.* Yet, the protocol maintains secrecy (which holds vacuously for this problem since each party can infer the input of the other party from its own input and the function value), and is certainly "correct" in the sense that the output fits the input that P_1 "contributes" to the computation.

This example seems to indicate that the a secure protocol must guarantee that the input that a party contribute to the protocol should be chosen without knowledge of the inputs of the other parties (at least those who follow the protocol). This, in turn, suggests that the correctness and secrecy requirements are in fact intertwined, namely they are two facets of a single requirement, rather than two different requirements.

The same example also brings forth another security requirement from protocols, in addition to correctness and secrecy: We want to prevent one party from *influencing* the output of the other parties in illegitimate ways, even when plain correctness is not violated.

Additional problems arise when the function to be evaluated is *probabilistic,* namely the parties wish to jointly "sample" from a given distribution that may depend on secret values held by the parties. Here it seems clear that correctness should take the form of some statistical requirement from the output distribution. In particular, each party should be able to influence the output distribution only to the extent that the function allows, namely only in ways that can be interpreted as providing a legitimately determined input to the function. Furthermore, as demonstrated by the following example, the case of probabilistic functions puts forth an additional, implicit secrecy requirement. (We note that this concern arises even in the simplified case where all parties are trusted to follow the protocol instructions and the goal is to prevent illegitimate information leakage.)

Assume that the parties want to toss k coins, where k is a security parameter; formally, the evaluated function is $f(\cdot,\cdot) = r$, where $r \overset{\text{R}}{\leftarrow} \{0,1\}^k$. Let f be a one-way permutation on domain $\{0,1\}^k$ (i.e., given a random k-bit value x, it is infeasible to compute $f^{-1}(x)$). The protocol instructs P_0 to choose $s \overset{\text{R}}{\leftarrow} \{0,1\}^k$ and send $r = f(s)$ to P_1. Both parties output r.

This protocol preserves secrecy vacuously (since the parties do not have any secret inputs), and is also perfectly correct in the sense that the distribution of the joint output is perfectly uniform. However, the protocol lets P_0 hold some "secret trapdoor information" on the joint random string. Furthermore, P_1 does not have this information, and cannot feasibly compute it (assuming that f is one-way). This "quirk" of the protocol is not merely an aesthetic concern. Having such trapdoor information can be devastating for security if the output string r is used within other protocols. This example seems to suggest that a definition of security should somehow specify also the *process* in which the output is to be obtained.

Other concerns, not discussed here, include issues of *fairness* in obtaining the outputs (namely, preventing parties from aborting the computation after they received their outputs but before other parties received theirs), and addressing *break-ins* into parties that occur during the course of the computation.

The trusted party paradigm. The trusted party paradigm follows the "unified requirement" approach mentioned above. The idea (which originates in [GMW87], albeit very informally) proceeds as follows. In order to determine whether a given protocol is secure for some cryptographic task, first envision an ideal process for carrying out the task in a secure way. In the ideal process all parties secretly hand their inputs to an external trusted party who locally computes the outputs according to the specification, and secretly hands each party its prescribed outputs. This ideal process can be regarded as a "formal specification" of the security requirements of the task. (For instance, to capture the above *secure function evaluation* task, the trusted party simply evaluates the function on the inputs provided by the parties, and hands the outputs back to the parties. If the function is probabilistic then the trusted party also makes the necessary random choices.) The protocol is said to securely realize a task if running the protocol amounts to "emulating" the ideal process for the task, in the sense that any damage that can be caused by an adversary interacting with the protocol can also be caused by an adversary in the ideal process for the task.

In principle, this idea seems to have the potential to answer all the concerns discussed above. Indeed, in the ideal process both correctness and lack of influence are guaranteed *in fiat,* since the inputs provided by any adversarial set of parties cannot depend on the inputs provided by the other parties in any way, and furthermore all parties obtain the correct output value according to the specification. Secrecy is also immediately guaranteed, since the only information obtained by any adversarial coalition of parties is the legitimate outputs of the parties in this coalition. In particular, no implicit leakage of side-information correlated with the output is possible. Another attractive property of this approach is its apparent generality: It seems possible to capture the requirements of very different tasks by considering different sets of instructions for the external trusted party.

It remains to substantiate this definitional approach in a way that maintains its intuitive appeal and security guarantees, and at the same time allows for reasonable analysis of "natural" protocols. In this tutorial we describe several formalizations, that differ in their complexity, generality and composability guarantees. Yet, all these formalizations follow the same outline, sketched as follows. The definition proceeds in three steps. First we formalize the process of executing a distributed protocol in the presence of adversarial behavior of some parts of the system. Here the adversarial behavior is embodied via a single, centralized computational entity called the adversary. Next we formalize the ideal process for the task at hand. The formalized ideal process also involves an adversary, but this adversary is rather limited and its influence on the computation is tightly controlled. Finally, we say that a protocol π securely realizes a task $\mathcal{F}$ if for *any* adversary $\mathcal{A}$ that interacts with π there *exists* an adversary $\mathcal{S}$ that interacts with the trusted party for $\mathcal{F}$, such that no "external environment," that gives inputs to the parties and reads their outputs, can tell whether it is interacting with π or with the trusted party for $\mathcal{F}$. (Here the "environment" represents "everything that happens outside the protocol execution," including both the the immediate users of the protocol and other parties and protocols.)

Very informally, the goal of the above requirement is to guarantee that any information gathered by the adversary $\mathcal{A}$ when interacting with π, as well as any "damage" caused by $\mathcal{A}$, could have also been gathered or caused by an adversary $\mathcal{S}$ in the ideal process with $\mathcal{F}$. Now, since the ideal process is designed so that *no* $\mathcal{S}$ can gather information or cause damage more than what is explicitly permitted in the ideal process for $\mathcal{F}$, we can conclude that $\mathcal{A}$ too, when interacting with π, cannot gather information or cause damage more than what is explicitly permitted by $\mathcal{F}$.

We note that the definitional approach of comparing an execution to an idealized system can be viewed as a natural extension of the approach taken when defining semantic security of encryption scheme [GM84] and zero-knowledge proofs [GMRa89]. Furthermore, the formulation described here can be seen as a direct generalization of the formulations in [GM84, GMRa89].

Jumping ahead, we also note that the above formulation has an apparent intuitive "compositionality" guarantee: Since it is explicitly required that no "environment" can tell the protocol from the trusted party, it makes sense to expect that a protocol will exhibit the same properties regardless of the activity in the rest of the system. We postpone additional discussion of this important issue to later sections.

3 Basic security: A simplified case

For the first formalization, we consider a relatively simple setting: As in the previous section, we restrict ourselves to two-party secure function evaluation, namely the case of two parties that wish to jointly compute a function of their inputs. We also restrict ourselves to the "stand-alone" case, where the protocol is executed once, and no other parties and no other protocol executions are considered. Furthermore, we are only concerned with the case where one of the two parties is adversarial. In particular, the communication links are considered "trusted", in the sense that each party receives all the messages sent by the other party, and only those messages. It turns out that this setting, while highly simplified, still captures much of the complexity of the general problem. We thus present it in detail before considering more complex (and more realistic) settings.

Section 3.2 presents the definition. Section 3.3 exemplifies the definition by providing some definitions of cryptographic tasks, cast in this model. First, however, we present the underlying model of distributed computation, in Section 3.1.

3.1 A basic system model

Before defining security of protocols, one should first formulate a model for representing distributed systems and protocols within them. Informally, we wish to capture a system of (resource bounded) computing elements that communicate in an arbitrary asynchronous manner. This section sketches such a model; since we only need to capture two-party protocols, the model is somewhat simplified (it is extended later). Still, readers that are satisfied with a more informal notion of distributed systems, protocols, and polynomial-time computation can safely skip this section.

Several general models for representing and arguing about distributed systems exist, e.g. the CSP model of Hoare [H85], the π-calculus of Milner [M89, M99], or the I/O automata of Lynch and Tuttle [LT89]. Here we build on the interactive Turing machines (ITMs) model, put forth in Goldwasser, Micali and Rackoff [GMRa89] (see also [G01]). Indeed, while the ITM model is more "low level" and provides fewer and less elegant abstraction mechanisms than the above models, it allows for capturing in a natural way the subtle relations between randomization, interaction, and resource-bounded adversarial behavior. Specifically, we formulate a simplified version of the model of [C01, 2005 revision]. (Some models that aim at combining the computational advantages of the ITM model with the analytical advantages of more abstract models include [PW00, PW01, C+06, K06].)

Interactive Turing Machines. Interactive Turing machines (ITMs) are probabilistic Turing machines augmented with mechanisms that allow transferal of data between different machines.

Specifically, an ITM is a Turing machine with some externally writable tapes, namely tapes that can be written into by other machines. It will be convenient to distinguish three externally writable tapes: An input tape, representing inputs provided by the "invoking program", an incoming communication tape, representing messages coming from the network, and a subroutine output tape, representing outputs provided by subroutines invoked by the present program. The input tape represents information coming from "outside the protocol instance", while the incoming communication tape and the subroutine output tapes provide information that is "internal to a protocol instance." In addition, the incoming communication tape models information coming from untrusted sources, while the information on the subroutine output tapes is treated as coming from a trusted source.

Systems of ITMs. The model of computation consists of several instances of ITMs that can write on the externally writable tapes of each other, subject to some global rules. We call an ITM instance an ITI. Different ITIs can run the same code (ITM); however they would, in general, have different local states.

An execution of a systems of ITMs consists of a sequence of activations of ITIs. In each activation, the active ITI proceeds according to its current state and contents of tapes until it enters a special wait state. In order to allow the writing ITI to specify the target ITI we enumerate the ITIs in the system in some arbitrary order, and require that the write instruction specify the numeral of the target ITI. (This addressing mechanism essentially means that each two ITIs in the system have a "direct link" between them. A more general addressing mechanism is described in Section 4.1.) The order of activation is determined as follows: There is a pre-determined ITI, called the initial ITI, which is the first one to be activated. At the end of each activation, the ITI whose tape was written to is activated next. If no external write operation was made then the initial ITI is activated. The execution ends when the initial ITI halts. (To disambiguate the order of activations, we allow an ITI to write on an externally writable tape of at most one other ITI per invocation.)

In principle, the global input of an execution should be the initial inputs of all ITIs. For simplicity, however, we define the global input as the input of the initial ITI alone. Similarly, the output of an execution is the output of the initial ITI. (This formulation will suffice for our purposes.) A final ingredient of a system of ITMs is the control function, which determines which tapes of which ITI can each ITI write on. As we'll see, the control function will be instrumental in defining different notions of security.

Looking ahead, we remark that this very rudimentary model of communication, with its simple and sequential scheduling of events, actually proves sufficient for expressing general synchrony, concurrency, and scheduling concerns.

Polynomial-Time ITMs. In order to model resource-bounded programs and adversaries, we need to define resource-bounded ITMs. We concentrate on polynomial time ITMs. We wish to stick with the traditional interpretation of polynomial time as "polynomial in the length of the input." However, since in our model ITMs can write on the tapes of each other, care should be taken to guarantee that the overall running time of the system remains polynomial in the initial parameters. We thus say that an ITM M is polynomial time (PT) if there exists a polynomial $p(\cdot)$ such that at any point during the computation the overall number of steps taken by M is at most $p(n)$, where n is the overall number of bits written so far into the *input* tape of M, minus the number of bits written by M to the input tapes of other ITIs. This guarantees that a system of communicating ITMs completes in polynomial time in the overall length of inputs, even when ITIs write on the input tapes of each other. (An alternative, somewhat simpler formulation says that the overall running time of an ITM should be polynomial in the value of a "security parameter". However, this formulation considerably limits the expressibility of the model, especially in the case of reactive computation. See [C01] for more discussion on notions of PPT ITMs.)

Protocols. A protocol is defined simply as an ITM. This ITM represents the code to be run by each participant, namely the set of instructions to be carried out upon receipt of an input, incoming message, or subroutine output (namely, output from a subroutine). If the protocol has different instructions for different roles, then the ITM representing the protocol should specify the behaviors of all roles. A protocol is PT if it is PT as an ITM.

3.2 The definition of security

We flesh out the definitional plan from Section 2, for the case of two-party, stand-alone, non-reactive tasks (see Figure 1).

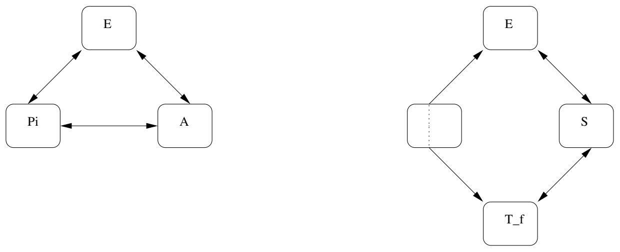

Figure 1: The definition of security at a glance. The left figure depicts an execution of the protocol with an adversary $\mathcal{A}$. The right figure depicts the ideal process for a function f; here a party and the adversary interact via a trusted party $\mathcal{T}_f$. A protocol π securely evaluates a function f if for any adversary $\mathcal{A}$ there is an adversary $\mathcal{S}$ such that no environment can tell with significant probability whether it is interacting with $\mathcal{A}$ and a party running π or with $\mathcal{S}$ in the ideal process for f.

The protocol execution experiment.[1] Let π be a two-party protocol. The protocol execution experiment proceeds as follows. There are three entities (modeled as ITIs): an entity P, that runs the code of π, the adversary, denoted $\mathcal{A}$, and the environment, denoted $\mathcal{E}$.

The environment (who is activated first) provides initial inputs to $\mathcal{A}$ and the party P running π; later, it obtains the final outputs of P and $\mathcal{A}$. (The initial inputs can be thought of as encoded in $\mathcal{E}$'s own input.)

Once either P or $\mathcal{A}$ is activated, with either an input value or an incoming message (i.e., a value written on the incoming communication tape), it runs its code and potentially generates a message to be written on the other party's incoming communication tape, or an output, to be read by $\mathcal{E}$. Both P and $\mathcal{A}$ can generate only a single output value throughout the computation.

The final output of the execution is the output of the environment. As we'll see, it's enough to let this output consist of s single bit.

[1]The presentation below is somewhat informal. Formal description, in terms of a system of ITMs as sketched in the previous section, can be easily inferred. In particular, the various model restrictions are enforced via an appropriate control function.

We use the following notation. Let $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(x)$ denote the random variable describing the output of environment $\mathcal{E}$ when interacting with adversary $\mathcal{A}$ and protocol π on input x (for $\mathcal{E}$). Here the probability is taken over the random choices of all the participating entities. Let $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$ denote the ensemble of distributions $\{\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(x)\}_{x\in\{0,1\}^*}$.

The ideal process. Next an ideal process for two-party function evaluation is formulated. Let $f : (\{0,1\}^*)^2 \to (\{0,1\}^*)^2$ be the (potentially probabilistic) two-party function to be evaluated.

We want to formalize a process where the parties hand their inputs to a trusted entity which evaluates f on the provided inputs and hands each party its prescribed output. For that purpose, we add to the system an additional entity (ITI), denoted $\mathcal{T}_f$, which represents the trusted party and captures the desired functionality. P now runs the following simple ideal protocol for f: When receiving input value, P forwards this input to $\mathcal{T}_f$. When receiving an output from $\mathcal{T}_f$, P forwards this output to $\mathcal{E}$.

$\mathcal{T}_f$ proceeds as follows: It first waits to receive input (b,x) from P and input x' from the adversary $\mathcal{A}$, where $b \in \{1,2\}$ denotes whether x is to be taken as the first or second input to f. Once the inputs are received, $\mathcal{T}_f$ evaluates the function, namely it lets $x_b \leftarrow x$, $x_{3-b} \leftarrow x'$, and $(y_1,y_2) \leftarrow f(x_1,x_2)$. Next, $\mathcal{T}_f$ outputs y_{3-b} to $\mathcal{A}$. Once it receives an ok message from $\mathcal{A}$, $\mathcal{T}_f$ outputs y_b to P.

Analogously to the protocol execution experiment, let $\text{IDEAL}_{f,\mathcal{A},\mathcal{E}}(x)$ denote the random variable describing the output of environment $\mathcal{E}$ when interacting with adversary $\mathcal{A}$ and the ideal protocol for f on input x (for $\mathcal{E}$), where the probability is taken over the random choices of all the participating entities. Let $\text{IDEAL}_{f,\mathcal{A},\mathcal{E}}$ denote the ensemble $\{\text{IDEAL}_{f,\mathcal{A},\mathcal{E}}(x)\}_{x\in\{0,1\}^*}$.

Securely evaluating a function. Essentially, a two-party protocol π is said to securely evaluate a two-party function f if for *any* adversary $\mathcal{A}$, that interacts with π, there *exists* another adversary, denoted $\mathcal{S}$, that interacts with $\mathcal{T}_f$, such that no environment will be able to tell whether it is interacting with π and $\mathcal{A}$, or alternatively with $\mathcal{T}_f$ and $\mathcal{S}$.

