Universally Composable Security: A New Paradigm for Cryptographic Protocols

Ran Canetti[*]

July 16, 2013

Abstract

We present a general framework for representing cryptographic protocols and analyzing their security. The framework allows specifying the security requirements of practically any cryptographic task in a unified and systematic way. Furthermore, in this framework the security of protocols is preserved under a general protocol composition operation, called universal composition.

The proposed framework with its security-preserving composition operation allows for modular design and analysis of complex cryptographic protocols from relatively simple building blocks. Moreover, within this framework, protocols are guaranteed to maintain their security in any context, even in the presence of an unbounded number of arbitrary protocol instances that run concurrently in an adversarially controlled manner. This is a useful guarantee, that allows arguing about the security of cryptographic protocols in complex and unpredictable environments such as modern communication networks.

Keywords: cryptographic protocols, security analysis, protocol composition, universal composition.

[*]IBM T.J. Watson Research Center, canetti@watson.ibm.com. Supported by NSF CyberTrust Grant #0430450.

Contents

1 Introduction

Rigorously demonstrating that a protocol "does its job securely" is an essential component of cryptographic protocol design. This requires coming up with an appropriate mathematical model for representing protocols, and then formulating, within that model, a *definition of security* that captures the requirements of the task at hand. Once such a definition is in place, we can show that a protocol "does its job securely" by demonstrating that its mathematical representation satisfies the definition of security within the devised mathematical model.

However, coming up with a good mathematical model for representing protocols, and even more so formulating adequate definitions of security within the devised model, turns out to be a tricky business. The model should be rich enough to represent all realistic adversarial behaviors, and the definition should guarantee that the intuitive notion of security is captured with respect to any adversarial behavior under consideration.

One main challenge in formulating the security of cryptographic protocols is capturing the threats coming from the execution environment, and in particular potential "bad interactions" with other protocols that are running in the same system or network. Another, related challenge is the need to come up with notions that allow building cryptographic protocols and applications from simpler building blocks while preserving security. Addressing these challenges is the focal point of this work.

Initially, definitions of security for cryptographic tasks considered only a single execution of the analyzed protocol. This is indeed a good choice for first-cut definitions of security. In particular, it allows for relatively concise and intuitive problem statement, and for simpler analysis of protocols. However, in many cases it turned out that the initial definitions are insufficient in more complex contexts, where protocols are deployed within more general protocol environments. Some examples include encryption, where the basic notion of semantic security [GM84] was later augmented with several flavors of security against chosen ciphertext attacks [NY90, DDN00, RS91, BDPR98] and adaptive security [BH92, CFGN96], in order to address general protocol settings; Commitment, where the original notions were later augmented with some flavors of non-malleability [DDN00, DIO98, FF00] and equivocation [BCC88, B96] in order to address the requirement of some applications; Zero-Knowledge protocols, where the original notions [GMRa89, GO94] were shown not to be closed under parallel and concurrent composition, and new notions and constructions were needed [GK89, F91, DNS98, RK99, BGGL04]; Key Exchange, where the original notions do not suffice for providing secure sessions [BR93, BCK98, Sh99, CK01]; Oblivious Transfer [R81, EGL85, GM00], where the first definitions do not guarantee security under concurrent composition.

One way to capture the security concerns that arise in a specific protocol environment or in a given application is to directly represent the given environment or application within an extended definition of security. Such an approach is taken, for instance in the cases of key-exchange [BR93, CK01], non-malleable commitments [DDN00], concurrent zero-knowledge [DNS98] and general concurrently secure protocols [P04, BS05, G11], where the definitions explicitly model several adversarially coordinated instances of the protocol in question. This approach, however, results in definitions with ever-growing complexity, and is inherently limited in scope since it addresses only specific environments and concerns.

An alternative approach, taken in this work, is to use definitions that consider the protocol in isolation, but guarantee *secure composition.* In other words, here definitions of security inspect only a single instance of the protocol *"in vitro"*. Security *"in vivo"*, namely in more realistic settings where a protocol instance may run concurrently with other protocols, is guaranteed by making sure that the security is preserved under a general *composition operation* on protocols. This approach considerably simplifies the process of formulating a definition of security and analyzing protocols. Furthermore, it guarantees security in arbitrary protocol environments, even ones which have not been explicitly considered.

In order to make such an approach meaningful, we first need to have a general framework for representing cryptographic protocols and their security properties. Indeed, otherwise it is not clear what "preserving security when running alongside other protocols" means, especially when these other protocols and the security requirements from them are arbitrary. Several general definitions of secure protocols were developed over the years, e.g. [GL90, MR91, B91, BCG93, PW94, C00, HM00, PSW00, DM00, PW00]. These definitions are obvious candidates for such a general framework. However, many of these works consider only restricted settings and classes of tasks; more importantly, the composition operations considered in those works fall short of guaranteeing general secure composition of cryptographic protocols, especially in settings where security holds only for computationally bounded adversaries and many protocol instances may be running concurrently in an adversarially coordinated way. We further elaborate on these works and their relation to the present one in Appendix A.

This work proposes a framework for representing and analyzing the security of cryptographic protocols. Within this framework, we formulate a general methodology for expressing the security requirements of cryptographic tasks. Furthermore, we define a very general method for composing protocols, and show that notions of security expressed within this framework preserve security under this composition operation. We call this composition operation universal composition and say that definitions of security in this framework (and the protocols that satisfy them) are universally composable (UC). Consequently, we dub this framework the UC security framework.[1] As we'll see, the fact that security in this framework is preserved under universal composition implies that a secure protocol for some task remains secure even it is running in an arbitrary and unknown multi-party, multi-execution environment. In particular, some standard security concerns, such as non-malleability and security under concurrent composition, are satisfied even with respect to an unbounded number of instances of either the same protocol or other protocols.

The rest of the Introduction is organized as follows. Section 1.1 presents the basic definitional approach and the ideas underlying the formalism. Section 1.2 presents the universal composition operation and theorem. Section 1.3 discusses the issues associated with substantiating the general approach into a rigorous and usable framework. Related work, including both prior work and work that was done following the publication of the first version of this work, is reviewed in Appendix A.