To provide a more rigorous definition, we first define indistinguishability of probability ensembles. A function is negligible if it tends to zero faster than any polynomial fraction, when its argument tends to infinity. Two distribution ensembles $\mathcal{X} = \{X_i\}_{a\in\{0,1\}^*}$ and $\mathcal{X}' = \{X'_i\}_{a\in\{0,1\}^*}$ are indistinguishable (denoted $\mathcal{X} \approx \mathcal{X}'$) if for any $a, a' \in \{0,1\}^k$ the statistical distance between distributions X_a and X'_a is a negligible function of k.[2] Secure evaluation is then defined as follows:

Definition 1 (Basic security for two-party function evaluation) *A two-party protocol π* securely evaluates *a two-party function f if for any PT adversary $\mathcal{A}$ there exists a PT adversary $\mathcal{S}$ such that for all PT environments $\mathcal{E}$ that output only one bit:*

$$\text{IDEAL}_{f,\mathcal{S},\mathcal{E}} \approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$$

3.2.1 Discussion

Motivating some choices in the model. Recall that the protocol execution experiment involves only a *single* party running the two-party protocol, where the messages are exchanged with

[2]The use of an asymptotic notion of similarity between distribution ensembles greatly simplifies the presentation and argumentation. However it inevitably introduces some slack in measuring distance. More precise and quantitative notions of similarity may be needed to determine the exact quantitative security of protocols. Also, note that we do not define *computational* indistinguishability of probability ensembles. This is so since we will only be interested in ensembles of distributions over the binary domain $\{0,1\}$, and for these ensembles the two notions are equivalent.

the adversary rather than with another party running the protocol. This models the fact that we consider the behavior of the system when one of the parties follows the protocol while the other follows a potentially different strategy. In two-party protocols where there are two distinct roles there will be two distinct cases depending on the role played by the party who is running the protocol. However, since the role can be modeled as part of the input, this distinction need not be made within the general modeling.

Recall that the environment only sees the inputs and outputs of the adversary and the party running the protocol; it does not have direct access to the communication between the parties. Indeed, the environment captures the "external system" that provides inputs to the parties and obtains their outputs. the communication between the parties is treated as internal to the protocol rather than part of its functionality.

Also, notice that no generality is lost by restricting the environment to output only one bit, since a definition that allows the environment to generate long outputs would end up being equivalent to the present one.

Interpreting the definition. It is instructive to see how the informal description of Section 2 is substantiated. First, the ideal process represents in a straightforward way the intuitive notion of a trusted party that obtains the inputs from the parties and locally computes the desired outputs. In particular, the input provided by the adversary depends only in the information it was initially given from $\mathcal{E}$. Furthermore, $\mathcal{A}$ obtains only the specified function value.

Now, assume there existed an adversary $\mathcal{A}$ that could interact with the protocol and exhibit "bad behavior" that cannot be exhibited in the ideal process, by any adversary $\mathcal{S}$. Then there would exist an environment $\mathcal{E}$ that outputs '1' with significantly different probabilities in the two executions, and the definition would be violated.

The notion of "bad behavior" is interpreted in terms of the joint distribution of the outputs of P and $\mathcal{A}$ on any given input. This interpretation is very broad: For instance, it guarantees that the protocol does not allow the adversary to gather information on the other party's input, where this information is not available in the ideal process (since otherwise the protocol execution would have no ideal-process counterpart). It also guarantees that the protocol does not allow an adversarial party to *influence* the output of the other party in ways that are not possible in the ideal process. In particular, it is guaranteed that the adversary $\mathcal{S}$ in the ideal process is able to generate an "effective adversarial input" x_2 to the trusted party that is consistent with P's input and output (namely, x_2 satisfies $y_1 = f(x1, x2)_1$, where x_1 is P s input and y_1 is P's output).

In addition, the environment can choose to provide $\mathcal{A}$ with input that is either uncorrelated with P's input, or alternatively partially or fully correlated with P's input. This guarantees that the above properties of the protocol hold regardless of how much "partial information" on P's input is already known to the adversary in advance.

Also, notice that the correctness guarantee takes a somewhat different flavor for deterministic and probabilistic functions: For deterministic functions, P's output is guaranteed to be the exact function value, except for negligible probability, *for any potential input value.* For probabilistic functions, it is only guaranteed that the distribution of P's output is *computationally indistinguishable* from the distribution specified by the function. This difference allows the analyst to choose which level of security to require, by specifying an appropriate f.

Yet, the present formulation of the ideal process does not guarantee *fairness:* $\mathcal{A}$ always receives the output first, and can then decide whether P will obtain its output.

Extensions. The definition can be modified in natural ways to require an information-theoretic (or, *statistical*) level of security, by considering computationally unbounded adversaries and environments, or even *perfect* security, by requiring in addition that the two sides of (1) be identical. (To preserve meaningfulness, $\mathcal{S}$ should still be polynomial in the complexity of $\mathcal{A}$, even when $\mathcal{A}$ and $\mathcal{E}$ are unbounded.)

Similarly, the definition can be modified to consider only restricted types of malicious behavior of the parties, by appropriately restricting the adversary. For instance, security against "semi-honest" parties that follow the protocol, but may still try to gather additional information, can be captured by requiring $\mathcal{A}$ to follow the original protocol. (Indeed, in situations where it is plausible to assume that all parties always follow the prescribed protocol such a weaker security guarantee suffices.)

3.3 Examples

To exemplify the use of Definition 1 for capturing the security requirements of cryptographic tasks, we use it to capture the security requirements of three quite different tasks. That is, for each of these tasks we formulate a two-party function that captures the security requirements of the task.

Database Intersection. As a first example, consider the task mentioned in Section 2: Two parties, each having a list of items, wish to find out which items appear in both lists. Here both parties have private inputs and both have private outputs which are different than, but related to, each other. Still, it can be formulated as a function in a straightforward way: $f_{\text{DI}}((x_1^1, ..., x_n^1), (x_1^2, ..., x_m^2)) = ((b_1^1, ..., b_n^1), (b_1^2, ..., b_m^2))$, where $b_j^i = 1$ if x_j^i equals $x_{j'}^{3-i}$ for some j', and $b_j^i = 0$ otherwise. This would mean that a party P which follows the protocol is guaranteed to get a valid answer based on its own database x and *some* database x', where x' was determined by the other party *based only on the initial input of the other party.* Furthermore, the information learned by the other party is computed based on the same two values x and x'. Also, if there is reason to believe that the other party used some concrete "real" database x', then correctness is guaranteed with respect to that specific x'. Recall, however, that the definition does not guarantee fairness. That is, the other party may obtain the output value first, and based on that value decide whether P will obtain its output value. In Section 4 we will see how to express fairness within an extended formalism.

Common Randomness. Next, we consider a task that involves randomness requirements from the outputs of the parties. Specifically, we consider the task of generating a common string that is guaranteed to be taken from a pre-defined distribution, say the uniform distribution over the strings of some length: $f_{\text{CR}}^k(-,-) = (r, r)$, where r is a random k-bit string. Here the parties are guaranteed that the output r is distributed (pseudo)randomly over $\{0,1\}^k$. Furthermore, each party is guaranteed that the other party does not have any "trapdoor information" on r that cannot be efficiently computed from r alone. As mentioned in the Introduction, this guarantee becomes crucial in some cryptographic applications. Finally, as in the previous case, fairness is not guaranteed.

Zero Knowledge. Let $R : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$ be a binary relation, and consider the bivariate function $f_{\text{ZK}}^R((x, w), -) = (-, (x, R(x, w)))$. That is, the first party (the "prover") has input (x, w), while the second party (the "verifier") has empty input. The verifier should learn x plus the one-bit value $R(x, w)$, and nothing else. The prover should learn nothing from the interaction. In particular, when R is the relation associated with an NP language L (that is, $L = L_R \stackrel{\text{def}}{=} \{x|\exists w \text{ s.t. } R(x,w)=1\}$), these requirements are very reminiscent of the requirements from a Zero-Knowledge protocol for L: The verifier is guaranteed that it accepts, or outputs $(x,1)$, only when $x \in L$ (soundness), and the prover is guaranteed that the verifier learns nothing more other than whether $x \in L$ (zero-knowledge).

It is tempting to conclude that a protocol is Zero-Knowledge for language L_R as in [GMRa89] if and only if it securely realizes f^R_{ZK}. This statement is true "in spirit", but some technical caveats exist. Below we discuss these caveats; readers that are satisfied with a more intuitive notion of Zero-Knowledge or are not familiar with its classic definition may safely skip this discussion.

The first caveat is that [GMRa89] define Zero Knowledge so that both parties receive x as input, whereas here the verifier learns x only via the protocol. This difference, however, is only "cosmetic" and can be resolved via simple syntactic transformations between protocols. The remaining two differences are more substantial: First, securely realizing f^R_{ZK} only guarantees "computational soundness", namely soundness against PT adversarial provers. Second, securely realizing f^R_{ZK} implies an additional, somewhat implicit requirement: When the adversary plays the role of a potentially misbehaving prover, the definition requires the simulator to explicitly hand the input *x and the witness w* to the trusted party. To do this, the simulator should be able to "extract" these values from the messages sent by the adversary. This requirement has the flavor of a *proof of knowledge* (see e.g. [G01]), albeit in a slightly milder form that does not require a *black-box* extractor.

In conclusion, we have that a protocol securely realizes f^R_{ZK} if and only if a slight modification of the protocol is a computationally sound Zero-Knowledge Proof of Knowledge for L_R (with potentially non black-box extractors).

4 Basic security: The general case

Section 3 provides a framework for defining security of a restricted class of protocols for a restricted class of tasks: protocols that involve only two parties, and tasks that can be captured as two-party functions. While this case captures much of the essence of the general notion, it lacks in terms of the expressibility and generality of the definitional paradigm.

This section generalizes the treatment of Section 3 in several ways, so as to capture a wider class of cryptographic tasks. First we consider *multi-party* tasks and protocols, namely the case where multiple (even unboundedly many) parties contribute inputs and obtain outputs. This requires capturing various synchrony and scheduling concerns. Second, we consider also *reactive* tasks, where a party provides inputs and obtains outputs multiple times, and new inputs may depend on previously obtained outputs. Next, we let the adversary be a separate entity, rather than taking the place of some of the participants. This allows considering also tasks which require security against "the network", namely against parties that do not take legitimate part in the protocol but may have access to the communication. It also allows expressing situations where parties get "corrupted", or "broken into" in an adaptive way throughout the computation. Next, we allow the adversary interact freely with the trusted party. This allows capturing security requirements in a more fine-grained way by specifying the allowed information leakage and adversarial influence.

Still, throughout this section we only consider the case of a single execution of a protocol, run in isolation. Treatment of systems where multiple protocol executions co-exist is deferred to the next sections.

The necessary extensions to the basic system model are presented first, in Section 4.1. Section 4.2 presents the extensions to the definition of security, while Section 4.3 provides some additional examples. Finally, Section 4.4 briefly reviews some basic feasibility results for this definition.

Overall, this section is somewhat more detail-oriented. While useful for understanding many crucial details in modeling security protocols, it can be safely skipped (or only skimmed) at first reading.

4.1 The system model

In many respects, the system model from Section 3.1 suffices for capturing general multi-party protocols and their security. (In fact, some existing formalisms offer comparable generality, in the sense that they do not include the extensions described below.) Still, that model has some limitations: First, it can only handle a *fixed number* of interacting ITIs. This suffices for protocols where the number of participants is fixed. However, it does not allow modeling protocols where the number of parties can grow in an adaptive way based on the execution of the protocol, or even only as a function of a security parameter. Such situations may indeed occur in real life, say in an on-line auction or gambling application. Another limitation is that the addressing mechanism for external write requests is highly idealized, and does not allow for natural modeling of routing and identity management issues. While this level of abstraction is sufficient for systems with small number of participants that know each other in advance, it does not suffice for open systems, where parties may learn about each other only via the protocol execution.

We thus extend the model of Section 3.1 in two ways (again, following [c01, 2005 revision]). First, we allow for new ITIs to be added to the system during the course of the computation. This is done via a special "invoke new ITI" instruction that can be executed by a currently running ITI. The code of the new ITI should be specified in the invocation instruction. The effect of the instruction is that a new ITI with the specified code is added to the system. The externally writable tapes of the new ITI can now be written to by other ITIs. Note that, given the new formalism, a system of ITMs can now be specified by a single ITM, the initial ITM, along with the control function. All other ITIs in the system can be generated dynamically during the course of the execution. The notion of PT ITMs from Section 3.1 remains valid, in the sense that it is still guaranteed that a system of ITMs is guaranteed to complete each execution in polynomial time, as long as the initial ITM is PT and the control function is polynomially computable.

The second change is to add a special identity tape to the description of an ITM. This tape will be written to once, upon invocation, and will be readable by the ITM itself. This means that the behavior of the ITM can depend on its identity (namely on the contents of its identity tape). Furthermore, an external write instruction will now specify the target ITM via its identity, rather than via a "dedicated link" (represented via some external index).

The identity of an ITI is determined by the ITI that invokes it. To guarantee unambiguous addressing, we require that identities (often dubbed IDs) be unique. That is, an invocation instruction that specifies an existing ID is rejected. (This rule can be implemented, say, by the control function.)

4.2 Definition of Security

We extend the definition of security in several steps. First, we extend the model of protocol execution. Next, we extend the ideal process. Finally, we extend the notion of realizing a trusted party. As we'll see, in some respects the present more general definition is simpler to specify than the one from Section 3.

The protocol execution experiment. We describe the generalized protocol execution experiment. Let π be a protocol to be executed. As before, the model for executing π is parameterized by an environment $\mathcal{E}$ and an adversary $\mathcal{A}$.

Initially, the system consists only of $\mathcal{E}$ and $\mathcal{A}$. During the execution, $\mathcal{E}$ invoke as many parties (ITIs) as it wishes, and determine their identities. All of these parties run π. In addition, $\mathcal{E}$ can write on the input tapes of the parties throughout the computation, and parties can hand outputs to $\mathcal{E}$. In addition, $\mathcal{E}$ can give initial input to $\mathcal{A}$ and can obtain a single (presumable final) output message from $\mathcal{A}$. No other interaction between $\mathcal{E}$ and the system is allowed.

Once a party is activated, either with an input value, or with an incoming message, it follows its code and potentially generates an outgoing message or an output. All outgoing messages are handed to the adversary, regardless of the stated destinations of the messages. Outputs are handed to $\mathcal{E}$. Parties may also invoke new subroutines (ITIs), that may run either π or another code. However, these subroutines are not allowed to directly communicate with $\mathcal{E}$.

Once the adversary is activated, it can deliver a message to a party, i.e. write the message on the party's incoming communication tape. In its last activation it can also generate an output, i.e. write the output value on the incoming communication tape of $\mathcal{E}$.

As before, the final output of the execution is the (one bit) output of the environment. With little chance of confusion, we re-define the notation $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$ to refer to the present modeling.

The ideal process. The main difference from the ideal process in Section 3 is that, instead of considering only trusted parties that perform a restricted set of operations (such as evaluating a function), we let the trusted party run arbitrary code, and in particular to repeatedly interact with the parties, as well as directly with the adversary. We say that the code run by the trusted party is the ideal functionality representing the task.

In addition, the richer system model allows us to simplify the presentation by formulating the ideal process as a special case of the general protocol execution experiment. That is, given an ideal functionality $\mathcal{F}$, we define an ideal protocol $I_{\mathcal{F}}$ as follows: When a party running $I_{\mathcal{F}}$ obtains an input value, it immediately copies this value to the input of $\mathcal{F}$. (The first party to do so will also invoke $\mathcal{F}$.) When a party receives an output from $\mathcal{F}$ (on its subroutine output tape), it immediately outputs this value to $\mathcal{E}$.

The notation $\text{IDEAL}_{\mathcal{F},\mathcal{A},\mathcal{E}}$ from Section 3.2 is no longer needed; it is replaced by $\text{EXEC}_{I_{\mathcal{F}},\mathcal{A},\mathcal{E}}$.