1.1 The definitional approach

We briefly sketch the proposed framework and highlight some of its properties. A more comprehensive motivational presentation appears in [C06]. The overall definitional approach is the same as in most other general definitional frameworks mentioned above, and goes back to the seminal work of Goldreich, Micali and Wigderson [GMW87]: 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 hand their inputs to a *trusted party* who locally computes

[1]We use similar names for two very different objects: A notion of security and a composition operation. This choice of names is motivated in Appendix A.

the outputs, and hands each party its prescribed output. This ideal process can be regarded as a "formal specification" of the security requirements of the task. A protocol is said to *securely realize* the task if running the protocol "emulates" 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.

Several formalizations of this general definitional approach exist, including the definitional works mentioned above, providing a range of secure composability guarantees in a variety of computational models. To better understand the present framework, we first briefly sketch the definitional framework of [C00], which provides a basic instantiation of the "ideal process paradigm" for the traditional task of secure function evaluation, namely evaluating a known function of the secret inputs of the parties in a synchronous and ideally authenticated network.

The model of protocol execution considered in [C00] consists of a set of interacting computing elements, representing the parties running the protocol. Formally, these elements are modeled as interactive Turing machines (ITMs).[2] An additional ITM represents the adversary, who controls a subset of the parties. In addition, the adversary has some control over the scheduling of message delivery, subject to the synchrony guarantee. The parties and adversary interact on a given set of inputs and each party eventually generates local output. The concatenation of the local outputs of the adversary and all parties is called the global output. In the ideal process for evaluating some function f, all parties ideally hand their inputs to an incorruptible *trusted party,* who computes the function values and hands them to the parties as specified. Here the adversary is limited to interacting with the trusted party in the name of the corrupted parties. Protocol π securely evaluates a function f if for any adversary $\mathcal{A}$ (that interacts with the protocol) there exists an ideal-process adversary $\mathcal{S}$ such that, for any set of inputs to the parties, the global output of running π with $\mathcal{A}$ is indistinguishable from the global output of the ideal process for f with adversary $\mathcal{S}$.

This definition suffices for capturing the security of protocols in a "stand-alone" setting where only a single protocol instance runs in isolation. Indeed, if π securely evaluates f then the parties running π are guaranteed to generate outputs that are indistinguishable from the values of f on the same inputs. Furthermore, any information gathered by an adversary that interacts with π is generatable by an adversary that only gets the inputs and outputs of the corrupted parties. In addition, this definition is shown to guarantee security under non-concurrent composition, namely as long as no two protocol instances run concurrently. However, when protocol instances run concurrently, this definition no longer guarantees security: There are natural protocols that meet the [C00] definition but are insecure when as few as *two* instances run concurrently. We refer the reader to [C00, C06] for more discussions on the implications of, and motivation for, this definitional approach. Some examples for the failure to preserve security under concurrent composition are given in [C06].

The UC framework preserves the overall structure of that approach. The difference lies in new formulations of the model of computation and the notion of "emulation". As a preliminary step towards presenting these new formulation, we first present an alternative and equivalent formulation of the [C00] definition. In that formulation a new algorithmic entity, called the environment machine, is added to the model of computation. (The environment machine can be regarded as representing *whatever is external to the current protocol execution.* This includes other protocol executions

[2]While following tradition, the specific choice of Turing machines as the underlying computational model is somewhat arbitrary. Any other model that provides a concrete way to measure the complexity of computations, such as e.g. RAM or PRAM machines, and boolean or arithmetic circuits would be adequate.

and their adversaries, human users, etc.) The environment interacts with the protocol execution twice: First, it hands arbitrary inputs of its choosing to the parties and to the adversary. Next, it collects the outputs from the parties and the adversary. Finally, the environment outputs a single bit, which is interpreted as saying whether the environment thinks that it has interacted with the protocol or with the ideal process for f. Now, say that protocol π securely evaluates a function f if for any adversary $\mathcal{A}$ there exists an "ideal adversary" $\mathcal{S}$ such that no environment $\mathcal{E}$ can tell with non-negligible probability whether it is interacting with π and $\mathcal{A}$ or with $\mathcal{S}$ and the ideal process for f. (In fact, a similar notion of environment is already used in [C00] to capture non-concurrent composability for adaptive adversaries.)

The main difference between the UC framework and the basic framework of [C00] is in the way the environment interacts with the adversary. Specifically, in the UC framework the environment and the adversary are allowed to interact freely throughout the course of the computation. In particular, they can exchange information after each message or output generated by a party running the protocol. If protocol π securely realizes function f with respect to this type of "interactive environment" then we say that π UC-realizes f.

This seemingly small difference in the formulation of the computational models is in fact very significant. From a conceptual point of view, it represents the fact that "information flow" between the protocol instance under consideration and the rest of the network may happen at any time during the run of the protocol, rather than only at input or output events. Furthermore, at each point the information flow may be directed both "from the outside in" and "from the inside out". Modeling such "circular" information flow is essential for capturing the threats of a multi-instance concurrent execution environment. (See some concrete examples in [C06].) From a technical point of view, the environment now serves as an "interactive distinguisher" between the protocol execution and the ideal process. This imposes a considerably more severe restriction on the ideal adversary $\mathcal{S}$, which must be constructed in the proof of security: In order to make sure that the environment $\mathcal{E}$ cannot tell between a real protocol execution and the ideal process, $\mathcal{S}$ now has to interact with $\mathcal{E}$ throughout the execution, just as $\mathcal{A}$ did. Furthermore, $\mathcal{S}$ cannot "rewind" $\mathcal{E}$. Indeed, it is this pattern of free interaction between $\mathcal{E}$ and $\mathcal{A}$ that allows proving that security is preserved under universal composition.

An additional difference between the UC framework and the basic framework of [C00] is that the UC framework allows capturing not only secure function evaluation but also *reactive* tasks where new input values are received and new output values are generated throughout the computation.Furthermore, new inputs may depend on previously generated outputs, and new outputs may depend on all past inputs and local random choices. This is obtained by replacing the "trusted party" in the ideal process for secure function evaluation with a general algorithmic entity called an ideal functionality. The ideal functionality, which is modeled as another ITM, repeatedly receives inputs from the parties and provides them with appropriate output values, while maintaining local state in between. This modeling guarantees that the outputs of the parties in the ideal process have the expected properties with respect to the inputs, even when new inputs are chosen adaptively based on previous outputs. We note that this extension of the model is "orthogonal" to the previous one, in the sense that either extension is valid on its own. Some other differences from [C00] (e.g., capturing different communication models and the ability to dynamically generate programs) are discussed in later sections.