Protocol emulation and secure realization. The notion of realizing an ideal process remains essentially the same. Yet, formalizing the ideal process as an execution of a special type of a protocol allows formalizing the definition of realizing an ideal functionality as a special case of the more general notion of emulating one protocol by another. That is:

Definition 2 (Protocol emulation with basic security) *A protocol π* emulates *protocol ϕ if for any PT adversary $\mathcal{A}$ there exists a PT adversary $\mathcal{S}$ such that for all PT environments $\mathcal{E}$ that output only one bit:*

$$\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}} \approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$$

Definition 3 (Realizing functionalities with basic security) *A protocol π* realizes *an ideal functionality $\mathcal{F}$ if π emulates $I_{\mathcal{F}}$, the ideal protocol for $\mathcal{F}$.*

Secure evaluation vs. observational equivalence. We compare the notion of emulation with the notion of *observational equivalence*, used in the π-calculus formalism of Milner [M89, M99], and elsewhere. (This notion is sometimes called also *bi-simulatability.*) The two notions have somewhat of the same flavor, in the sense that both require that an external environment (or, *context*) will be unable to tell whether it is interacting with one process or with another. (In the work of Milner, the environment is computationally unbounded. A relaxation to the case of computationally bounded environments appears in [LMMS98].) However, emulation is a significantly more lenient notion, since it provides the additional "leeway" of constructing an appropriate simulator $\mathcal{S}$ that will help "fool" the external environment.

In other words, while "process A is observationally equivalent to process B" essentially means that A and B look the same from the outside, "A emulates B" means that A *can be made* to look the same as B by variating *only the adversarial component.*

This extra lenience of the notion of emulation is in fact at the core of what makes it realizable for interesting cryptographic tasks, while maintaining much of the meaningfulness. (Another consequence is that the present notion is not symmetric, whereas observational equivalence is.)

4.2.1 Discussion

Some modeling decisions. We highlight some characteristics of the extended model of protocol execution. First, the present model continues to treat the environment and adversary as *centralized* entities that have global views of the distributed computation. While in the two-party case this was a natural choice, in the multi-party case this modeling becomes an abstraction of reality. This modeling seems instrumental for capturing security in an appropriate way, since we would want security to hold *even when* the adversarial entities do have global view of the computation. Still, it does not allow formulating requirements that relate to a non-centralized ideal adversary.

Another point is the restricted communication between $\mathcal{E}$ and $\mathcal{A}$. Recall that $\mathcal{E}$ cannot directly provide information to $\mathcal{A}$ other than at invocation time, and $\mathcal{A}$ can directly provide information to $\mathcal{E}$ only at the end of its execution. (Of course, $\mathcal{E}$ and $\mathcal{A}$ can exchange information indirectly, via the parties, but this type of exchange is limited by the properties of the specific protocol π in question.) This restriction is indeed natural in a stand-alone setting, since there is no reason to let the adversarial activity against the protocol depend in an artificial way on the local inputs and outputs of the non-corrupted parties. Furthermore, it is very important technically, since it allows proving security of protocols that are intuitively secure, such as the [GMW87] protocol (see Section 4.4).

Also, note that the present modeling of asynchronous scheduling of events, while typical in cryptography, is different than the standard modeling of asynchronous scheduling in general distributed systems, such as those mentioned in Section 3.1. In particular, there asynchrony is typically captured via *non-deterministic* scheduling, where the non-determinism is resolved by an all-powerful scheduler that has access to the entire current state of the system. Here, in contrast, the scheduling is determined by the environment and adversary, namely in an algorithmic and computationally bounded way. This modeling of asynchrony, while admittedly weaker, seems essential for capturing security that holds only against computationally bounded attacks. Combining non-deterministic and adversarial scheduling is an interesting challenge.

Modeling various corruption and communication methods. The simplified model of Section 3 is concentrated on the case where exactly one of the two parties is corrupted. Furthermore, this party is corrupted in advance, before the protocol starts. In contrast, the extended model postulates that all parties follow the specified protocol π; no deviations are allowed. Deviations from the original protocol are captured as additional protocol instructions that "get activated" upon receiving special corruption messages from the adversary. For instance, to capture arbitrary deviation from the protocol, instruct a party to follow the adversary's instructions once it receives a special corrupted message. To capture parties that continue following the protocol but pool all their information together (aka honest-but-curious corruptions, a party that receives a corrupted message will send all its internal state to the adversary, and otherwise continue to follow the protocol. Other types of corruptions can be captured analogously. This way of modeling corruptions has two advantages: First it simplifies the basic model by avoiding the need to explicitly model party corruption, and second it provides flexibility in considering multiple types of corruptions within the same model, and even within the same execution.

The above experiment gives the adversary full control over the communication, thus representing completely asynchronous, unreliable and unauthenticated communication. More abstract communication models, providing various levels of authentication, secrecy, reliability and synchrony, can be captured by appropriately restricting the adversary. (For instance, to model authenticated communication, restrict the adversary to deliver only messages that were previously sent by parties, and include the identity of the source within each message.) In addition, as will be seen in Section 7, all these communication models can be captured as different abstractions within the same basic model, rather than having to re-define the underlying model for each one.

On the generalized modeling of the ideal process. Modeling the trusted party as a general ITM greatly enhances the expressibility of the definitional framework, in terms of the types of concerns and levels of security that can be captured. Indeed, it becomes possible to "fine-tune" the requirements at wish. The down side of this generality is that the exact security implication of a given ideal functionality (or, "code for the trusted party") is not always immediately obvious, and small changes in the formulation often result in substantial changes in the security requirements. One way to address this difficulty, especially when the ideal functionality code is non-trivial, is to explicitly analyze certain key properties of that code (see e.g. [PSW00a, CK02]). Here we very briefly try to highlight some salient aspects of the formalism, as well as useful "programming techniques" for ideal functionalities.

Two obvious aspects of the general formulation are that it is now possible to formulate *multi-party* and *reactive* tasks. In addition, letting the ideal functionality interact directly with the adversary in the ideal process (namely, with the "simulator") has two main effects. First, providing information to the adversary can be naturally used to capture the "allowed leakage of information" by protocols that realize the task. For instance, if some partial information on the output value can be leaked without violating the requirements, then the ideal functionality might explicitly hand this partial information to the adversary. (For instance, to capture the fact that an encryption scheme need not hide th length of the plaintext, simply let the trusted party explicitly give the length of th plaintext to the adversary.)

Receiving information directly from the adversary is useful in capturing the "allowed influence" of the adversary on the computation. For instance, if the timing of a certain output event is allowed to be adversarially controlled (say, within some limits), then the ideal functionality might wait for a trigger from the adversary before generating that output. Alternatively, if several different output values are legitimate for a given set of inputs, the ideal functionality might let the adversary choose the actual output within the given constraints. In some cases it might even be useful to let the adversary hand some arbitrary *code* to be executed by the ideal functionality in a "monitored way," namely subject to constraints set by the ideal functionality.

In either case, since the communication between the ideal functionality and the adversary is not part of the input-output interface of the actual parties, the effect of this communication is always to *relax* the security requirements of the task.

An example of the use of direct communication between the adversary and the ideal functionality is the modeling of the allowable information leakage and adversarial influence upon party corruption. In the ideal process, party corruption is captured via a special message from the adversary to the ideal functionality. In response to that message, the ideal functionality might provide the adversary with appropriate information (such as past inputs and outputs of the corrupted party), allow the adversary to change the contributed input values of the corrupted parties, or even change its behavior in more global ways (say, when the number of corrupted parties exceeds some threshold).

Finally, recall that the ideal functionality receives input directly from the environment, and provides outputs directly to the environment, without intervention of the adversary. This has the effect that an ideal protocol can guarantee delivery of messages, as well as concerns like *fairness,* in the sense that one party obtains output if and only if another party does. In fact, special care should be taken, when writing an ideal functionality, to make sure that the functionality allows the adversary to delay delivery of outputs (say, by waiting for a trigger from the adversary before actually writing to the subroutine output tape of the recipient party); otherwise the specification may be too strong and unrealizable by a distributed protocol.

4.3 More examples

Definition 3 allows capturing the security and correctness requirements of practically any distributed task, in a stand-alone setting. This includes, e.g., all the tasks mentioned in the introduction. Here we sketch ideal functionalities that capture the security requirements of three basic tasks. Each example is intended to highlight different aspects of the formalism.

Commitment. First we formulate an ideal functionality that captures the security requirements from a commitment protocol, as informally sketched in the introduction. Commitment is inherently a *two step process*, namely commitment and opening. Thus it cannot be naturally captured within the formalism of Section 3, in spite of the fact that it is a two-party functionality.

The ideal commitment functionality, $\mathcal{F}_{\text{COM}}$, formalizes the "sealed envelope" intuition in a straightforward way. That is, when receiving from the committer C an input requesting to commit to value x to a receiver R, $\mathcal{F}_{\text{COM}}$ records (x, R) and notifies R and the adversary that C has committed to some value. (Notifying the adversary means that the fact that a commitment took place need not be hidden.) The opening phase is initiated by the committer inputting a request to open the recorded value. In response, $\mathcal{F}_{\text{COM}}$ outputs x to R and the adversary. (Giving x to the adversary means that the opened value can be publicly available.)

In order to correctly handle adaptive corruption of the committer during the course of the execution, $\mathcal{F}_{\text{COM}}$ responds to a request by the adversary to corrupt C by first outputting a corruption output to C, and then revealing the recorded value x to the adversary. In addition, if the `Receipt` value was not yet delivered to R, then $\mathcal{F}_{\text{COM}}$ allows the adversary to modify the committed value. This last stipulation captures the fact that the committed value is fixed only at the end of the commit phase, thus if the committer is corrupted during that phase then the adversary might still be able to modify the committed value. (Corruption of the receiver does not require any move.)

$\mathcal{F}_{\text{COM}}$ is described in Figure 2. For brevity, we use the following terminology: The instruction "send a delayed output x to party P" should be interpreted as "send (x, P) to the adversary; when receiving `ok` from the adversary, output x to P."

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

1. Upon receiving an input `(Commit,`x`)` from party C, record (C, R, x) and generate a delayed output `(Receipt)` to R. Ignore any subsequent `(Commit...)` inputs.
2. Upon receiving an input `(Open)` from C, do: If there is a recorded value x then generate a delayed output `(Open,`x`)` to R. Otherwise, do nothing.
3. Upon receiving a message `(Corrupt,`C`)` from the adversary, output a `Corrupted` value to C, and send x to the adversary. Furthermore, if the adversary now provides a value x', and the `(Receipt)` output was not yet written on R's tape, then change the recorded value to x'.

Figure 2: The Ideal Commitment functionality, $\mathcal{F}_{\text{COM}}$

Realizing $\mathcal{F}_{\text{COM}}$ is a stronger requirement than the basic notions of commitment in the literature (see e.g. [G01]). In particular, this notion requires both "extractability" and "equivocality" for the committed value. These notions (which are left undefined here) become important when using commitment within other protocols; they are discussed in subsequent sections, as well as in [CF01, C01]. Still, $\mathcal{F}_{\text{COM}}$ is realizable by standard constructions, assuming authenticated communication channels.

Key Exchange. Key exchange (KE) is a task where two parties wish to agree on a random value (a "key") that will remain secret from third parties. Typically, the key is then used to encrypt and authenticate the communication between the two parties. Key exchange may seem reminiscent of the coin-tossing task, discussed in Section 3.3. However, it is actually quite different: Essentially, in the case of key-exchange the two parties wish to jointly thwart an external attacker, whereas in coin-tossing the parties wish to protect themselves from *each other.* More precisely, for key-exchange we only care about the fact that the key is random when both parties follow their protocol, whereas in coin-tossing the output should remain random and unpredictable even when one or the parties deviates from the protocol. On the other hand, in key exchange it is crucial that the key remains secret from third parties, whereas in coin-tossing secrecy from third parties is typically not a concern. Furthermore, since key-exchange is usually carried out in a multi-party environment with asynchronous and unauthenticated communication, issues such as precise timing of events and binding of the output key to specific identities become crucial. Thus, modeling of key-exchange naturally involves an interactive interface, as well communicating directly with the adversary.

Functionality $\mathcal{F}_{\text{KE}}$, presented in Figure 3, proceeds as follows. Upon receiving an `(Initiate,`I, R`)` input from some party I (called the *initiator*), $\mathcal{F}_{\text{KE}}$ sends a delayed output `(Initiate,`I`)` to R. Upon receiving the input `(Respond)` from R, $\mathcal{F}_{\text{KE}}$ forwards this input to the adversary. Now, when receiving a value `(Key,`$P, \tilde{k}$`)` from the adversary, $\mathcal{F}_{\text{KE}}$ first verifies that $P \in \{I, R\}$, else P gets no output. If the two peers are currently uncorrupted, then P obtains a truly random and secret key κ for that session. If any of the peers is corrupted then P receives the key $\tilde{k}$ determined by the adversary.

$\mathcal{F}_{\text{KE}}$ attempts to make only a minimal set of requirements from a candidate protocol. In particular, it attempts o allow the adversary maximum flexibility in determining the order in which the parties obtain their outputs. Also, the fact that there is no requirement on the key when one of the parties is corrupted is captured by allowing the adversary to determine the key in this case. Still,

Functionality $\mathcal{F}_{\mathrm{KE}}$

1. Upon receiving an input $(\texttt{Initiate}, I, R)$ from party I, send a delayed output $(\texttt{Initiate}, I)$ to R. Upon receiving $(\texttt{Respond})$ from party R, send $(\texttt{Respond})$ to the adversary.
2. Upon receiving a message $(\texttt{Corrupt}, P)$ from the adversary, for $P \in \{I, R\}$, mark P as corrupted and output $(\texttt{Corrupted})$ to P.
3. Upon receiving a message $(\texttt{Key}, P, \tilde{k})$ from the adversary, for $P \in \{I, R\}$ do:
   (a) If there is no recorded key κ then choose $\kappa \overset{\mathrm{R}}{\leftarrow} \{0,1\}^k$ and record κ.
   (b) If neither I nor R are corrupted then output $(\texttt{Key}, \kappa)$ to P. Else, output $(\texttt{Key}, \tilde{k})$ to P.

Figure 3: The Key Exchange functionality, $\mathcal{F}_{\mathrm{KE}}$

$\mathcal{F}_{\mathrm{KE}}$ guarantees that if two uncorrupted parties locally obtain a key, then they obtain the same value, and this value is uniformly generated and independent from the adversary's view.

Key Exchange is impossible to realize without some form of authentication set-up, say pre-shared keys, authentication servers, or public-key infrastructure. Still, the formulation of $\mathcal{F}_{\mathrm{KE}}$ is agnostic to the particular set-up in use. It only specifies the desired overall functionality. In each of these cases, $\mathcal{F}_{\mathrm{KE}}$ is realizable by standard protocols, both with respect to basic security and with respect to UC security, discussed in Section 7.

Byzantine Agreement. Next we formulate an ideal functionality that captures (one variant of) the Byzantine Agreement task. Here each party has binary input, and the parties wish to output a common value with the only restriction that if all parties have the same input value then they output that value. The functionality, $\mathcal{F}_{\mathrm{BA}}$, is presented in Figure 4. Let us highlight some aspects of its formulation. First, the number of parties (which is a parameter to $\mathcal{F}_{\mathrm{BA}}$) can depend on the environment. Also the identities of the participants can be determined adaptively as they join the protocol. Second, the fact that the adversary is notified on any new input captures the fact that secrecy of the inputs of the parties is not guaranteed. Third, $\mathcal{F}_{\mathrm{BA}}$ allows the output value to take any adversarially chosen value, unless all parties have the same input. (In particular, the parties are not guaranteed to compute any pre-determined function of their inputs.) Four, $\mathcal{F}_{\mathrm{BA}}$ captures a *blocking* primitive, namely no party obtains output unless all parties provide inputs. It also guarantees *fair* output delivery: As soon as one party obtains its output, all parties who ask for their output receive it without delay. (Note that if $\mathcal{F}_{\mathrm{BA}}$ would have simply sent the outputs to all parties, then fairness would not have been guaranteed since the adversary could have prevented the delivery to some parties by not returning control to $\mathcal{F}_{\mathrm{BA}}$.) Five, while $\mathcal{F}_{\mathrm{BA}}$ does not restrict the identities of participants, the output of each participant includes the set of all participants. Alternatively, $\mathcal{F}_{\mathrm{BA}}$ could allow a party to become a participant only if it satisfies some criteria. Finally, $\mathcal{F}_{\mathrm{BA}}$ does not have a postulation for the case of party corruption. This captures the fact that corrupting a party should give no advantage to the adversary.

Note that $\mathcal{F}_{\mathrm{BA}}$ is agnostic to the specific model of computation in which it is realized. Naturally, realizing $\mathcal{F}_{\mathrm{BA}}$ requires different techniques in different settings (depending e.g. on the level of synchrony and the specific authentication set-up). We conjecture that, in each such setting, realizing $\mathcal{F}_{\mathrm{BA}}$ is essentially equivalent to the standard definition of the primitive in that model. (In particular, it is easy to see that if half or more of the parties are corrupted then $\mathcal{F}_{\mathrm{BA}}$ becomes unrealizable in *any* computational model. Indeed, in such settings the *Byzantine Broadcast* formulation, where

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

$\mathcal{F}_{\text{BA}}$ proceeds as follows, when parameterized by the number n of participants. A set $\mathcal{P}$ of participant identities is initialized to empty. Then:

1. Upon receiving input `(Input,`v`)` from some new party $P \notin \mathcal{P}$, where $v \in \{0,1\}$, add P to $\mathcal{P}$, set $x_P = v$, and send a message `(Input,`P, v`)` to the adversary. As soon as $|\mathcal{P}| = n$, ignore additional `(Input...)` inputs.
2. Upon receiving input `(Output)` from a party $P \in \mathcal{P}$, if $|\mathcal{P}| < n$ then do nothing. Else:
   (a) If the output value y is not yet determined then determine y as follows: If there exists a value $b \in \{0,1\}$ such that $x_P = b$ for all parties $P \in \mathcal{P}$, then set $y = b$. Else, obtain a value y from the adversary.
   (b) Output $(\mathcal{P}, y)$ to P.

Figure 4: The Byzantine Agreement functionality, $\mathcal{F}_{\text{BA}}$

only one party has input, is preferable.)

4.4 Feasibility

We very briefly mention some of the basic feasibility results for cryptographic protocols, which establish a remarkable fact: Practically any cryptographic task can be realized, in principle, by a polynomial-time interactive protocol.

The first work to provide a general feasibility result is Yao [Y86], which showed how to securely evaluate any two-party function by a two-party protocol, in a setting which corresponds to that of Section 3, in the case of "honest-but-curious corruptions" where even corrupted parties continue to follow the protocol.

The basic idea is as follows. Given a function f, first have one party, X, with input x, prepare a binary circuit C_x^f such that for any y, $C_x^f(y) = f(x,y)$. Then X sends to the other party, Y, an "obfuscated version" of C_x^f, so that Y can only evaluate C_x^f on a single input of its choice, without learning any additional information on the "internals" of C_x^f. The obfuscation method involves preparing a "garbled version" of each gate in the circuit, plus allowing Y to obtain a matching "garbled version" of one of the possible two values of each input line. Given this information, Y will be able to evaluate the circuit in a gate by gate fashion, and obtain a "garbled version" of the output line of the circuit. Finally, X will send Y a table that maps each possible garbled value of the output line to the corresponding real value.

Goldreich, Micali and Wigderson [GMW87] generalize [Y86] in two main respects. First, they generalize Yao's "obfuscated circuit" technique to *multi-party* functions. Here all parties participate in evaluating the "garbled gates". Further generalization to reactive functionalities can be done in a straightforward way, as demonstrated in [CLOS02].

Perhaps more importantly, [GMW87] generalize Yao's paradigm to handle also *Byzantine* corruptions, where corrupted parties may deviate from the protocol in arbitrary ways. This is done via a generic and powerful application of Zero-Knowledge protocols. A somewhat over-simplified description of the idea follows: In order to obtain a protocol π that realizes some task for Byzantine corruptions, first design a protocol π' that realizes the task for honest-but-curious corruptions. Now, in protocol π each party P runs the code of π', and in addition, along with each message m

sent by π', P sends a Zero-Knowledge proof that the message m was computed correctly, according to π', based on *some* secret input and the (publicly available) messages that P received. The protocols of [GMW87] withstand any number of faults, without providing *fairness* in output generation. Fairness is guaranteed only if the corrupted parties are a minority.

Ben-Or, Goldwasser and Wigderson [BGW88] demonstrate, using algebraic techniques, that if the parties are equipped with ideally secret pairwise communication channels, then it is possible to securely evaluate any multi-party function in a *perfect way* (see discussion following Definition 1), in the presence of honest-but-curious corruption of any minority of the parties. (A similar result, with statistical rather than perfect security, is given by Chaum, Crepeau and Damgaard [CCD88].) The same holds even for Byzantine corruptions, as long as less only less than a *third* of the parties are corrupted. Rabin and Ben Or [RB89] showed how to withstand any dishonest minority in the above model, assuming a broadcast channel, and at the price of allowing *statistical* security. These bounds are tight. A nice feature of the [BGW88, RB89] protocols is that, in contrast to the [GMW87] protocols, they are secure even against adaptive corruptions. Security against adaptive corruptions without ideally secure communication channels can be obtained by combining these protocols with adaptively secure encryption protocols such as [BH92, CFGN96].

All the above results assume ideally authenticated communication. If an authenticated set-up stage is allowed, then obtaining authenticated communication is simple, say by digitally signing each message relative to pre-distributed verification keys. When no authenticated set-up is available, however, then no task that requires some form of authentication of the participants can be realized. Still, as shown in Barak et.al. [B$^+$05], an "unauthenticated variant" of any cryptographic task can still be realized, much in the spirit of [Y86, GMW87], even without any authenticated set-up. Interestingly, the proof of this result uses in an essential way protocols that are *securely composable,* namely retain their security properties even when running together in the same system. This can be seen as a demonstration of the fact that secure composability, discussed next, is in fact a very basic security requirement for cryptographic protocols.

5 Protocol composition

So far, we have only considered security in a setting where the protocol in question is executed once, in isolation. This setting is indeed appropriate as a first one to consider when the goal is to understand the basic security properties of a protocol. However, analyzing security of a protocol in this stand-alone setting does not allow discovering potential weaknesses that come to play when the protocol runs alongside other protocols, or even alongside other executions of the same protocol. Consequently, so far the only method we have for analyzing security of some system is to model the entire system as a single protocol and analyze it as an atomic unit.

Analyzing security of systems in this way is challenging even for modest-size systems. When considering security of modern, multi-party, complex systems, the above one-shot approach becomes completely impractical. Furthermore, in open systems (such as today's Internet) whose makeup may change dynamically, and arbitrary new protocols might be added after the time of analysis, the above notion does not provide an adequate security guarantee to begin with.

Instead, we would like to be able to carve out pieces of a large system, analyze the security of each piece as if it were stand-alone, and then use the security of the individual pieces to deduce security properties of the overall system. Furthermore, this should be doable even when the overall system is not fully known at the time of analysis. To do that, we need to be able to argue about the behavior and security of protocols when running alongside, or composed with, other protocols.

It turns out that this is a non-trivial task.

This section provides an introduction to the security issues associated with protocol composition. We start (in Section 5.1) with some examples that demonstrate various ways in which security properties might fail to hold when composing together protocols, even when the composed protocols guarantee these properties when run in isolation. We then proceed (in Section 5.2) to provide a brief taxonomy of the main types of composition operations considered in the literature. Finally, we motivate and present the concept of security-preserving composition (Section 5.3).

5.1 What might go wrong

To get some feel for the potential security pitfalls in protocol composition, we sketch three examples that demonstrate different ways in which protocols that are arguably secure in a stand-alone setting become insecure when run in conjunction with other protocols. In all the examples the problem is the same: The attacker uses information learned in one execution to "break" the security of another execution. In each example, this attack takes on a different form. The presentation is very informal throughout this section; indeed, the problems discussed are basic ones, and do not depend on the details of a specific definition of security.

Key Exchange and Secure Communication. This example demonstrates how two protocols can interact badly in settings where the parties uses *secret local outputs* obtained from one protocol as input for the other. It highlights the subtleties involved in maintaining overall security of a system that is designed in a modular way and consists of different interacting protocols.

Consider the task of Key Exchange, discussed in Section 4.3. Recall that here two parties, an initiator I and a responder R wish to jointly generate a key that remains unknown to an external adversary. This key is typically used in order to encrypt and authenticate messages between I and R. Let π be key-exchange protocol that's proven to be secure in a stand-alone setting (say, with unauthenticated communication), and consider the protocol π' that's identical to π except that the following instruction is added to the code of I and R: "If the key has already been generated, and the incoming message includes the correct value of the key, then send a message `yes`. Else send `no`."

We first claim that, in a stand-alone setting, π' is just as secure as π. Indeed, since π is a secure protocol, then certainly it does not instruct any party to send the generated key in the clear. Furthermore, the adversary will be unable to figure out the value of the key just by interacting with the protocol. Thus, the added instruction will never be activated (except perhaps with negligible probability), and π' is effectively identical to π.

On the other hand, consider a setting where π runs in conjunction with a protocol that uses the key to encrypt messages. Furthermore, assume that the message takes one out of two possible values (say, either "sell" or "buy"), and furthermore that the encryption scheme in use is one-time-pad. That is, the encryption protocol obtains the key k from π', and has one party (say, I) send a ciphertext c which is either $k \oplus$ "*sell*" or $k \oplus$ "*buy*". (Here $\oplus$ stands for bitwise exclusive or.) We claim that now an adversary can use c in order to find out both k and the plaintext. In fact, all the adversary has to do is to compute $c' = c \oplus$ "*buy*" and send it to the other party as a message of π'. Now, if the encrypted message was "buy", then $c' = c \oplus$ "*buy*" $= k \oplus$ "*buy*" $\oplus$ "*buy*" $= k$ and R will respond with `yes`. If the encrypted message was "sell", then R will respond with `no`.

The point of this example (which is a variant of an observation of Rackoff from '95), is that π' allows the attacker to use the legitimate parties as "oracles" for testing guesses regarding the value of the key. As long as the system runs only π', and the key is never *used,* this "weakness" has no effect. However, as soon as the key is used and some values of the key become more plausible than others, the weakness becomes devastating. Finally, we remark that some prominent definitions of security for key-exchange in the literature (e.g., that of [BR93]) do not rule out this deceivingly simple weakness.

Parallel composition of Zero-Knowledge protocols. This example shows how certain protocols may be secure when run in a stand-alone setting, but lose their basic security properties as soon as even *two* instances of the *same* protocol are executed concurrently in the same system. This holds even if the system involves no other protocols. (Examples of a similar nature are given in [LLR02] for authenticated Byzantine Agreement protocols, and in [KLR06]. An interesting aspect of the [KLR06] example is that it remains valid even when all parties are computationally unbounded.)

Recall the task of Zero-Knowledge (ZK), discussed in Section 3.3. Here we have a public binary relation R. The prover P transmits a value x to a verifier V, and in addition wants to convince V that it (P) has a secret "witness" w such that $R(x, w)$ holds. This should be done so that V learns nothing more than the fact that P has such a witness.

The example is essentially the one in [GK89, F91]. It uses a combinatorial gadget, which we describe here only very informally. Assume we have a "puzzle system" where both the prover and the verifier can generate puzzles p that have the following properties. First, the prover can solve any given puzzle. Second, the verifier cannot feasibly solve puzzles; in fact, the verifier cannot even verify the validity of a solution. That is, even for puzzles generated by the verifier, the verifier cannot distinguish between a valid solution or a random, invalid one. (Such a gadget can be shown to exist, either via allowing the prover to be computationally unbounded, as in [GK89], or based on some trapdoor information held by the prover, as in [F91].)

Now, let π be a ZK protocol (for some relation R). Construct the protocol π' where the parties first run π, and then continue with the following interaction. First, P sends a random puzzle p to V. Then, V responds with a purported solution s for p, plus a puzzle p'. If s is a correct solution, then P reveals the secret witness w. Otherwise, P sends to V a solution s' for the puzzle p' provided by V.

We first argue that if π is ZK in a stand-alone setting, then π' satisfies the ZK requirement. Intuitively, this holds since, by assumption, V cannot solve puzzles, thus in a stand-alone execution of π P never reveals w (except perhaps with negligible probability). Furthermore, the fact that P provides V with a solution s' to the puzzle p' is not really a problem in a stand-alone setting, since V cannot distinguish s' from a random value (which V could have generated by itself).

However, when a prover P runs two concurrent executions of π' with V (say, on the same input (x, w)), then a cheating V can easily extract the witness: V first waits to receive the puzzles p_1 and p_2 from P in the two sessions. It then sends (s, p_2) to P in the first session, for some arbitrary s. In response, V gets from P a solution s_2 to p_2, which it returns to P in the second session. Since s_2 is a correct solution, P will now disclose w.

Malleability of commitment. The following example highlight two issues. First, it demonstrates that a multi-execution system brings forth entirely new *security concerns* that do not exist in a stand-alone setting. Second, it highlights the difficulty in arguing security of a protocol with respect to *arbitrary* other protocols, especially protocols that have been designed specifically so as to "interact badly" with the analyzed protocol.

Recall the task of commitment, discussed in Section 4.3. This is a two-stage task, where in the commit stage a committer C provides a receiver R with a "commitment value" c to a secret value x. In the opening stage C discloses x. (For simplicity, we assume here that both stages consist of a single message from C to R.) There are essentially two security requirements: A secrecy requirement, that x remains completely secret throughout the commit stage, and a binding requirement, that there is at most one value x that R will accept as a valid opening for a commitment value c.

Consider the following natural sealed-bid auction protocol: Each party commits to its bid (say, over a broadcast channel). Once the bidding stage is over, all parties open their commitments and the winner is decided. It is tempting to deduce that any secure commitment protocol would suffice here. It turns out, however, that there exist natural commitment protocols that satisfy both secrecy and binding (and in fact satisfy the definition from Section 4.3), but which are susceptible to the following attack: An attacker might use a commitment c, that was generated by an honest committer C that commits to a value x, to generate a commitment c'; later, when C opens c to value x, the attacker is able to "open" c' to a value x' that is related to x (say, $x' = x + 1$).[3] Of course, this attack is devastating for the auction protocol, in spite of the fact that neither secrecy nor binding of the commitment protocol were violated here. Rather, a new concern arises, namely the need to maintain "independence" between the committed values in different executions of the protocol. This concern (which is called non-malleability in the literature, following [DDN00] who pointed out this concern and showed how to address it) does not come to play in a stand-alone execution.

Several non-malleable commitment schemes have been constructed, using different set-up and network assumptions. Indeed, these schemes are not susceptible to the above attack. However, notice that this attack captures only a limited aspect of the "independence" problem, where there are only *two* executions, and more importantly the executions are of the *same protocol.* What about independence between an execution of a commitment protocol π and an execution of another protocol, π'? This seems like a hopeless goal, especially when π' is designed specifically to interact with π. To see this, consider the following example. Let π be any (even non-malleable) commitment protocol, and let π' be the protocol where in order to commit to a value x, one runs π on committed value $x - 1$. Assume that C commits using protocol π, and that a malicious C' announces that it commits using protocol π'. Now, when C sends its commitment string c, all C' has to do is to copy c as its own commitment. When C opens c to a value x, C' can use the same opening to open c to the value $x + 1$. Note that C' can use π' in a completely different context, say with a set of parties that do not know about C or π. This will make the attack hard to detect.

Indeed, guaranteeing security against these "chosen protocol attacks" seems intuitively impossible. However, contrary to this intuition, Section 7 demonstrates that such attacks *can* be protected against in most cases, via appropriate use of some set-up assumptions.

[3]For instance, consider Pedersen's commitment scheme [P91]: Let G be an algebraic group of large prime order, and assume that two random generators g, h of G are publicly known (say, they are announced by the auctioneer). In the commit stage, C sends $c = g^x \cdot h^r$, where $x \in G$ is the committed value, and $r \overset{R}{\leftarrow} G$. To open, C sends x and r and R accepts if $c = g^x \cdot h^r$. Here secrecy is perfect (and unconditional). Binding holds under the assumption that computing discrete logarithms in G is infeasible. In fact, a somewhat augmented variant also realizes $\mathcal{F}_{\text{COM}}$ as in Definition 3. Still, consider a malicious committer C' that wishes to commit to the value committed by C, plus one. Then all C' has to do is to generate $c' = c \cdot g$. When C' sees a valid opening (x, r) of c, it can generate the valid opening $(x + 1, r)$ of c'.

5.2 How can protocols be composed

This section provides a brief taxonomy of the different types of protocol composition operations considered in the literature, namely the various ways of combining together protocols in a single system. Taking another point of view, these operations naturally correspond to different ways of de-composing a complex system into separate pieces, which we would like to view as individual "protocols."

We first list some salient parameters for composition operations. Next we discuss some well-studied settings in terms of these parameters. Finally, we show how all these settings can be cast as special cases of a single, general composition operation.

Timing coordination: This parameter refers to the possible ways in which the messages of the individual executions can interleave with each other. Salient options include:

Sequential composition: Here no two messages of different protocol executions may interleave. That is, when ordering the events of sending and receiving of messages in the system along a common time axis, then all the events related to each protocol execution must form an uninterrupted sequence.

Enforcing global sequentiality requires each party to locally coordinate the different executions in terms of the timing of message sending. It also requires some level of global coordination among the parties, to guarantee that no party "gets ahead of the pack" and starts sending messages of a new execution before other parties completed prior executions.

Non-concurrent composition: This is a somewhat more general variant that allows "nesting" of protocol executions, as long as there is no "interleaving" of messages. That is, assume some message of execution e_1 was delivered, and at a later point a message of execution e_2 was delivered. Then, once another message of execution e_1 is delivered, messages of execution e_2 can no longer be delivered. Also here, guaranteeing global non-interleaving requires global coordination.