The resulting definition of security turns out to be quite robust, in the sense that several natural definitional variants end up being equivalent. For instance, the notion of security as stated above is equivalent to the seemingly weaker variants where $\mathcal{S}$ may depend on the environment, or where the real-life adversary $\mathcal{A}$ is restricted to simply serve as a channel for relaying information between the environment and the protocol. It is also equivalent to the seemingly stronger variant where the ideal adversary $\mathcal{S}$ is restricted to black-box access to the adversary $\mathcal{A}$. (We remark that in other frameworks these variants result in different formal requirements; see e.g. [HU05].)

1.2 Universal Composition

Consider the following method for composing two protocols into a single composite protocol. (It may be useful to think of this composition operation as a generalization of the "subroutine substitution" operation for sequential algorithms to the case of distributed protocols.) Let π be some arbitrary protocol where the parties make ideal calls to some ideal functionality $\mathcal{F}$; in fact, they may make calls to multiple instances of $\mathcal{F}$. That is, in addition to the standard set of instructions, π may include instructions to provide instances of $\mathcal{F}$ with some input values, and to obtain output values from these instances of $\mathcal{F}$. Here the different instances of $\mathcal{F}$ are running at the same time without any global coordination. We call such protocols $\mathcal{F}$-hybrid protocols. (For instance, π may be a zero-knowledge protocol and $\mathcal{F}$ may provide the functionality of "digital envelopes", representing ideal commitment.) The burden of distinguishing among the instances of $\mathcal{F}$ is left with protocol π; we provide a generic mechanism for doing so, using *session identifiers.*

Now, let ρ be a protocol that UC-realizes $\mathcal{F}$, according to the above definition. Construct the composed protocol π^ρ by starting with protocol π, and replacing each invocation of a new instance of $\mathcal{F}$ with an invocation of a new instance of ρ. Similarly, inputs given to an instance of $\mathcal{F}$ are now given to the corresponding instance of ρ, and any output of an instance of ρ is treated as an output obtained from the corresponding instance of $\mathcal{F}$. It is stressed that, since protocol π may use an unbounded number of instances of $\mathcal{F}$ at the same time, we have that in protocol π^ρ there may be an unbounded number of instances of ρ which are running concurrently on related and dynamically chosen inputs.

The universal composition theorem states that running protocol π^ρ, with no access to $\mathcal{F}$, has essentially the same effect as running the original $\mathcal{F}$-hybrid protocol π. More precisely, it guarantees that for any adversary $\mathcal{A}$ there exists an adversary $\mathcal{A}_\mathcal{F}$ such that no environment machine can tell with non-negligible probability whether it is interacting with $\mathcal{A}$ and parties running π^ρ, or with $\mathcal{A}_\mathcal{F}$ and parties running π. In particular, if π UC-realizes some ideal functionality $\mathcal{G}$ then so does π^ρ.

On the universality of universal composition. Many different ways of "composing together" protocols into larger systems are considered in the literature. Examples include sequential, parallel, and concurrent composition, of varying number of protocol instances, where the composed instances are run either by the same set of parties or by different sets of parties, use either the same program or different programs, and have either the same input or different inputs. A more detailed discussion appears in [C06].

All these composition methods can be captured as special cases of universal composition. That is, any such method for composing together protocol instances can be captured via an appropriate "calling protocol" ρ that uses the appropriate number of protocol instances as subroutines, provides them with appropriately chosen inputs, and arranges for the appropriate synchronization in message delivery among the various subroutine instances. Consequently, it is guaranteed that a protocol that UC-realizes an ideal functionality $\mathcal{F}$ continues to UC-realize $\mathcal{F}$ even when composed with other protocols using any of the composition operations considered in the literature. In addition, universal composition allows expressing new ways of composing protocols, such as composing protocols where some of the local states are shared by multiple instances.

Universal composition also allows formulating new ways to put together protocols (or, equivalently, new ways to decompose complex systems); a salient example here is the case of composition of protocol instances that have some joint state and randomness.

Interpreting the composition theorem. Traditionally, secure composition theorems are treated as tools for modular design and analysis of complex protocols. (For instance, this is the main motivation in [MR91, C00, DM00, PW00, PW01].) That is, given a complex task, first partition the task to several, simpler sub-tasks. Then, design protocols for securely realizing the sub-tasks, and in addition design a protocol for realizing the given task assuming that evaluation of the sub-tasks is possible. Finally, use the composition theorem to argue that the protocol composed from the already-designed sub-protocols securely realizes the given task. Note that in this interpretation the protocol designer knows in advance which protocol instances are running together and can control how protocols are scheduled.

The above application is indeed very useful. In addition, this work proposes another interpretation of the composition theorem, which is arguably stronger: We use it as a tool for addressing the concern described at the beginning of the Introduction, namely for gaining confidence in the sufficiency of a definition of security in a given protocol environment. Indeed, protocols that UC-realize some functionality are guaranteed to continue doing so within any protocol environment — even environments that are not known a-priori, and even environments where the participants in a protocol execution are unaware of other protocol instances that may be running concurrently in the system in an adversarially coordinated manner. This is a very useful (in fact, almost essential) security guarantee for protocols that run in complex and unpredictable environments, such as modern communication networks.

1.3 Using the framework

Perhaps the most important criterion for a general analytical framework such as the present one is its usefulness in analyzing the security of protocols of interest in settings of interest. There are a number of aspects here. First off, a useful framework should allow us to represent the protocols we are interested in. It should also allow expressing the security requirements of tasks of interest. Similarly, it should allow us to adequately express the execution environment we are interested in; This includes capturing the security threats we are concerned about, as well as the security guarantees we are given. Finally, the framework should be as intuitive and easy to use as possible. This means that representing requirements and guarantees should be natural and transparent. It also means that the framework should be as simple as possible, and that there should be easy and flexible ways to delineate and isolate individual parts of a given, potentially complex system.

These considerations are the leading guidelines in the many definitional choices made in substantiating the definitional ideas described earlier. Here we briefly highlight few of these choices. More elaborate discussions of definitional choices appear throughout this work.