Parallel composition: Here it is assumed that the messages in each protocol execution are naturally associated with "rounds", where a "round i message" is sent only in response to receiving a "round $i-1$ message". The composed execution of a given set of protocol executions allows any interleaving of protocol messages, as long as all the "round i messages" of all the executions are delivered before any "round $i+1$ message" is delivered.

While this composition method is also quite restrictive and requires global timing coordination among the executions, it is natural in synchronous systems where messages are naturally associated with rounds.

Concurrent composition: Here any interleaving of messages from different protocol executions is allowed. Clearly, concurrent composition allows both sequential and parallel composition as special cases. It also allows many other special types of interleaving, such as the common case where various executions wait for an external global event to proceed. Concurrent composition is very powerful in that it requires no timing coordination among the various executions. Indeed, the timing of events may of course be adversarially coordinated.

We note that the level of timing coordination between executions is in principle unrelated to the synchrony guarantees of the underlying communication network. For instance, different executions can be composed concurrently and "asynchronously" even when each execution is synchronous within itself. Also, sequential or non-concurrent composition can be sometimes guaranteed even in a completely asynchronous communication network.

Input coordination: This parameter refers to the possible relations between the input values to the various protocol executions. We distinguish three variants:

Same input: Here each party has the same input value for all the executions. Taking the role of a party in a protocol as part of its input, this means that each party has the same role in all the executions it participates in. Still, different executions may include different parties. (A somewhat more restrictive case is where the same set of parties participate in all executions.)

Fixed inputs: Here the inputs to different executions can be arbitrarily different from each other. In particular, a party may have different roles in different executions. (For instance a party may be a receiver in one execution of a commitment protocol, and a committer in a different execution.) Still, all inputs, including the set of participants in each execution, are fixed in advance before the execution of the composed system starts.

Adaptively chosen inputs: Here each input to each party in each execution can be determined adaptively based on the current state of the composed system. This is of course the most general setting of this parameter, and includes the above two settings as special cases. Variants of this setting depend on the amount of information available to the entities that choose the inputs; for instance, the inputs of a given party may be determined only based on the information available to that party, or alternatively based on the current global state of the system.

Protocol coordination: This parameter refers to the possible relations between the *programs,* or *codes,* executed in different executions. We distinguish two main cases:

Self composition: Here all executions run the same program. A closely related case is where different executions may run different programs, but the set of programs is fixed and known in advance. (Indeed, running a fixed number of programs is equivalent to running a single program that multiplexes between the many programs depending on the input.)

General composition: Here a given execution of a protocol may be running alongside arbitrary other protocols (i.e., programs) that may not be known in advance. Furthermore, these programs may be determined adaptively, depending on the protocol in question and potentially even on the current state of the composed system. This is indeed a highly adversarial setting. Still, it seems to adequately model the situation in open and unregulated networks such as the global Internet.

State coordination: This parameter refers to the amount and type of information that is shared among different executions. We distinguish the following cases:

Independent states: This is the "classic" case of protocol composition where different executions have no shared state. That is, The local variables of each execution within each participant are seen only by that execution. Also, the random choices made within each execution are independent from those in other executions. (Of course, different executions can still have related inputs.)

Joint state: Here some variables or random choices may be visible to multiple protocol executions. One salient example of such a setting is a protocol where the same secret signing key for a signature scheme is used in multiple protocol executions (say, for generating multiple session keys). Another example is a "common random string", namely a public string that is drawn from some distribution and is assumed to be globally available in the system. Here the "joint part" is typically modeled as a "subroutine protocol" that takes input from and provides output to multiple protocol executions. We note that, although this type of composition is somewhat non-traditional, without it it would not be possible to de-compose such systems into smaller components — such as a single exchange of a key in a key-exchange protocol.

Number of executions: This parameter determines the number of protocol executions that run together in the composed system. It is crucial, in the sense that, for most settings of the rest of the parameters and for each i, it is possible to construct protocols that "compose securely" as long as at most i executions run together, but break as soon as the system involves $i + 1$ executions. Three salient settings are:

Fixed number of executions: Here the number of executions is fixed in advance. In particular, it is does not depend on the input, nor on a security parameter.

Bounded number of executions: The maximum number of executions may depend on public information, such as the security parameter or some global input, but is known when designing the protocol. In particular, the complexity of the protocol may depend on this bound.

Unbounded number of executions: The number of executions is chosen adversarially in an adaptive way, and is limited only by the runtime of the adversary. In particular, it may depend on the execution, and remain unknown to all or some of the parties.

Some studied settings. Almost any combination of the above parameters yields a meaningful setting for the study of security-preserving protocol composition. Yet, some settings have been the focus of much dedicated study, both in the context of specific primitives such as key-exchange, zero-knowledge or commitment, and in more general contexts. We briefly mention some of these settings. (For sake of conciseness and brevity, we do not expand here on the specific contributions of the works mentioned below, nor on the notions of security that are obtained in each of these settings.)

Perhaps the simplest setting to consider is that of sequential self-composition with same input. This setting is studied in the context of zero-knowledge in [GO94] and general function evaluation in [B91]. In the case of parallel and concurrent composition, it was demonstrated in Section 5.1 that zero-knowledge is not preserved under same-input self-composition of even *two* executions [GK89, F91]. Still, protocols that remain zero-knowledge in this setting exist [GO94]. This primitive case of concurrent composition is generalized in a number of directions. One direction is that of multiple concurrent instances, while keeping the restriction to same input. Obtaining zero-knowledge in this case, especially when the number of executions is *unbounded* and not known a priori, turns out to be a non-trivial problem that requires new protocol techniques [F91, DNS98, RK99, PRS02].

Another extension is to the case of concurrent self-composition when parties can have different inputs in different executions. The case of *two* copies and *fixed* inputs, studied in [DDN00] and its many follow-up papers, brings about the concern of *malleability*, or *input independence.* Generalizing to adaptively chosen inputs and a bounded number of concurrent instances, or else to fixed inputs and an unbounded number of sessions, requires yet another set of techniques (e.g. [PR03, P04, PR05a, PR05b]), while a general solution for the case of adaptively chosen inputs and an unbounded number of instances requires either some initial set-up [L04] or some relaxation of the notion of security [BS05].

So far, we discussed the case of self-composition. General composition was first studied in the non-concurrent case, where it was shown to preserve some general ideal-model based notions of security for function evaluation [MR91, C00]. Notions of ideal-model based security that are preserved under concurrent general composition were subsequently developed, e.g. [DM00, PW00, PW01, C01, MRST06]. Methods for arguing about composition with joint state were developed in the context of general composition, e.g. [CR03, CDPW07].

Universal composition. Next we describe a single composition operation (namely, a way of combining several protocols into a single protocol) that can be used to express all the settings discussed above. Having such a generic composition operation is convenient in that composability properties proven with respect to this operation apply to all settings. Furthermore, this specific operation seems to closely correspond to the structure of actual protocols. It also meshes nicely with the trusted party paradigm (we'll see this in the next section).

The composition operation, which we call universal composition, is a natural extension of the "subroutine composition" operation on sequential algorithms to distributed protocols. That is, let ρ be a protocol (i.e., a set of instructions for the participants), where the instructions of each party include an instruction to provide input to some "subroutine program," denoted ϕ, as well as instructions on what to do when the subroutine program ϕ generates output. (Using the formalism of Section 3.1, the system contains ITIs running the code ρ, alongside ITIs running the code ϕ; the ITIs running ρ write in the input tapes of ITIs running ϕ, and the ITIs running ϕ write on the subroutine output tapes of ITIs running ρ.)

Let π be another protocol. Then the composed protocol, denoted $\rho^{\pi/\phi}$, is the protocol where the code of each party is the same as that of ρ, with the exception that the instance of ϕ is replaced by an instance of π. That is, each instruction to provide input to ϕ is replaced by an instruction to provide the same input to π, and the instructions to be carried out upon receipt of an output from ϕ are now carried out upon receipt of an output from π. It is stressed that the replacement is done separately within each party running ρ. In particular, an execution of $\rho^{\pi/\phi}$ involves an entire distributed instance of protocol π, where the different parties of this instance exchange messages among themselves.

The case where ρ uses multiple (potentially unboundedly many) instances of ϕ is defined analogously. That is, each instance of ϕ is replaced by an instance of π. It is assumed that protocol ρ has some mechanism to distinguish among the various instances of ϕ; this mechanism remains the same with respect to distinguishing among the instances of π. While in principle there is no need to specify a particular mechanism, for sake of concreteness we assume that ρ associates a unique session identifier (SID) with each instance of ϕ, where the SID is included in all inputs to and outputs from this instance. Then the composed protocol $\rho^{\pi/\phi}$ keeps the same SIDs as in ρ.

Now, the various settings described above for protocol composition can be captured via different codes for the "high-level protocol", ρ. For instance, concurrent self composition with same input is captured by the protocol ρ that simply runs multiple instances of its subroutine ϕ on the same input, and outputs whatever these subroutines output. To capture fixed or adaptively chosen inputs modify ρ accordingly, to obtain the inputs for the various instances in advance or during the course of the execution. General composition is captured by allowing ρ to be arbitrary.

Sequential self composition in a synchronous execution setting is captured by the protocol ρ that runs multiple instances of its subroutine ϕ, one after the other in a sequential way, either with the same input or with different inputs, as may be the case, and outputs whatever these subroutines output. To capture parallel composition, ρ runs all instances of ϕ together and in each round delivers all the current messages of all instances. in lockstep. Non-concurrent general composition allows ρ to be arbitrary, as long as all parties start and end each instance of ϕ at the same global round, and only messages of this instance of ϕ are sent while this instance is active.

Finally, we note that the above description of universal composition treats the protocol ϕ merely as a formal "placeholder" in the description of protocol ρ. Yet, as seen in the next section, protocol ϕ can have a central role in specifying the security properties required from protocol composition.

5.3 Security preserving composition

So far, we have treated the security requirements from cryptographic protocols under composition in an informal way. That is, we have expressed the desire to have protocols that "maintain their security properties" when run alongside other protocols. We have also observed, in Section 5.1, that some desirable security properties may no longer hold in such settings. How can we formalize the security requirements from protocols under composition?

One way, of course, is to list a set of specific properties that we would like to guarantee, and demonstrate that these properties hold. For instance, for protocols that evaluate some function of the inputs of the parties, we can require that *correctness* is preserved, in the sense that in all instances the outputs of the parties agrees with the value of the function at their inputs. If the evaluated function is probabilistic then we can also require that the randomness used in each execution is in some sense "independent" of the randomness used in other executions. We can also require that secrecy of certain values is preserved even in the composed system. (An example of a setting where such a specific requirement is made is that of concurrent zero-knowledge, mentioned above.) An additional specific requirement is that of *input independence*, or *non-malleability*, namely that the outputs of a protocol execution will not depend in "illegitimate ways" on secret inputs to another execution.

However, in the spirit of Section 2, we prefer to make a single, unified security requirement that would imply all of the specific requirements mentioned above, as well as other potential requirements. And, again, in the spirit of Section 2, we use the ideal-model paradigm to do so.

Recall that, by this paradigm, a protocol π is considered a secure implementation for a given task if it behaves in essentially the same way as an ideal protocol ϕ for that task, where the ideal protocol instructs all parties to privately hand their inputs to a trusted party which computes the desired outputs and hands them back to the parties. Furthermore, the requirement "π behaves in essentially the same way as ϕ" is formalized to mean "π emulates ϕ" as in Definition 2. The compositionality requirement we make is analogous: Consider a task that is represented via an ideal protocol ϕ, and let π be a protocol that uses (potentially multiple instances of) ϕ. We say that π implements the task in a composable way with respect to π, if π continues to behave essentially the same when the instances of ϕ are replaced by instances of π. In the language of universal composition and emulation, we want that the protocol $\rho^{\pi/\phi}$ will emulate the original protocol ρ.

Definition 4 *Protocol π emulates an ideal protocol ϕ with ρ-composable security if it holds that $\rho^{\pi/\phi}$ emulates ρ.*

We observe that the notion of composable security indeed guarantees all the compositionality requirements listed above. Indeed, when ρ makes subroutine calls to the various instances of the ideal protocol ϕ, it is guaranteed that each instance of ϕ returns a correct function value, regardless of the activity in the rest of the system. The definition of emulation guarantees that ρ continues to exhibit essentially the same behavior when the instances of ϕ are replaced with instances of π. Similarly, since the trusted parties operate independently of each other, their outputs are computed using independent random choices. Also, the secrecy of data in each individual execution is guaranteed regardless of the rest of the system. Input independence is guaranteed since each party has to explicitly provide its inputs to each instance of ϕ, based only on its legitimate outputs from the various instances of ϕ. Again, the definition of emulation guarantees that ρ continues to exhibit essentially the same behavior when the instances of ϕ are replaced with instances of π.

The above line of reasoning considers a single "calling protocol", ρ. Secure composability with respect to different types of composition operations are captured by considering the corresponding classes of the calling protocol, as described in Section 5.2.

One potential shortcoming of Definition 4 is that the notion of emulation, as defined so far, does not necessarily imply composable security. This means that Definition 4 does not necessarily guarantee that security is preserved under "iterated composition". That is, the fact that π emulates ϕ with ρ-composable security does not necessarily imply that $\rho^{\pi/\phi}$ emulates ρ with ρ'-composable security for an arbitrary ρ' (or even for $\rho' = \rho$). See more discussion on this point in Section 7.

6 The composability properties of basic security

Intuitively, the trusted-party definitional paradigm as formalized in Section 4 appears to be "inherently compositional". In particular, the notion of protocol emulation seems to almost immediately guarantee — at least in spirit — that no external process will be able to distinguish between the emulating protocol and the emulated one. Thus it seems natural to expect that basic security will imply ρ-composable security with respect to *any* polytime protocol ρ. That is, it is natural to expect that if protocol π realizes an ideal functionality $\mathcal{F}$ with basic security (as in Definition 3), then $\rho^{\pi/\phi}$ would emulate ρ for any polytime protocol ρ.

It turns out that this this intuition can indeed be formalized for some types of composition, namely *non-concurrent* general composition. However, as soon as the non-concurrency condition is violated this intuition is incorrect. Details follow.

Recall that in *non-concurrent* composition it is guaranteed that no two protocol instances run concurrently with each other, except for simple nesting (see Section 5.2). More precisely, say that a protocol ρ is non-concurrent if any execution of ρ^{π}, with any subroutine protocol π, has the following property: Order all messages sent in the system along a single time axis, and Let e_1 and e_2 be two protocol executions where the first message of e_1 was sent before the first message of e_2. Then, once the first e_2-message is sent, no e_1-messages are sent until the last e_2 message is delivered. Then we have:

Theorem 5 ([C00]) *Let π and ϕ be protocols such that π emulates ϕ as in Definition 3. Then, π emulates ϕ with ρ-composable security for any* non-concurrent *protocol ρ.*

Proof idea. We very briefly sketch the main idea behind the proof. For simplicity we concentrate on the case where ρ uses only a single instance of ϕ. Since no two instances of ϕ run concurrently, it is straightforward to extend the proof to the case where ρ uses multiple instances of ϕ.

Let $\mathcal{A}$ be an adversary that interacts with parties running ρ^{π}. We need to construct an adversary $\mathcal{A}_\rho$, such that no environment $\mathcal{E}$ will be able to tell whether it is interacting with $\rho^{\pi/phi}$ and $\mathcal{A}$ or with ρ and $\mathcal{A}_\rho$. The idea is to construct $\mathcal{A}_\rho$ in two steps: First "cut out" of $\mathcal{A}$ a real-life adversary, denoted $\mathcal{A}_\pi$, that operates against protocol π as a stand-alone protocol. The fact that π emulates ϕ guarantees that there exist an adversary ("simulator") $\mathcal{A}_\phi$, such that no environment can tell whether it is interacting with π and $\mathcal{A}_\pi$ or with ϕ and $\mathcal{A}_\phi$. Next, construct $\mathcal{A}_\rho$ out of $\mathcal{A}$ and $\mathcal{A}_\phi$.

We sketch the above steps. Essentially, $\mathcal{A}_\pi$ represents the "segment" of $\mathcal{A}$ that interacts with protocol π. That is, $\mathcal{A}_\pi$ expects to receive in its input (coming from the environment $\mathcal{E}$) a configuration of $\mathcal{A}$, and simulates a run of $\mathcal{A}$ starting from this configuration. Once the execution of this instance of π has completed, $\mathcal{A}_\pi$ outputs the current configuration of the simulated $\mathcal{A}$.