One set of choices is geared towards enhancing the ability of the model to express a variety of realistic situations, protocol execution methods, and threats. Towards this goal, the model allows capturing open, multi-party distributed systems where no a-priori bound on the number of participants is known in advance, and where parties can join the system with programs and identities that are chosen dynamically during the course of the computation. Indeed, such modeling seems essential given the dynamic and reconfigurable nature of modern computer systems and networks, and the dynamic and polymorphic nature of modern attacks and viruses. To enable such modeling we develop formal mechanisms for participants to identify each other and to address messages to the desired recipient. We also extend the traditional definitions of resource-bounded computation so as to handle such dynamic systems. In addition, we provide mechanisms for identifying individual *protocols* and *protocol instances* within such systems, so that the number and identities of the participants in each protocol instance, as well as the number of protocol instances running concurrently, can change dynamically depending on the execution with no a-priori bound.

We remark that this modeling approach stands in contrast to existing models of distributed computing. Indeed, existing models typically impose more static restrictions on the system; this results in reduced ability to express scenarios and concerns that are prevalent in modern networks.

Another set of choices is geared toward allowing the protocol analyst to express security requirements in a precise and flexible way. This includes providing simple ways to express basic concerns, such as correctness, secrecy and fairness. It also includes providing ways to "fine-tune" the requirements at wish. In the present model, where a set of security requirements translates to a program of an ideal functionality, this means providing ways for writing ideal functionalities that express different types of requirements. A main tool here is to have the ideal functionality exchange information directly with the adversary throughout the computation. This allows expressing both the allowed adversarial influence and the allowed information leakage. In addition, the model allows the ideal functionality to execute code provided by the adversary; this allows expressing adversarial influence that is legitimate as long as it is carried out in isolation from the external environment.

Yet another set of choices is geared towards keeping the basic framework simple and concise, while allowing to capture a variety of communication, corruption, and trust models. For this purpose we use the same technical tool, ideal functionalities, to represent both a security specification for protocols, and the abstractions provided by a given communication, corruption or trust model. In the first use, we consider protocols that UC-realize the given ideal functionality, $\mathcal{F}$. In the second use, we consider $\mathcal{F}$-hybrid protocols, namely protocols that use the ideal functionality as a "trusted subroutine". In fact, it is often convenient to use an ideal functionality in both ways. Here the universal composition theorem implies that a protocol that UC-realizes $\mathcal{F}$ can be composed with a protocol designed in the abstract model captured by $\mathcal{F}$, to obtain a protocol that no longer needs the abstractions provided by $\mathcal{F}$.

Relying on this dual interpretation of ideal functionalities, we allow the framework to provide only very basic and rudimentary methods of communication between parties. This basic model is not intended to adequately capture any realistic setting. Realistic settings are then captured via formulating appropriate ideal functionalities within the basic model. This approach greatly simplifies the basic framework and facilitates arguing about it. It also makes assertions about the framework (such as the universal composition theorem) more general, since they directly apply to any communication, corruption or trust model devised within the framework. In addition, this approach provides greater flexibility in expressing different abstractions and variants thereof. The present modular approach to capturing abstract models should be contrasted with other models in the literature for security analysis, which are typically tied to a specific setting and have to be re-done with the appropriate modifications for each new setting.

To exemplify these aspects of the UC framework, we present a handful of ideal functionalities that are aimed at capturing some salient communication and corruption models. Other ideal functionalities appear in the literature.

Finally, to simplify the presentation of the model and make it more modular, we separate the description of the basic model of computation from the definition of security. That is, we first present a general model for representing multiple computational processes that run concurrently and interact with each other. In contrast with other concurrency models in the literature, this model is specifically geared towards capturing distributed computations that are computationally bounded yet adaptive and dynamically changing. We then formulate UC-emulation on top of this basic model. This separation highlights the importance of the underlying model, and allows considering alternative ones without losing in the overall structure of the definition and the composition theorem. Furthermore, we believe that the basic model proposed here is valuable in of itself, even without the notion of UC emulation that's built on top.

1.4 Organization

Due to its length, the review of related work is postponed to the Appendix. Section 2 contains an informal exposition of the framework, definition of security, and composition theorem. The basic model for representing multiparty protocols is presented in Section 3. The general definition of security is presented in Section 4. The composition theorem and its proof are presented in Section 5. Finally, Section 6 demonstrates how some salient models of computation may be captured within the UC framework.

2 The framework in a nutshell

This section presents a simplified and somewhat informal version of the definition of security and the composition theorem. The purpose of this presentation is to highlight the salient definitional ideas that underlie the UC framework. For this purpose, we intentionally "gloss over" many details that are essential for a rigorous general treatment; these are postponed to later sections. Still, this section intends to be self-contained.

Section 2.1 sketches a model for representing multiple interacting computer programs. Section 2.2 presents the definition of security. Section 2.3 presents the composition theorem and its proof.

2.1 The underlying computational model

As a first step, we present a model for representing computing elements that interact over an asynchronous and untrusted network. This model is rather rudimentary, and is formulated only for the purpose of the informal overview. A considerably more detailed and general model is presented in Section 3, along with motivating discussions.

The basic computing unit. The basic computing unit represents a running instance of a computer program (algorithm). For now, we omit a precise description of such a unit. Possible formalizations include an interactive Turing machine as in [GMRa89, G01], a random-access-memory (RAM) machine, a process (as in [M89, M99, H85, LMMS99]) an I/O automaton (as in [Ly96]), a system in the Abstract Cryptography model [MR11], etc. For the rest of this section we'll call such a unit a machine.

The model of computation consists of several machines that run "concurrently" (i.e., alongside each other) and provide each other with information. It will be convenient to distinguish three different ways in which a machine M can provide information to a machine M'. M can either provide input to M', send a message to M', or provide subroutine output to M'. Figure 2.1 presents a graphical depiction of a machine. (This partitioning between types of input is not essential; however we find it greatly clarifies and simplifies the mode.)

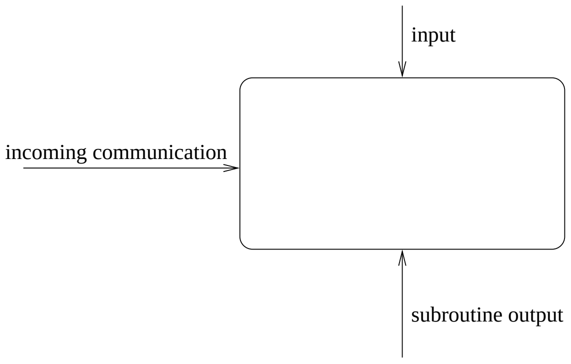

Figure 1: A basic computing unit (machine). Information from the outside world comes as either *input*, or *incoming communication*, or *subroutine output*. For graphical clarity, in future drawings we draw inputs as lines coming from above, incoming communication as lines coming from either side, and subroutine outputs as lines coming from below.

There are many ways to model a system whose components run concurrently. While the definitional approach of this work applies regardless of the specific modeling, for concreteness we consider a specific execution model. The model is simple: An execution of a system of machines $M_1, M_2, ...$ on input x starts by running M_1, called the *initial machine*, with input x. From this point on, the machines take turns in executing according to the following order: Initially, a single machine is active. Whenever a machine M provides information to machine M', the execution of M is suspended and the execution of M' begins (or resumes). It follows that at any point in time throughout the computation only a single machine is active.

Protocols and protocol instances. A protocol is an algorithm written for a distributed system. That is, a protocol consists of a collection of computer programs that exchange information with each other, where each program is run by a different participant and with different local input. For simplicity, in this section we restrict attention to protocols where the number of participants is fixed. A protocol with m participants is called an m-party protocol.

An instance of a protocol within a system of machines represents a sequence of machines, such that all these machines "relate to each other as part of the same protocol execution". In the general model, a precise formulation of this seemingly simple concept is tricky; for the purpose of this section we simply specify an instance of an m-party protocol $\pi = \pi_1, ..., \pi_m$ within a system $\mathcal{M} = M_1, M_2, ...$ of machines by naming a specific subsequence of $\mathcal{M}$, such that the ith machine in the subsequence runs the ith program, π_i.

Some of the machines (or, parties) in a protocol instance may be designated as subroutines of other parties in the instance. Intuitively, if party M' is a subroutine of party M then M will provide input to M' and obtain subroutine output from M'. A party of an instance of protocol π that is not a subroutine of another party of this instance of π is called a main party of that instance of π.

Polynomial time ITMs and protocols. We restrict attention to systems where all the machines have only "feasible" computation time, where feasible is interpreted as polynomial in some parameter. Furthermore, we assume that the overall computation of a system is "polynomial", in the sense that it can be simulated on a standard polynomially bounded Turing machine. We defer more precise treatment to subsequent sections.

2.2 Defining security of protocols

Following [GMW87], security of protocols with respect to a given task is defined by comparing an execution of the protocol to an ideal process where the outputs are computed by a trusted party that sees all the inputs. We substantiate this approach as follows. First, we substantiate the process of executing a protocol in the presence of an adversary and in a given computational environment. Next, the "ideal process" for carrying out the task is substantiated. Finally, we define what it means for an execution of the protocol to "mimic" the ideal process.

The model of protocol execution. We describe the model of executing an m-party protocol π in the presence of an adversary and in a given execution environment (or, "context").

The model consists of a system of machines $(\mathcal{E}, \mathcal{A}, \pi_1, ..., \pi_m)$ where $\mathcal{E}$ is called the environment, $\mathcal{A}$ is called the adversary, and $\pi_1, ..., \pi_m$ are an instance of protocol π. Intuitively, the environment represents all the other protocols running in the system, including the protocols that provide inputs to, and obtain outputs from, the protocol instance under consideration. The adversary represents adversarial activity that is directly aimed at the protocol execution under consideration, including attacks on protocol messages and corruption of protocol participants.

We impose the following restrictions on the way in which the machines may interact. The environment $\mathcal{E}$ is allowed to provide only *inputs* to other machines. Furthermore, it can provide inputs only to $\mathcal{A}$ and to the *main* parties of π. A party of π may send messages to $\mathcal{A}$, give inputs to its subroutines, or give subroutine output to the machines whose subroutine it is. If this party is a main party of π then it may provide subroutine output to $\mathcal{E}$. The adversary $\mathcal{A}$ may send messages to the parties of π or give subroutine output to $\mathcal{E}$. A graphical depiction of the model of protocol execution appears in Figure 2.

Let $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(z)$ denote the random variable (over the local random choices of all the involved machines) describing the output of environment $\mathcal{E}$ when interacting with adversary $\mathcal{A}$ and parties running protocol π on input z as described above. Let $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$ denote the ensemble $\{\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(z)\}_{z\in\{0,1\}^*}$. For the purpose of the present framework, it suffices to consider the case where the environment is allowed to output only a single bit. In other words, the ensemble $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$ is an ensemble of distributions over $\{0, 1\}$.

Discussion. Several remarks are in order at this point. First note that, throughout the process of protocol execution, the environment $\mathcal{E}$ has access only to the inputs and outputs of the main parties of π. It has direct access neither to the communication among the parties, nor to the inputs and outputs of the subroutines of π. The adversary $\mathcal{A}$ has access only to the communication among the parties and has no access to their inputs and outputs. This is in keeping with the intuition that

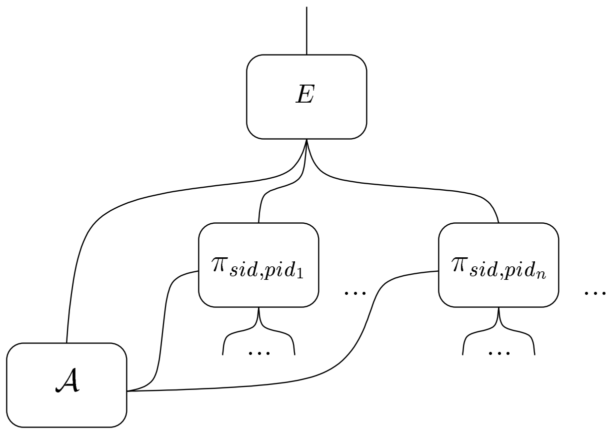

Figure 2: The model of protocol execution. The environment $\mathcal{E}$ writes the inputs and reads the subroutine outputs of the main parties running the protocol, while the adversary $\mathcal{A}$ controls the communication. In addition, $\mathcal{E}$ and $\mathcal{A}$ interact freely. The parties of π may have subroutines, to which $\mathcal{E}$ has no direct access.

$\mathcal{E}$ represents the protocols that provides inputs to and obtains outputs from the present instance of π, while $\mathcal{A}$ represents an adversary that attacks the protocol via the communication links, without having access to the local (and potentially secret) inputs and outputs.