Adversary $\mathcal{A}_\rho$ is essentially the adversary $\mathcal{A}$, where the segment that interacts with π is replaced by the simulator $\mathcal{A}_\phi$. That is, $\mathcal{A}_\rho$ starts by invoking a copy of $\mathcal{A}$ and following $\mathcal{A}$'s instructions, up to the point where the first message of π is sent. At this point, $\mathcal{A}$ expects to interact with π, whereas $\mathcal{A}_\rho$ interacts with ϕ. To continue running $\mathcal{A}$, adversary $\mathcal{A}_\rho$ runs $\mathcal{A}_\phi$, with input that describes the *current* state of $\mathcal{A}$. The interaction between $\mathcal{A}_\phi$ and ϕ is emulated by $\mathcal{A}_\rho$, using $\mathcal{A}_\rho$'s own access to ϕ. Recall that the output of $\mathcal{A}_\phi$ is a (simulated) internal state of $\mathcal{A}$ at the completion of protocol π. Once protocol π completes its execution and the parties return to running ρ, adversary $\mathcal{A}_\rho$ returns to running $\mathcal{A}$ (starting from the state in $\mathcal{A}_\phi$'s output) and follows the instructions of $\mathcal{A}$.

The validity of the construction is demonstrated by reduction: Assume that there is an environment $\mathcal{E}$ that distinguishes between an interaction with ρ and $\mathcal{A}_\rho$, and an interaction with $\rho^{\pi/\phi}$ and $\mathcal{A}$. Then one constructs an environment, $\mathcal{E}_\pi$, that distinguishes between an interaction with ϕ and $\mathcal{A}_\phi$, and an interaction with π and $\mathcal{A}_\pi$. Essentially, $\mathcal{E}_\pi$ runs $\mathcal{E}$, where the interaction between $\mathcal{E}$, ρ, and the segment of $\mathcal{A}$ that does not interact with the subroutine, is simulated internally. The interaction with the subroutine (either π or ϕ) and its adversary (either $\mathcal{A}_\pi$ or $\mathcal{A}_\phi$) is taken to be the interaction with the actual external protocol and adversary.

Finally, it is shown that the view of $\mathcal{E}$, when simulated by environment $\mathcal{E}_\pi$ that interacts with adversary $\mathcal{A}_\pi$ and parties running π, is distributed identically to the view of $\mathcal{E}$ that interacts with adversary $\mathcal{A}$ and parties running $\rho^{\pi/\phi}$. Similarly, the view of $\mathcal{E}$, when simulated by environment $\mathcal{E}_\pi$ that interacts with adversary $\mathcal{A}_\phi$ and parties running ϕ, is distributed *identically* to the view of $\mathcal{E}$ that interacts with adversary $\mathcal{A}$ and parties running ρ. (These two equivalences are essentially standard bisimulation arguments from the distributed systems community.) It is stressed that the bisimulation is exact and the distributions over the views are identical. Consequently, the "loss in security" incurred by the theorem is zero.

Basic security under concurrent composition. Can these composability results be extended to concurrent protocol composition? It turns out that the answer is strongly negative. In fact, we have already seen a counter-example: As argued in Section 3.3, the set of protocols that realize f^R_{ZK}, the zero-knowledge function with relation R, roughly corresponds to a class of zero-knowledge protocols for the language L_R. Furthermore, as seen in section 5.1, it is possible to construct zero-knowledge protocols (for any given language) where running even two instances of the protocol in parallel allows the verifier to extract the entire witness. Indeed, this example can be easily extended to come up with a protocol π and a relation R such that π realizes f^R_{ZK}, but $\rho^{\pi/\phi}$ does not emulate $\rho^{f^R_{\text{ZK}}}$ where ρ is the protocol that runs two instances of its subroutine concurrently, on the same input. Similarly, it can be demonstrated that basic security does not guarantee non-malleability. Further discussion on why this is the case appears in the next section.

7 Universally Composable Security

In spite of the intuitive appeal and expressive power of the basic notion of security developed in Sections 3 and 4, we have seen in Section 6 that this notion provides only limited compositionality guarantees: As soon as protocols are allowed to run concurrently — as they often do in actual composed systems — no security guarantees are given. Furthermore, we have seen examples where security breaks down completely.

Universally Composable (UC) security is a strengthening of the basic notion of security, that comes to address the issue of preserving security under concurrent composition. The goal is to have a notion of security that guarantees security under all commonplace types of protocol composition, and in particular the ones described in Section 5.2. This should be done without losing on the intuitive appeal and expressive power, and with as mild as possible additional requirements from protocols.[4] This section is organized as follows. Section 7.1 presents and motivates the notion of UC security and its relation to the basic notion from previous sections. Section 7.3 very briefly presents the known results regarding the realizability of this notion. Finally, Section 7.3.2 touches upon directions for relaxing UC security while retaining some of its security and composability guarantees.

7.1 The definition

Why does the basic definition of security from Sections 3 and 4 fail to guarantee security under concurrent composition? When reviewing the definition in an attempt to answer this question, one notices that the model of protocol execution as defined there allows the environment, which models the "external world", to exchange information with the adversary, which models a coordinated attack against a single protocol execution, only once at the beginning of the execution, where the environment provides information to the adversary, and once at the end, where the adversary provides output to the environment. In a way, this modeling treats an execution of a protocol as an "atomic step," where there is no "information flow" between the protocol execution and the external environment during the protocol execution. (Some protocols may indeed allow the adversary and environment to exchange additional information via the inputs and outputs to the parties, but such exchanges are protocol-dependent and cannot be used in general arguments on the model.)

This modeling is indeed appropriate in a system where only a single protocol execution is active at any given point in time. However, it seems insufficient for capturing the often "circular" information flow among protocol executions that run *concurrently.* In particular, it fails to capture situations such as the ones described in Section 5.1, where an attacker uses information gathered in one execution in order to extract information in another execution, and then uses the extracted information back in the first execution.

UC security is aimed at correcting this shortcoming of the basic definition. The idea is to modify the model of protocol execution so as to allow the environment and the adversary to interact freely throughout the course of the computation. That is, whenever the environment is activated, it is allowed to provide input not only to the parties running the protocol, but also to the adversary.

[4]The term *universally composable security* might be somewhat confusing, given that the term *universal composition* was used to denote a specific composition operation. In particular, several different definitions of security are known to be "universally composable", in the sense that they support a universal composition theorem such as Theorem 7 below. We thus use the acronym "UC security" to refer to the specific notion discussed here. (The duplicate terminology can be somewhat justified by Proposition 8 below, which implies that UC security is in a sense a *minimal* extension of basic security that is preserved under universal composition.)

Similarly, whenever the adversary is activated, it can provide output to the environment. This means that the environment and the adversary can communicate before and after each activation of a party running the protocol; in other words, the "atomic unit" of uninterrupted execution is now a single activation of a party, rather than an entire execution of a protocol. As seen below, this change to the model turns out to suffice for proving general composability. It also changes the set of acceptable protocols in a radical way.

Another, more technical modification of the model from Section 4 is to add more structure to the communication model in order to facilitate the distinction between protocol instances in a composite system. A more detailed description follows.

The system model. We use the system model from Section 4.1, with one change. To facilitate the distinction among different protocol executions in a system, we assume that the identity of each party (i.e., the contents of the identity tape) consists of two fields: a session ID (SID) and a party ID (PID). The SID is used to specify the "session", or "protocol instance" to which the ITI "belongs", and is joint to all the ITIs in a session. The PID distinguishes the ITI from other ITIs in that protocol instance. It can also be used to associate an ITI with a "cluster" of ITIs, such as the cluster of procedures running on a single physical computer. An instance of a protocol π with SID s in a certain configuration of a system is now defined to be the set of ITIs that have code π and SID s.

Remark: The above modeling of the SID is only one out of many possible ways for representing and distinguishing among protocol instances in a composite system. Still, the fact that all ITIs in a protocol instance have the same SID, which is determined by the invoking ITI, seems like a natural choice. In particular, it is easy to realize (say, by letting the party which initiates an instance to determine the SID and communicate it to all other participants). It also often facilitates the design and analysis of protocols, by providing to the participants a common value that is unique to the instance.

The protocol execution experiment. The protocol execution experiment is the same as the one in Section 4.2, with the following two modifications. First, as mentioned above, we allow the environment to provide inputs to the adversary at any time. Similarly, we allow the adversary to provide outputs to the environment at any time.

Second, recall that in the model of Section 4.2 all parties (ITIs) invoked by the environment must run the same protocol (ITM). Furthermore, all the parties were treated as participating in a single protocol instance. In the present model, unless explicitly restricted, the environment can in principle invoke multiple protocol instances, by giving different SIDs to different parties. To keep in the spirit of a single instance, we require that all the parties invoked by the environment participate in the same protocol instance, namely they all have the same SID. (The value of the SID is of course chosen by the environment.)

Analogously to the notation $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$ from Section 4.2, let $\text{UC-EXEC}_{\pi,\mathcal{A},\mathcal{E}}(x)$ denote the random variable describing the output of environment $\mathcal{E}$ when interacting with adversary $\mathcal{A}$ and protocol π on input x (for $\mathcal{E}$) in the present model. $\text{UC-EXEC}_{\pi,\mathcal{A},\mathcal{E}}$ denotes the ensemble $\{\text{UC-EXEC}_{\pi,\mathcal{A},\mathcal{E}}(x)\}_{x\in\{0,1\}^*}$.

Restricting the environment to run only a single protocol instance significantly simplifies the model and the analysis of protocols. On the down side, it comes at the price of some restrictions on the class of protocols which can be composed in a secure way. See more discussion in Section 7.2.

The ideal process. The ideal process remains the same as the one in Section 4.2, with the following exception: We restrict attention to ideal functionalities $\mathcal{F}$ where an instance ignores inputs that do not specify its SID. (Recall that the SID of an instance is determined by the ITI that called this instance for the first time.) Similarly, we assume that $\mathcal{F}$ includes its SID in all of its outputs. We note that this restriction is not essential; its purpose is to simplify the modeling and analysis of protocols.

Protocol emulation. The notion of protocol emulation and realizing functionalities is the same as in Section 4.2, except that it relates to the present execution experiments:

Definition 6 *UC protocol emulation and realization A protocol π* UC-emulates *protocol ϕ if for any PT adversary $\mathcal{A}$ there exists a PT adversary $\mathcal{S}$ such that for all PT environments $\mathcal{E}$ that output only one bit:*

$$\text{UC-EXEC}_{\phi,\mathcal{S},\mathcal{E}} \approx \text{UC-EXEC}_{\pi,\mathcal{A},\mathcal{E}}$$

A protocol π UC-realizes *an ideal functionality $\mathcal{F}$ if π UC-emulates the ideal protocol for $\mathcal{F}$.*

7.2 Composability

The main attraction in UC security is that it guarantees composable security with respect to almost any PT calling protocol. That is, we restrict the way a protocol receives inputs from and provides output to the surrounding system in the following natural way: We assume that the only component of the "subroutine protocol" that receives inputs from the outside and provides outputs to the outside is the "top-level program". More precisely, recall that an ITI P is called a subroutine of an ITI P' if P takes input from P' or provides output to P'; P is a subsidiary of P' if it a subroutine of P' or of a subsidiary of P'. Say that an ITM π is subroutine respecting if any ITI P running the code π has the property that all subsidiaries of P are subroutines only of P or of subsidiaries of P. A protocol is subroutine respecting if it is subroutine respecting as an ITM. We have:

Theorem 7 *Let π and ϕ be subroutine-respecting PT protocols such that π UC-emulates ϕ. Then $\rho^{\pi/\phi}$ UC-emulates ρ for any PT protocol ρ.*

Historical note. Theorem 7 was first proven in [PW00, PW01] for the case where ρ invokes a single instance of the subroutine protocol ϕ. (These proofs are set in their formalism, which has several technical differences from the one presented here.) The case where ρ may invoke an unbounded number of instances of ϕ was first proven in [C01] in a model similar to the one presented here, and subsequently re-proven in a number of different models, e.g. [BPW04, DKMR05, K06, CKLP06].

Proof idea. At high level, the proof of Theorem 7 follows the same steps as the proof of Theorem 5, with the exception that here protocol ρ may call multiple instances of ϕ, where these instances run concurrently. Consequently, the adversary $\mathcal{A}_\rho$ that interacts with protocol ρ concurrently invokes multiple instances of the simulator $\mathcal{A}_\phi$, where each instance of $\mathcal{A}_\phi$ interacts with a single instance of ϕ. In order to be able to carry out the overall interaction with ρ and the environment in a globally consistent manner, $\mathcal{A}_\rho$ uses the fact that each instance of $\mathcal{A}_\phi$ outputs the necessary information *after each activation.* This allows $\mathcal{A}_\rho$ to use information generated in one instance of $\mathcal{A}_\phi$ as input to another instance of $\mathcal{A}_\phi$. (Recall that in the case of basic security $\mathcal{A}_\phi$ is required to generate output only at the end of the execution; such a guarantee would not suffice for the present case.)

As in the proof of Theorem 5, the proof of validity of $\mathcal{A}_\rho$ proceeds by reduction to the validity of $\mathcal{A}_\phi$. The main difference from Theorem 5 is that here there are multiple instances of a subroutine protocol (either ϕ or π), running concurrently. Thus, we need to demonstrate that no environment can tell the difference between the case where all instances of ϕ are replaced by π and the case where none of the instances of ϕ are replaced by π. This is done via a standard hybrid argument, namely by considering multiple hybrid executions where in each execution one more instance of ϕ is replaced by π. An environment that distinguishes between two consecutive instances is now translated into an environment that contradicts the validity of $\mathcal{A}_\phi$. We omit further details.

7.2.1 Discussion

To interpret Theorem 7 recall that, for any given calling protocol ρ, the fact that $\rho^{\pi/\phi}$ UC-emulates ρ implies that replacing the instances of ϕ by instances of π does not change the behavior of ρ with respect to PT adversaries in a noticeable way; in particular, it does not introduce any new vulnerabilities to $\rho^{\pi/\phi}$. Furthermore, recall that any of the composition scenarios mentioned in Section 5.2 (with the exception of joint-state composition, discussed below) can be captured as universal composition with some set of calling protocols. Thus, Theorem 7 guarantees security-preserving composition in any of these scenarios. Some additional aspects of the theorem are discussed next.

Modular protocol analysis. The fact that Theorem 7 puts very few restrictions on the calling protocol ρ makes it conducive to carrying out the plan from the preamble of Section 4.3 in a way that meshes naturally with the structure of common protocols. That is, the theorem allows decomposing protocols to many simple subroutines, analyzing each subroutine separately, and then deducing the security of the overall protocol from the security of the subroutines. In particular, the partitioning to subroutines can be nested in an arbitrary way. This is a powerful methodology, especially given the fact that rigorous analysis of even simple cryptographic protocols tends to be dauntingly complex.

Enabling sound symbolic and automated analysis. Another advantage of Theorem 7 is that it allows to "abstract away" cryptographic imperfections such as computational bounds and error probabilities, while maintaining soundness of the abstractions. This enables applying automated proof tools that require symbolic representations of protocols (as in, say, [DY83]) and cannot directly handle asymptotic modeling and cryptographic imperfections. To do that, first devise functionalities that capture in an ideal way the security properties of the cryptographic primitives (say, encryption in the case of [DY83]) used in the analyzed system. Next, re-write the protocols to be analyzed in a symbolic, non-asymptotic model that corresponds to having access to the devised ideal functionalities. Now, one can apply an automated tool to the symbolic representation of the protocol. Finally, use the UC theorem to deduce that, if the cryptographic protocols in use UC-realize the devised ideal functionalities, then the overall system enjoys the same properties proven for the abstract version. Some works that take this approach include [BJP02, BPW03, CH04, SB+06].

Two things should be kept in mind, however, when taking this approach. First, for the analysis to be of value, one has to make sure that the asserted abstract security properties have meaningful translations to concrete security properties of concrete protocols. Second, the complexity of automated analysis tools grows very rapidly as a function of the number of messages and sessions in the analyzed system (see e.g. the undecidability and NP-completeness results in [EG82, DLMS99]). Consequently, a viable instantiation of the above approach would need to break down protocols to simple subroutines and analyze each subroutine separately as a single session. Here the UC theorem is once again a crucial enabler.

Representing communication models. Another use of Theorem 7 is for modular representation of various communication models within the basic model of computation described above. That is, to capture a given communication model, simply devise an ideal functionality $\mathcal{F}$ that guarantees the abstractions provided by that model. Now, designing protocols in that model is translated to designing protocols that run in the basic model and make calls to $\mathcal{F}$. In order to further simplify the code of $\mathcal{F}$, one can allow for multiple instances of $\mathcal{F}$ to run concurrently, where each instance deals with a single use of the underlying model (say, a single sending of a message in the case of an authenticated communication abstraction). Here we do not necessarily intend to *realize* $\mathcal{F}$ in an algorithmic way; rather, $\mathcal{F}$ merely serves as a functional description of the desired abstraction. Still, in some cases the same ideal functionality can be used both as the basis for a communication model and as a target to be realized by cryptographic protocols. Some communication models that have been captured this way include authenticated communication, secure communication, and synchronous communication (see e.g. [C01, 2005 revision]).