In addition, $\mathcal{E}$ and $\mathcal{A}$ may exchange information freely during the course of the computation. It may appear at first that no generality is lost by assuming that $\mathcal{A}$ and $\mathcal{E}$ disclose their entire internal states to each other. A closer look shows that, while no generality is lost by assuming that $\mathcal{A}$ reveals its entire state to $\mathcal{E}$, the interesting cases occur when $\mathcal{E}$ holds some "secret" information back from $\mathcal{A}$, and tests whether the information received from $\mathcal{A}$ is correlated with the "secret" information. In fact, as we'll see, keeping $\mathcal{A}$ and $\mathcal{E}$ separate is crucial for the notion of security to make sense.

Another point to notice is that this model gives the adversary complete control over the communication. That is, the model represents a completely asynchronous, unauthenticated, and unreliable network. This is indeed a very rudimentary model for communication; we call it the bare model. More "abstract" (or, "idealized") models are defined later, building on this bare model.

Yet another point is that the model does not contain a dedicated instruction for party corruption. Party corruption is modeled as a special type of incoming message to the party; the party's response to this message is determined by a special part of the party's program. (To guarantee that the environment knows who is corrupted, we restrict attention to adversaries that notify the environment upon each corruption of a party.)

Finally, note that the only external input to the process of protocol execution is the input of $\mathcal{E}$. This input can be seen as representing an initial state of the system; in particular, it includes the inputs of all parties. From a complexity-theoretic point of view, providing the environment with arbitrary input (of polynomial length) is equivalent to stating that the environment is a non-uniform (polynomial time) ITM.

Ideal functionalities and ideal protocols. Security of protocols is defined via comparing the protocol execution to an *ideal process* for carrying out the task at hand. For convenience of presentation, we formulate the ideal process for a task as a special protocol within the above model

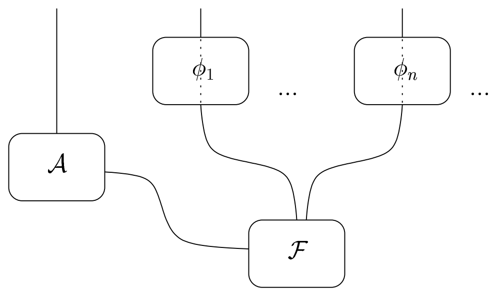

Figure 3: The ideal protocol $\text{IDEAL}_{\mathcal{F}}$ for an ideal functionality $\mathcal{F}$. The main parties of $\text{IDEAL}_{\mathcal{F}}$, namely $\phi_1, ..., \phi_n$, are "dummy parties": they only relay inputs to $\mathcal{F}$, and relay outputs of $\mathcal{F}$ to the calling ITIs. The adversary $\mathcal{A}$ communicates only with $\mathcal{F}$.

of protocol execution. (This avoids formulating an ideal process from scratch.) A key ingredient in this special protocol, called the ideal protocol, is an ideal functionality that captures the desired functionality, or the specification, of the task by way of a set of instructions for a "trusted party".

That is, let $\mathcal{F}$ be an ideal functionality (i.e., an algorithm for the trusted party), and assume that $\mathcal{F}$ expects to interact with m participants. Then an instance of the ideal protocol $\text{IDEAL}_{\mathcal{F}}$ consists of m main parties, called dummy parties, plus a party $\mathcal{F}$ that's a subroutine of all the main parties. Upon receiving an input v, each dummy party forwards v as input to the subroutine running $\mathcal{F}$. Any subroutine output coming from $\mathcal{F}$ is forwarded by the dummy party as subroutine output to the environment. Note that the inputs and outputs are handed over directly and reliably. A graphical depiction of the ideal protocol appears in Figure 3.

Note that $\mathcal{F}$ can model reactive computation, in the sense that it can maintain local state and its outputs may depend on all the inputs received and all random choices so far. In addition, $\mathcal{F}$ may receive messages directly from the adversary $\mathcal{A}$, and may contain instructions to send messages to $\mathcal{A}$. This "back-door channel" of direct communication between $\mathcal{F}$ and $\mathcal{A}$ provides a way to *relax* the security guarantees provided $\mathcal{F}$. Specifically, by letting $\mathcal{F}$ take into account information received from $\mathcal{A}$, it is possible to capture the "allowed influence" of the adversary on the outputs of the parties, in terms of both contents and timing. By letting $\mathcal{F}$ provide information directly to $\mathcal{A}$ it is possible to capture the "allowed leakage" of information on the inputs and outputs of the parties.

Protocol emulation. It remains to define what it means for a protocol to "mimic" or "emulate" the ideal process for some task. As a step towards this goal, we first formulate a more general notion of emulation, which applies to any two protocols. Informally, protocol π emulates protocol ϕ if, from the point of view of any environment, protocol π is "just as good" as ϕ, in the sense that no environment can tell whether it is interacting with π and some (known) adversary, or with ϕ and some other adversary. More precisely:

Definition (protocol emulation, informal statement): *Protocol π* UC-emulates *protocol ϕ if for any adversary $\mathcal{A}$ there exists an adversary $\mathcal{S}$ such that, for any environment $\mathcal{E}$ the ensembles $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$ and $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}}$ are indistinguishable. That is, on any input, the probability that $\mathcal{E}$ outputs 1 after interacting with $\mathcal{A}$ and parties running π differs by at most a negligible amount from the probability that $\mathcal{E}$ outputs 1 after interacting with $\mathcal{S}$ and ϕ.*

We often call the adversary $\mathcal{S}$ a *simulator.* This is due to the fact that in typical proofs of security the constructed $\mathcal{S}$ operates by simulating an execution of $\mathcal{A}$. Also, we call the emulated protocol ϕ as a reminder that in the definition of realizing a functionality (see below), ϕ takes the role of the ideal protocol for some ideal functionality $\mathcal{F}$.

The notion of protocol emulation treats the environment as an "interactive distinguisher" between the process of running protocol π with adversary $\mathcal{A}$ and the process of running protocol ϕ with adversary $\mathcal{S}$; it is required that the two interactions remain indistinguishable even given the ability to interact with them as they evolve. In this sense, the present notion strengthens the traditional notion of indistinguishability of distributions, and in particular the security notion of [c00]. Still, this notion is a relaxation of the notion of *observational equivalence* of processes (see, e.g., [m89]); indeed, observational equivalence essentially fixes the entire system outside the protocol instances, whereas protocol emulation allows the analyst to choose an appropriate simulator that will make the two systems look observationally equivalent.

Securely realizing an ideal functionality. Once the general notion of protocol emulation is defined, the notion of realizing an ideal functionality is immediate:

Definition (realizing functionalities, informal statement): *Protocol* π UC-realizes *an ideal functionality* $\mathcal{F}$ *if* π *emulates* IDEAL$_{\mathcal{F}}$*, the ideal protocol for* $\mathcal{F}$*.*

We recall the rationale behind this definition. Consider a protocol π that UC-realizes and ideal functionality $\mathcal{F}$. Observe that the parties running π are guaranteed to generate outputs that are indistinguishable from the outputs provided by $\mathcal{F}$ on the same inputs; this guarantees *correctness.* Furthermore, any information gathered by an adversary that interacts with π is obtainable by an adversary that only interacts with $\mathcal{F}$; this guarantees *secrecy.* See [c06] for more discussion on the motivation for and meaning of this definitional style.

2.3 The composition theorem

As in the case of protocol emulation, we present the composition operation and theorem in the more general context of composing two arbitrary protocols. The case of ideal protocols and protocols that UC-realize them follows as a corollary.

We first define what it means for one protocol to use another protocol as a subroutine. Essentially, a protocol ρ uses protocol ϕ as a subroutine if some or all of the programs of ρ use programs of ϕ as subroutines, and these programs of ϕ communicate with each other as a single protocol instance. Said otherwise, protocol instance $\vec{\phi} = \phi_1, ..., \phi_m$ of ϕ is a subroutine of protocol instance $\vec{\rho} = \rho_1, ..., \rho_{m'}$ of ρ if each machine in $\vec{\phi}$ appears as a machine in $\vec{\rho}$ and is a subroutine of another machine in $\vec{\rho}$. We note that an instance of ρ may use multiple subroutine instances of ϕ; these instances of ϕ may run concurrently, in the sense that activations of parties in the two instances may interleave an an arbitrary order.

The universal composition operation. The universal composition operation is a natural generalization of the "subroutine substitution" operation for sequential algorithms to the case of distributed protocols. That is, let ρ be a protocol that uses protocol ϕ as a subroutine, and let π be a protocol that UC-emulates ϕ. The composed protocol, denoted $\rho^{\phi\rightarrow\pi}$, is the protocol that is identical to ρ, except for the following change. For *each instance of* ϕ that's a subroutine of this instance of ρ, and for all i, the ith machine of this instance of ϕ now runs the program of the ith machine of π. In particular, all the inputs provided to an instance of ϕ that's a subroutine of this instance of ρ are now given to the corresponding instance of π, and all the outputs of this instance of π are treated as coming from the corresponding instance of ϕ. It is stressed that an instance of ρ may use multiple instances of ϕ; In this case, an instance $\rho^{\phi\to\pi}$ will use multiple instances of ρ. A graphical depiction of the composition operation appears in Figure 4.

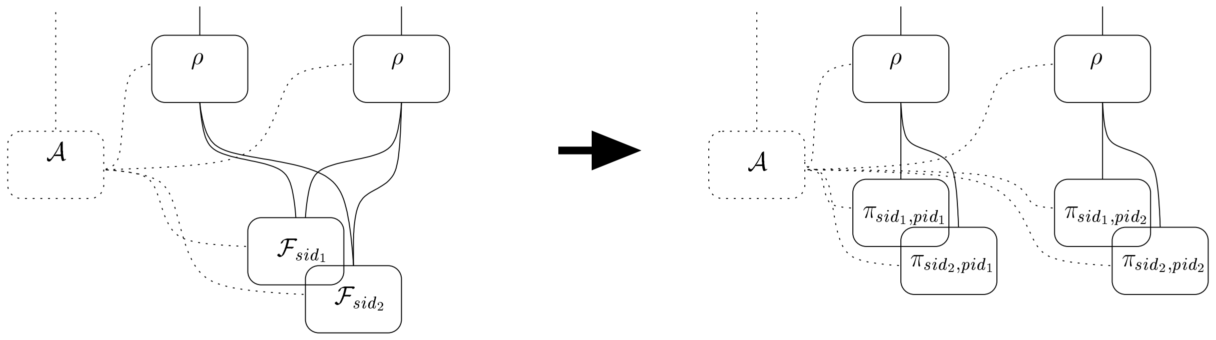

Figure 4: The universal composition operation, for the case where the replaced protocol is an ideal protocol for $\mathcal{F}$. Each instance of $\mathcal{F}$ (left figure) is replaced by an instance of π (right figure). Specifically, for $i = 1, 2$, $\pi_{i,1}$ and $\pi_{i,2}$ constitute an instance of protocol π that replaces the instance $\mathcal{F}_i$ of $\mathcal{F}$. The solid lines represent inputs and outputs. The dotted lines represent communication. The "dummy parties" for $\mathcal{F}$ are omitted from the left figure for graphical clarity.

The composition theorem. In its general form, the composition theorem says that if protocol π UC-emulates protocol ϕ then, for any protocol ρ, the composed protocol $\rho^{\phi\to\pi}$ emulates ρ. This can be interpreted as asserting that replacing calls to ϕ with calls to π does not affect the behavior of ρ in any distinguishable way:

Theorem (universal composition, informal statement): *Let ρ, ϕ, π be protocols such that ρ uses ϕ as subroutine and π UC-emulates ϕ. Then protocol $\rho^{\phi\to\pi}$ UC-emulates ρ.*

A first, immediate corollary of the general theorem states that if protocol π UC-realizes an ideal functionality $\mathcal{F}$, and ρ uses as subroutine protocol $\text{IDEAL}_{\mathcal{F}}$, the ideal protocol for $\mathcal{F}$, then the composed protocol $\rho^{\phi\to\pi^{\mathcal{F}}}$ UC-emulates ρ. Another corollary is that if ρ UC-realizes an ideal functionality $\mathcal{G}$, then so does $\rho^{\phi\to\pi}$

On the proof of the universal composition theorem. We briefly sketch the main ideas behind the proof. Let $\mathcal{A}$ be an adversary that interacts with parties running $\rho^{\phi\to\pi}$. We need to construct an adversary $\mathcal{S}$, such that no environment $\mathcal{E}$ will be able to tell whether it is interacting with $\rho^{\phi\to\pi}$ and $\mathcal{A}$ or with ρ and $\mathcal{S}$. The idea is to construct $\mathcal{S}$ in two steps: First we define a special adversary, denoted $\mathcal{D}$, that operates against protocol π as a stand-alone protocol. The fact that π emulates ϕ guarantees that there exist an adversary ("simulator") $\mathcal{S}_\pi$, such that no environment can tell whether it is interacting with π and $\mathcal{D}$ or with ϕ and $\mathcal{S}_\pi$. Next, we construct $\mathcal{S}$ out of $\mathcal{A}$ and $\mathcal{S}_\pi$.