Composition with joint state. The restriction to subroutine-respecting protocols, made in Theorem 7, excludes the case of composition with joint state, namely in the case where parties in two or more protocol instances have access to the same instance of some subroutine program. We currently have two alternative methods to deal with this situation. A first method is to explicitly model the subroutine as an entity that interacts with multiple protocol instances (even arbitrary ones). This in turn requires working with a strong variant of UC security, called *generalized UC*, which allows capturing such subroutines and the protocols that use them. See details in [CDPW07].

A second option is to demonstrate that *all* the protocols that use the joint subroutine do so via an interface that satisfies a certain condition. Essentially, this condition requires that the interface looks like the interface of multiple independent instances of a simpler procedure. In this case, one can again demonstrate a security-preserving composition result similar to Theorem 7. See details in [CR03].

Some equivalent variants. Finally, we note that several variants of Definition 6 turn out to be equivalent to the present formulation. First, allowing the environment to output an arbitrarily long string, or alternatively restricting the environment to deterministic computation do not change the definition. Also, restricting the adversary $\mathcal{A}$ to only forward messages from the environment to the parties and back results in a definition that is equivalent to the present formulation. Similarly, restricting the adversary $\mathcal{S}$ to have only black-box access to $\mathcal{A}$ results in an equivalent definition. Finally, letting the simulator depend on the environment results in an equivalent definition. We remark that most equivalences hold also in other formalisms (see e.g. [PSW00]). However, the last equivalence does not hold in other formalisms, where entities are required to be polynomial in a global security parameter rather than in the length of local inputs [HU05].

7.3 Feasibility and relaxations

We very briefly survey the feasibility results regarding UC-realizing ideal functionalities. As we'll see, in spite of the apparent syntactic similarity with basic security (Section 4.4), UC security is in general a considerably more restrictive notion. In particular, some far-reaching impossibility results exist. Consequently, several relaxations and work-arounds have been proposed. We will briefly survey these as well.

Encryption, signing, and secure communication. We start with some positive results. It turns out that for the basic tasks of encryption, digital signatures, and other tasks associated with secure communication, there are universally composable formulations that are realizable by known and natural protocols. In fact, in some cases the UC definitions are closely related, or even equivalent, to standard definitions (which use some special-purpose formulations).

Two salient examples are the ideal public-key encryption functionality, $\mathcal{F}_{\text{PKE}}$, and the ideal signature functionality, $\mathcal{F}_{\text{SIG}}$, which capture the basic requirements of encryption and signature in an abstract and unconditional way. UC-realizing $\mathcal{F}_{\text{PKE}}$ (for non-adaptive party corruptions) is essentially equivalent to the standard notion of security against chosen ciphertext attacks [DDN00, RS91]. UC-realizing $\mathcal{F}_{\text{SIG}}$ is essentially equivalent to the standard notion of existential unforgeability against chosen message attacks [GMRi88].

Another class of examples are functionalities related to the task of obtaining secure communication. These include the key-exchange functionality from Section 4.3, as well as ideal functionalities capturing authenticated and secure communication sessions, entity authentication, and related tasks. All of these functionalities can be UC-realized by simple and known protocols. For instance, see the modeling of certified mail in [PSW00a] or secure channels in [PW01, CK02]. In addition, both the ISO 9798-3 key-exchange protocol and IKEv2 (the revised key exchange protocol of the IPSEC standard) UC-realize the ideal key-exchange functionality [CK02, CK02a].

General feasibility. Can the general feasibility results for basic security assuming authenticated communication (see Section 4.4) be carried over to UC security? When the majority of the parties are honest (i.e., they are guaranteed to follow the protocol), the answer is positive. In fact, some known protocols for general secure function evaluation turn out to be universally composable. For instance, the [BGW88] protocol (both with and without the simplification of [GRR98]), together with encrypting each message using non-committing encryption [CFGN96], is universally composable as long as less than a third of the parties are corrupted, and authenticated and synchronous communication is available. Using [RB89], any corrupted minority is tolerable. Asynchronous communication can be handled using the techniques of [BCG93, BKR94]. Note that here some of the participants may be "helpers" (e.g., dedicated servers) that have no local inputs or outputs; they only participate in order to let other parties obtain their outputs in a secure way.

However, things are different when honest majority of the parties is not guaranteed, and in particular in the case where only two parties participate in the protocol and either one of the parties may be corrupted. First, one of the most common proof-techniques for cryptographic protocols, namely black-box simulation with rewinding of the adversary, does not in general work in the present framework. The reason for that is that in the present framework the ideal adversary has to interact directly with the environment which cannot be "rewound". (Indeed, it can be argued that the meaningfulness of black-box simulation with rewinding in a concurrent execution setting is questionable.)

Furthermore, in the UC framework many interesting functionalities cannot be realized at all by plain protocols. (A *plain* protocol uses no ideal functionality other than the authenticated communication functionality.) For one, the ideal commitment functionality from Section 4.3 cannot be UC-realized by plain two-party protocols [CF01]. Similar impossibility results hold for the ideal coin tossing functionality, the ideal Zero-Knowledge functionality, and the ideal Oblivious Transfer functionality [C01]. These results extend to unrealizability by plain protocols of almost all "non-trivial" deterministic two-party functions and many probabilistic two-party functions [CKL03], and to impossibility of realizing *any* "ideal commitment functionality", namely any functionality that satisfies the basic correctness, binding and secrecy properties of commitment in a perfect way [DDMRS06]. These results apply also to multi-party extensions of these primitives, whenever the honest parties are not in majority.

Three main approaches for circumventing these impossibility results have been considered in the literature. The first approach is simply to try to formulate more *relaxed* ideal functionalities, that will be easier to realize, but will still capture the security requirements of the desired task. This is a task-specific and delicate endeavor. Some works that take this approach are [CK02, PS05]; a salient characteristic of these relaxations are that security is guaranteed only in a computational sense *even in the ideal process.*

A second approach is to assume that the parties have access to some trusted set-up. A third approach is to *relax* the UC-emulation requirement. These approaches are described in Sections 7.3.1 and 7.3.2, respectively.

7.3.1 Adding set-up assumptions

It turns out that general feasibility can be regained when some trusted set-up is assumed. One such trusted set-up assumption, called the key registration (KR) model, assumes that there exists a trusted "registration authority" where parties can register public keys associated with their identities, while demonstrating that they have access to the corresponding secret keys. (Alternatively, parties can let the authority choose public keys for them; here the corresponding secret keys need not be revealed, even to the "owners" of the public keys.) Then, parties can query the authority for a party identity and obtain the registered public key for that identity. Practically any ideal functionality can be UC-realized by interactive protocols in the key registration model, under standard computational hardness assumptions. Furthermore, the protocols remain secure even in the presence of arbitrary other protocols *that use the same public keys.*

Taking a short detour, it is interesting to compare this set-up assumption to the set-up assumptions needed for guaranteeing authenticated communication. To obtain authenticated communication (namely, to UC-realize an ideal functionality that provides an authenticated communication service), it is necessary and sufficient to have access to an ideal functionality that allows parties to register public keys that will be associated with their identities, *without* having to disclose the secret keys to the registration authority. This set-up is structurally similar to the key registration set-up, except that the trust put in the registration authority is considerably milder.

An alternative set-up assumption, called the common random string (CRS) model, is that all parties have access to a string that is guaranteed to be taken from a predetermined distribution, typically the uniform distribution. Furthermore, it is assumed that the string was "ideally generated" in the sense that no set of participants have any "side information" on the common string (such as the preimage of the string according to some one-way function). This assumption is attractive in that it can be realized by physical processes that minimize the trust that participants need to put in external authorities. Also, it does not require parties to explicitly register before participating in the computation. However, here the general feasibility results are weaker, in the sense that the protocols are not (and, in fact, provably cannot be) shown secure in the presence of arbitrary other protocols that use the same common string. Instead, security is shown only when all protocols that use the common string do so using a very specific interface.

Yet another alternative set-up assumption, called the timing model, is of a somewhat different flavor: It assumes that there is a bound on the delay of messages delivered in the network, as well as on the mutual discrepancy in local time measurements, and that these bounds are known to all parties. Here too it is possible to realize any ideal functionality, under standard hardness assumptions [LPT04].

Historically, general feasibility results were first demonstrated in the CRS model [CLOS02]. The overall structure of that protocol is the same as in [GMW87], as sketched in Section 4.4. The main difference is in the zero-knowledge and coin-tossing components, which are very different. In particular, the new components (based partly on the UC commitment protocol in [CF01]) allow for simulation "without rewinding", using the CRS set-up. Protocols in the KR model again use the same structure. For non-adaptive party corruptions, it was observed that the [CLOS02] protocols can be modified to work in the KR model [BCNP04]. For adaptive party corruptions some new protocols have been developed [CDPW07].

Can we characterize which functionalities are realizable without set-up? or only given authenticated communication? Alternatively, can we characterize the set-up functionalities that suffice for realizing a given task? Some limited answers to the former question, for the case of evaluating a pre-determined function of the parties' inputs, and for the case of functionalities aimed at guaranteeing secure commitment, are known [CKL03, DDMRS06]. Otherwise, these are interesting open questions.

7.3.2 Relaxing UC security

In light of the restrictiveness of UC-emulation, and in particular given the above impossibility results regarding realizing UC-realizing functionalities without initial set-up, it is natural to look for alternative notions of security, that will still provide some general security and composability guarantees while being easier to realize.

This question is highlighted by the fact that UC-emulation appears to be overly strong with respect to the notion of composable security (Definition 4). That is, Theorem 7 states that $\rho^{\pi/\phi}$ UC-emulates ρ, where Definition 4 only requires that $\rho^{\pi/\phi}$ emulates ρ according to the basic notion of emulation, namely Definition 2. On the one hand, this extra strength is useful, in that it guarantees that security is preserved even after multiple applications of the universal composition operation. On the other hand, though, this extra strength raises the question of whether there is a less demanding variant of UC-emulation that would still satisfy Definition 4.

It turns out, however, that the answer to this question is negative. That is, it can be seen that *any* notion of emulation that satisfies Definition 4 with respect to any calling protocol ρ implies UC-emulation. That is:

Proposition 8 *Assume that protocol π emulates protocol ϕ with ρ-composable security for any subroutine-respecting protocol ρ. Then π UC-emulates ϕ.*

The idea here is that an arbitrary calling protocol ρ can essentially mimic any interactive environment $\mathcal{E}$, even in the basic setting where the external environment cannot interact with the adversary during the execution. This holds even though ρ^{π} is only required to emulate ρ^{ϕ} according to the basic notion, since the instructions of ρ can require the adversary to provide "on-line" information in the same way that $\mathcal{E}$ expects to have in the UC modeling. We omit further details.

Proposition 8 can be interpreted as stating that UC-security is in some sense a "minimal" requirement that guarantees both composability and basic security. It also means that the extra strength in the conclusion of Theorem 7 comes without any additional requirements from the protocol. (Some closely related results appear in [L03, L04].)

Still, some relaxed variants of UC-emulation have been proposed and shown to be preserved under universal composition with arbitrary protocols [PS04, BS05, MMY06]. By Proposition 8, these variants necessarily provide security guarantees that are weaker than basic security. Still, the provided guarantees are often meaningful. In addition, it was demonstrated that these notions allow realizing any ideal functionality given only authenticated communication, under general (but stronger than usual) hardness assumptions.

Essentially, the way in which these notions weaken the security requirement is by allowing the "simulator" to run in super-polynomial time T. This means that meaningful security is guaranteed only when the following two conditions are met. First, the ideal functionality should be such that security is guaranteed even against adversaries running in time T. This condition is met by most common formulations of ideal functionalities; in fact, most common formulations provide "perfect" security, even against computationally unbounded ideal-model adversaries.

The second condition is a bit more subtle: Recall that the definition only guarantees that the environment, or the calling protocol, cannot tell whether it is interacting with the emulating protocol π and adversary $\mathcal{A}$ (which may be PT), or with the emulated protocol and adversary $\mathcal{S}$, which may run in time T. Thus, security is meaningful only when the the *calling protocol itself* withstands adversaries that run in time T. To exemplify this point, we note that it is possible to construct protocols that are secure according to this notion, and yet completely "break down" under self-composition of only two instances.

8 Conclusion

This tutorial addressed the challenges associated with rigorously modeling cryptographic protocols and capturing their security properties. Particular stress was put on guaranteeing security in settings where protocols are *composed* with each other in a number of ways. We have reviewed a general definitional approach, the *trusted party paradigm.* We saw two formalizations of this approach: A basic formalization, that is easier to satisfy but provides only limited secure composability guarantees, and a more advanced formalization that is considerably more restrictive in general, but provides very strong secure composability guarantees.

When looking back at the covered material, one thing becomes very clear: It is far from obvious what is "the right" way to capture and formalize security properties of cryptographic protocols. In fact, there probably is no single good way to do so, and different formalisms have incomparable strengths. Furthermore, seemingly small differences in the formalisms result in drastic differences — both in the meaningfulness (e.g. in the behavior under protocol composition), and also in the restrictiveness, namely in the ability to assert security of natural protocols.

One consequence of this fact is that finding viable notions of security for cryptographic protocols remains an intriguing and lively research area. Another consequence is that appropriately formulating the security requirements of a given cryptographic task can be a delicate challenge in itself. In fact, this is often the "hard part" of the security analysis, more so than actually asserting that a given protocol satisfies the formulated property in the devised model.

Acknowledgments. My thinking and understanding of cryptographic protocols has been shaped over the years by discussions with many insightful researchers, too numerous to mention here. I thank you all. Oded Goldreich and Hugo Krawczyk were particularly influential, with often conflicting (complementary?) views of the field. I'm also grateful to the editor, Sergio Rajsbaum, for his effective blend of flexibility and persistence and to Shai Halevi, Ralf Küsters, and Birgit Pfitzmann for helpful remarks.

References

[BJP02] M. Backes, C. Jacobi, B. Pfitzmann. Deriving Cryptographically Sound Implementations Using Composition and Formally Verified Bisimulation. In proceedings of Formal Methods Europe (FME) 2002, pp. 310-329.

[BPW03] M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations. In *10th ACM conference on computer and communications security (CCS)*, 2003. Extended version at the eprint archive, http://eprint.iacr.org/2003/015/.

[BPW04] M. Backes, B. Pfitzmann, and M. Waidner. A general composition theorem for secure reactive systems. In *1st Theory of Cryptography Conference (TCC)*, LNCS 2951 pp. 336–354, Feb. 2004.

[B$^+$05] B. Barak, R. Canetti, Y. Lindell, R. Pass and T. Rabin. Secure Computation Without Authentication. In Crypto'05, 2005.

[BCNP04] B. Barak, R. Canetti, J. B. Nielsen, R. Pass. Universally Composable Protocols with Relaxed Set-Up Assumptions. 45th FOCS, pp. 186–195. 2004.

[BS05] B. Barak and A. Sahai, How To Play Almost Any Mental Game Over the Net - Concurrent Composition via Super-Polynomial Simulation. 46th FOCS, 2005.

[B91] D. Beaver. Secure Multi-party Protocols and Zero-Knowledge Proof Systems Tolerating a Faulty Minority. J. Cryptology, (1991) 4: 75-122.

[BH92] D. Beaver and S. Haber. Cryptographic protocols provably secure against dynamic adversaries. In *Eurocrypt '92*, LNCS No. 658, 1992, pages 307–323.

[BR93] M. Bellare and P. Rogaway. Entity authentication and key distribution. *CRYPTO'93*, LNCS. 773, pp. 232-249, 1994.

[BCG93] M. Ben-Or, R. Canetti and O. Goldreich. Asynchronous Secure Computation. 25th Symposium on Theory of Computing (STOC), 1993, pp. 52-61. Longer version appears in TR #755, CS dept., Technion, 1992.

[BGW88] M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. 20th Symposium on Theory of Computing (STOC), ACM, 1988, pp. 1-10.

[BKR94] M. Ben-Or, B. Kelmer and T. Rabin. Asynchronous Secure Computations with Optimal Resilience. *13th PODC,* 1994, pp. 183-192.

[B82] M. Blum. Coin flipping by telephone. IEEE Spring COMPCOM, pp. 133-137, Feb. 1982.

[BCC88] G. Brassard, D. Chaum and C. Crépeau. Minimum Disclosure Proofs of Knowledge. *JCSS*, Vol. 37, No. 2, pages 156–189, 1988.

[c95] R. Canetti. Studies in Secure Multi-party Computation and Applications.*Ph.D. Thesis,* Weizmann Institute, Israel, 1995.

[c00] R. Canetti. Security and composition of multi-party cryptographic protocols. J. Cryptology, Vol. 13, No. 1, winter 2000.

[c01] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. Extended abstract in *42nd FOCS,* 2001. A revised version (2005) is available at IACR Eprint Archive, eprint.iacr.org/2000/067/ and at the ECCC archive, http://eccc.uni-trier.de/eccc-reports/2001/TR01-016/.

[c+06] R. Canetti, L. Cheung, D. Kaynar, M. Liskov, N. Lynch, O. Pereira, and R. Segala. Task-Structured Probabilistic I/O Automata. In Workshop on discrete event systems (WODES), 2006.

[c+06a] R. Canetti, L. Cheung, D. Kaynar, M. Liskov, N. Lynch, O. Pereira, and R. Segala. Time-Bounded Task-PIOAs: A Framework for Analyzing Security Protocols. In 20th symposium on distributed computing (DISC), 2006.

[cdpw07] R. Canetti, Y. Dodis, R. Pass and S. Walfish. Universally Composable Security with Pre-Existing Setup. *4th theory of Cryptology Conference (TCC),* 2007.

[cfgn96] R. Canetti, U. Feige, O. Goldreich and M. Naor. Adaptively Secure Computation. *28th Symposium on Theory of Computing (STOC),* ACM, 1996. Fuller version in MIT-LCS-TR 682, 1996.

[cf01] R. Canetti and M. Fischlin. Universally Composable Commitments. Crypto '01, 2001.

[ch04] R. Canetti and J. Herzog. Universally Composable Symbolic Analysis of Cryptographic Protocols (The case of encryption-based mutual authentication and key-exchange). Eprint archive, http://eprint.iacr.org/2004/334. Extended Abstract at 3rd TCC, 2006.

[ck02] R. Canetti and H. Krawczyk. Universally Composable Key Exchange and Secure Channels . Eurocrypt '02, pages 337–351, 2002. LNCS No. 2332. Extended version at http://eprint.iacr.org/2002/059.

[ck02a] R. Canetti and H. Krawczyk. Security Analysis of IKE's Signature-based Key-Exchange Protocol. Crypto '02, 2002. Extended version at http://eprint.iacr.org/2002/120.

[ckl03] R. Canetti, E. Kushilevitz, Y. Lindell. On the Limitations of Universally Composable Two-Party Computation without Set-up Assumptions. EUROCRYPT 2003, pp. 68–86, 2003. Extended version at the eprint archive, eprint.iacr.org/2004/116.

[clos02] R. Canetti, Y. Lindell, R. Ostrovsky, A. Sahai. Universally composable two-party and multi-party secure computation. 34th STOC, pp. 494–503, 2002.

[cr03] R. Canetti and T. Rabin. Universal Composition with Joint State. Crypto'03, 2003.

[ccd88] D. Chaum, C. Crepeau, and I. Damgaard. Multi-party Unconditionally Secure Protocols. In *Proc.* 20*th Annual Symp. on the Theory of Computing (STOC)*, pages 11–19, ACM, 1988.

[CKLP06] L. Cheung, D. Kaynar, N. Lynch, O. Pereira. Compositional Security for Task-PIOAs. Manuscript, 2006.

[CGKS95] B. Chor, O. Goldreich, E. Kushilevitz, M. Sudan. Private Information Retrieval. 36th FOCS, 1995, pp. 41-50.

[DDMRS06] A. Datta, A. Derek, J. C. Mitchell, A. Ramanathan and A. Scedrov. Games and the Impossibility of Realizable Ideal Functionality. 3rd theory of Cryptology Conference (TCC), 2006.

[DKMR05] A. Datta, R. Küsters, J. C. Mitchell and A. Ramanathan. On the Relationships between Notions of Simulation-based Security. 2nd theory of Cryptology Conference (TCC), 2005.

[DM00] Y. Dodis and S. Micali. Secure Computation. CRYPTO '00, 2000.

[DDN00] D. Dolev. C. Dwork and M. Naor. Non-malleable cryptography. SIAM. J. Computing, Vol. 30, No. 2, 2000, pp. 391-437. Preliminary version in 23rd Symposium on Theory of Computing (STOC), 1991.

[DY83] D. Dolev and A. Yao. On the security of public-key protocols. IEEE Transactions on Information Theory, 2(29), 1983.

[DLMS99] N.A. Durgin, P.D. Lincoln, J.C. Mitchell and A. Scedrov. Undecidability of bounded security protocols. Workshop on Formal Methods and Security Protocols (FMSP), 1999.:w

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

[EG82] S. Even and Oded Goldreich. On the Security of Multi-Party Ping-Pong Protocols. 24th FOCS, 1983.

[F91] U. Feige. Ph.D. thesis, Weizmann Institute of Science, 1991.

[GRR98] R. Gennaro, M. Rabin and T Rabin. Simplified VSS and Fast-track Multiparty Computations with Applications to Threshold Cryptography, *17th PODC,* 1998, pp. 101-112.

[G01] O. Goldreich. Foundations of Cryptography (Vol. 1). Cambridge Press, 2001.

[G04] O. Goldreich. Foundations of Cryptography (Vol. 2). Cambridge Press, 2004.

[GK89] O. Goldreich and H. Krawczyk. On the Composition of Zero-Knowledge Proof Systems. SIAM. J. Computing, Vol. 25, No. 1, 1996.

[GMW87] O. Goldreich, S. Micali and A. Wigderson. How to Play any Mental Game. 19th Symposium on Theory of Computing (STOC), 1987, pp. 218-229.

[GO94] O. Goldreich and Y. Oren. Definitions and properties of Zero-Knowledge proof systems. J. Cryptology, Vol. 7, No. 1, 1994, pp. 1–32.

[GL90] S. Goldwasser, and L. Levin. Fair Computation of General Functions in Presence of Immoral Majority. CRYPTO '90, LNCS 537, 1990.

[GM84] S. Goldwasser and S. Micali. Probabilistic encryption. JCSS, Vol. 28, No 2, April 1984, pp. 270-299.

[GMRa89] S. Goldwasser, S. Micali and C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. SIAM Journal on Comput., Vol. 18, No. 1, 1989, pp. 186-208.

[GMRi88] S. Goldwasser, S. Micali, and R.L. Rivest. A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks. SIAM J. Comput., April 1988, pages 281–308.

[HM00] M. Hirt and U. Maurer. Complete characterization of adversaries tolerable in secure multi-party computation. J. Cryptology, Vol 13, No. 1, 2000, pp. 31-60. Preliminary version in *16th Symp. on Principles of Distributed Computing (PODC),* ACM, 1997, pp. 25–34.

[H85] C. A. R. Hoare. Communicating Sequential Processes. International Series in Computer Science, Prentice Hall, 1985.

[HU05] D. Hofheinz and D. Unruh. Comparing Two Notions of Simulatability. *2nd theory of Cryptology Conference (TCC),* pp. 86-103, 2005.

[IPSEC] The IPSec working group of the IETF. See http://www.ietf.org/html.charters/ipsec-charter.html

[KLR06] E. Kushilevitz, Y. Lindell and T. Rabin. Information-Theoretically Secure Protocols and Security Under Composition. 38th STOC, pages 109-118, 2006.

[K06] R. Küsters. Simulation based security with inexhaustible interactive Turing machines. 19th CSFW, 2006.

[L03] Y. Lindell. General Composition and Universal Composability in Secure Multi-Party Computation. 43rd FOCS, pp. 394–403. 2003.

[L04] Y. Lindell. Lower Bounds for Concurrent Self Composition. 1st Theory of Cryptology Conference (TCC), pp. 203–222. 2004.

[LLR02] Y. Lindell, A. Lysyanskaya and T. Rabin. On the composition of authenticated Byzantine agreement. 34th STOC, 2002.

[LPT04] Y. Lindell, M. Prabhakaran, Y. Tauman. Concurrent General Composition of Secure Protocols in the Timing Model. Manuscript, 2004.

[LMMS98] P. Lincoln, J. Mitchell, M. Mitchell, A. Scedrov. A Probabilistic Poly-time Framework for Protocol Analysis. 5th ACM Conf. on Computer and Communication Security, 1998, pp. 112-121.

[LT89] N. Lynch and M. R. Tuttle. An introduction to input/output automata. CWIQuarterly, 2(3):219-246, September 1989.

[LSV03] N. Lynch, R. Segala and F. Vaandrager. Compositionality for Probabilistic Automata. 14th CONCUR, LNCS vol. 2761, pages 208-221, 2003. Fuller version appears in MIT Technical Report MIT-LCS-TR-907.

[MMY06] T. Malkin, R. Moriarty and N. Yakovenko. Generalized Environmental Security from Number Theoretic Assumptions. *3rd Theory of Cryptology Conference (TCC),* 2006, pp. 343-359.

[MMS03] P. Mateus, J. C. Mitchell and A. Scedrov. Composition of Cryptographic Protocols in a Probabilistic Polynomial-Time Process Calculus. 14th CONCUR, pp. 323-345. 2003.

[MR91] S. Micali and P. Rogaway. Secure Computation. unpublished manuscript, 1992. Preliminary version in CRYPTO '91, LNCS 576, 1991.

[M89] R. Milner. Communication and Concurrency. Prentice Hall, 1989.

[M99] R. Milner. Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press, 1999.

[MMS98] J. Mitchell, M. Mitchell, A. Scedrov. A Linguistic Characterization of Bounded Oracle Computation and Probabilistic Polynomial Time. 39th FOCS, 1998, pp. 725-734.

[MRST06] John C. Mitchell, Ajith Ramanathan, Andre Scedrov, Vanessa Teague. A probabilistic polynomial-time process calculus for the analysis of cryptographic protocols. Theor. Comput. Sci. 353(1-3): 118-164 (2006). Preliminary version in LICS'01.

[P04] R. Pass. Bounded-concurrent secure multi-party computation with a dishonest majority. 36th STOC, pp. 232–241. 2004.

[PR03] R. Pass, A. Rosen. Bounded-Concurrent Secure Two-Party Computation in a Constant Number of Rounds. *44th FOCS*, 2003

[PR05a] R. Pass, A. Rosen. New and improved constructions of non-malleable cryptographic protocols. *STOC,* pp. 533-542, 2005.

[PR05b] R. Pass, A. Rosen. Concurrent and Non-Malleable Commitments. *FOCS,* 2005.

[P91] T. P. Pedersen: Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. CRYPTO 1991: 129-140

[PW94] B. Pfitzmann and M. Waidner. A general framework for formal notions of secure systems. Hildesheimer Informatik-Berichte 11/94, Universitat Hildesheim, 1994. Available at http://www.semper.org/sirene/lit.

[PSW00] B. Pfitzmann, M. Schunter and M. Waidner. Secure Reactive Systems. IBM Research Report RZ 3206 (#93252), IBM Research, Zurich, May 2000.

[PSW00a] B. Pfitzmann, M. Schunter and M. Waidner. Provably Secure Certified Mail. IBM Research Report RZ 3207 (#93253), IBM Research, Zurich, August 2000.

[PW00] B. Pfitzmann and M. Waidner. Composition and integrity preservation of secure reactive systems. 7th ACM Conf. on Computer and Communication Security (CCS), 2000, pp. 245-254.

[PW01] B. Pfitzmann and M. Waidner. A model for asynchronous reactive systems and its application to secure message transmission. IEEE Symposium on Security and Privacy, May 2001. Preliminary version in http://eprint.iacr.org/2000/066 and IBM Research Report RZ 3304 (#93350), IBM Research, Zurich, December 2000.

[PRS02] M. Prabhakaran, A. Rosen, A. Sahai. Concurrent Zero Knowledge with Logarithmic Round-Complexity. 43rd FOCS, 2002: 366-375

[PS04] M. Prabhakaran, A. Sahai. New notions of security: achieving universal composability without trusted setup. 36th STOC, pp. 242–251. 2004.

[PS05] M. Prabhakaran, A. Sahai. Relaxing Environmental Security: Monitored Functionalities and Client-Server Computation. *2nd Theory of Cryptology Conference (TCC),* 2005.

[RB89] T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multi-party Protocols with Honest Majority. 21st Symposium on Theory of Computing (STOC), 1989, pp. 73-85.

[RS91] C. Rackoff and D. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. CRYPTO '91, 1991.

[RK99] R. Richardson and J. Kilian. On the Concurrent Composition of Zero-Knowledge Proofs. In Eurocrypt99, LNCS 1592, pages 415–413.

[SB+06] C. Sprenger, M. Backes, D. Basin, B. Pfitzmann and M. Waidner. Cryptographically Sound Theorem Proving. 19th Computer Security Foundations Workshop (CSFW), 2006.

[Y82A] A. Yao. Protocols for Secure Computation. In 23rd Annual Symp. on Foundations of Computer Science (FOCS), pages 160–164. 1982.

[Y86] A. Yao, How to generate and exchange secrets, In 27th Annual Symp. on Foundations of Computer Science (FOCS), pages 162–167. 1986.

A Trusted-party based security: A mini survey

This section briefly surveys some works that are directly relevant to the development of the trusted-party paradigm as a method for defining security of protocols. (Indeed, this is only a fraction of the body of work on modeling cryptographic protocols and asserting security properties.) More detailed surveys on this topic can be found in [C00, C01]. Also, some of these works have already been mentioned earlier and are not re-addressed here.

Two works that essentially "laid out the field" of general security definitions for cryptographic protocols are the work of Yao [Y82A], which expressed for the first time the need for a general "unified" framework for expressing the security requirements of cryptographic tasks and for analyzing cryptographic protocols; and the work of Goldreich, Micali and Wigderson [GMW87], which put forth the approach of defining security via comparison with an ideal process involving a trusted party (albeit in a very informal way).

The first rigorous definitional framework is that of Goldwasser and Levin [GL90]. It was followed shortly by the frameworks of Micali and Rogaway [MR91] and Beaver [B91]. In particular, the notion of "reducibility" in [MR91] directly underlies the notion of protocol composition in many subsequent works, including the notion of universal composition as descried here. Beaver's framework is the first to directly formalize the idea of comparing a run of a protocol to an ideal process. Still, the [MR91, B91] formalisms only address security in restricted settings; in particular, they do not deal with computational issues.

All the work mentioned above concentrate on *synchronous* communication and the task of *secure function evaluation.* An extension to asynchronous communication networks is formulated in [BCG93]. A system model and notion of security for reactive functionalities is sketched in Pfitzmann and Waidner [PW94].

Canetti [C95] provides the first ideal-process based definition of computational security against resource bounded adversaries. [C00] strengthens the framework of [C95] to handle secure composition. In particular, security of protocols in that framework is shown to be preserved under non-concurrent universal composition. This work also contains sketches on how to strengthen the definition to support concurrent composition. A closely related formulation appears in [G04].

The framework of Hirt and Maurer [HM00] give a rigorous treatment of the case of reactive functionalities. Dodis and Micali [DM00] build on the definition of Micali and Rogaway [MR91] for unconditionally secure function evaluation, and prove that their notion of security is preserved under a general concurrent composition operation similar to universal composition. However, their definition involve notions that make sense only in settings where the communication is ideally private; thus this definition does not apply to the common setting where the adversary has access to the communication between honest parties.

The framework of Pfitzmann, Schunter and Waidner [PSW00, PW00] is the first to rigorously address *concurrent* universal composition in a computational setting. They define security for reactive functionalities in a synchronous setting and prove that security is preserved when a *single* instance of a subroutine protocol is composed concurrently with the calling protocol. An extension of the [PSW00, PW00] framework and notion (called *reactive simulatability*) to asynchronous networks appears in [PW01].

Universal composability in its full generality was first considered in [C01], which addressed the case of unbounded number of concurrently composed protocols. This work also demonstrated how the security requirements of a number of commonplace and seemingly unrelated cryptographic tasks can be captured via the trusted-party paradigm in the devised model.

A process calculus for representing probabilistic polynomial time interacting processes is developed in [LMMS98, MRST06]. In [MMS03] the notion of protocol emulation and realizing an ideal functionality is formalized in this model, and shown to be preserved under universal composition with any calling protocol. Other models that define emulation-based security include [DKMR05, K06, C+06a].

At very high level, the notions of security in [PW01, C01, MMS03, DKMR05, K06] are similar. However, the underlying system models differ in a number of respects, which significantly affect the expressibility and generality of the respective models, namely the range of real-life situations and concerns that can be captured by the respective formalisms. They also differ in their simplicity and ease of use. In addition, the models provide different degrees of abstraction and different tools for arguing about security properties. We leave a more detailed comparison out of scope.

Finally, we note that the above notions of security leave little room for non-determinism in protocol description and run-time scheduling. This is a natural choice, since non-determinism that is resolved arbitrarily at run-time seems inherently incompatible with security against computationally bounded adversaries. However, such modeling does not allow utilizing the traditional analytical advantages of non-determinism in modeling of distributed protocols. First steps towards incorporating in the model non-determinism that's resolved at runtime are taken in [C+06, C+06a]; the main idea here is to allow some parts of the protocol execution to be determined arbitrarily *after* all the algorithmic components are fixed.