We sketch the above steps. Adversary $\mathcal{D}$ will be a "dummy adversary" that merely serves as a "channel" between $\mathcal{E}$ and a single instance of π. That is, $\mathcal{D}$ expects to receive in its input (coming from the environment $\mathcal{E}$) requests to deliver messages to prescribed parties of an instance of π. $\mathcal{D}$ then carries out these requests. In addition, any incoming message (from some party of the instance of π) is forwarded by $\mathcal{D}$ to its environment. We are now given an adversary ("simulator") $\mathcal{S}_\pi$, such that no environment can tell whether it is interacting with π and $\mathcal{D}$ or with ϕ and $\mathcal{S}_\pi$.

Simulator $\mathcal{S}$ makes use of $\mathcal{S}_\pi$ as follows. Recall that $\mathcal{A}$ expects to interact with an instance of ρ and multiple instances of the subroutine protocol π, whereas $\mathcal{S}$ interacts with an instance of ρ and instances of ϕ rather than instances of π. Now, $\mathcal{S}$ runs a simulated instance of $\mathcal{A}$ and follows the instructions of $\mathcal{A}$, with the exception that the interaction of $\mathcal{A}$ with the various instances of π is simulated using instances of $\mathcal{S}_\pi$. Here $\mathcal{S}$ plays the role of the environment for the instances of $\mathcal{S}_\pi$. The ability of $\mathcal{S}$ to obtain timely information from the instances of $\mathcal{S}_\pi$, by playing the environment for them, is at the crux of the proof. We also use the fact that $\mathcal{S}_\pi$ must be defined independently of the environment; this is guaranteed by the order of quantifiers. (Some important details, such as how $\mathcal{S}$ routes $\mathcal{A}$'s messages to the various instances of $\mathcal{S}_\pi$, are postponed to the full proof.)

The validity of the simulation is demonstrated via reduction to the validity of $\mathcal{S}_\pi$. Dealing with many instances of $\mathcal{S}_\pi$ running concurrently is done using a hybrid argument: We define many hybrid executions, where the first execution is an execution of ρ with $\mathcal{A}$, and in each successive hybrid execution one more instance of ϕ is replaced with an instance of π. Now, given an environment $\mathcal{E}$ that distinguishes between two successive hybrid executions, we construct an environment $\mathcal{E}_\pi$ that distinguishes between an execution of π with adversary $\mathcal{D}$ and an execution of ϕ with adversary $\mathcal{S}_\pi$. Essentially, $\mathcal{E}_\pi$ orchestrates for $\mathcal{E}$ an entire interaction with ρ, where some of the instances of ϕ are replaced with instances of π, and where one of these instances is relayed to the external system that $\mathcal{E}_\pi$ interacts with. This is done so that if $\mathcal{E}_\pi$ interacts with an instance of π then $\mathcal{E}$, run by $\mathcal{E}_\pi$, "sees" an interaction with one hybrid execution, and if $\mathcal{E}_\pi$ interacts with an instance of ϕ then $\mathcal{E}$ "sees" an interaction with the next hybrid execution. Here we use the fact that an execution of an entire system can be efficiently simulated on a single machine.

The composition theorem can be extended to handle polynomially many applications, namely polynomial "depth of nesting" in calls to subroutines. However, when dealing with computational security (i.e., polynomially bounded environment and adversaries), the composition theorem does not hold in general in cases where the emulated protocol ϕ is not polynomially bounded; indeed, in that case we no longer can simulate an entire execution on a polytime machine.

3 The model of computation

In order to turn the definitional sketch of Section 2 into a concrete definition, one first needs to formulate a rigorous model for representing distributed systems and interacting computer programs within them. This section presents such a model. We also put forth a definition of resource-bounded computation that behaves well within the proposed model. While the proposed model is rooted in existing ones, many of the choices are new. We try to motivate and justify departure from existing formalisms. In order to allow for adequate presentation and motivation of the main definitional ideas, we present the basic model of distributed computation separately from the definitions of security (Section 4). This also simplifies the presentation in subsequent sections.

It is tempting to dismiss the specific details of such a model as being "of no real significance" for the validity and meaningfulness of the notions of security built on top the model. This intuition is perhaps rooted in the Church-Turing thesis, which stipulates that "all reasonable models of computation lead to essentially equivalent notions and essentially the same results". However, at least in the case of distributed, resource-bounded systems with potentially adversarial components, this intuition does not seem to hold. There are numerous meaningful definitional choices to be made on several levels, including the basic modeling of resource-bounded computation and the communication between components, the modeling of scheduling and concurrency, and the representation of the dynamically changing aspects of distributed systems. Indeed, as we'll see, some seemingly small and "inconsequential" choices in the modeling turn out to have significant effects on the meaning of the notions of computability and security built on top of the model.

We thus strive to completely pinpoint the model of computation. When some details do not seem to matter, we explicitly say so but choose a default. This approach should be contrasted with the approach of, say, Abstract Cryptography, or the π-calculus [MR11, M99] that aim at capturing properties that hold irrespective to a specific implementation and its complexity.

Still, we aim to achieve both *simplicity* and *generality.* That is, the model attempts to be general and flexible enough so as to allow expressing the salient aspects of modern distributed systems. At the same time, it attempts to be as simple and clear as possible, with a minimal number of formal and notational constructs. While the model is intended to make sense on its own as a model for distributed computation, it is of course geared towards enabling the representation of security properties of algorithms in a way that is as natural and simple as possible, and with as little as possible reference to the mechanics of the underlying model.

Section 3.1 presents the basic model. Section 3.2 presents the definition of resource-bounded computation. Some alternative formulations are mentioned throughout, as well as in the Appendix.

3.1 The basic model

The main objects we wish to analyze are *algorithms,* or *computer programs,* written for a distributed system. (We often use the term *protocols* when referring to such algorithms.) There are several differences between distributed algorithms and algorithms written for standard one-shot sequential execution: First, a distributed algorithm (protocol) may consist of several programs, where each program runs on a separate computing device, with its own local input, and where the programs interact by exchanging information with each other. We may be interested both in global properties of executions and in properties that are local to program. Also, computations are often *reactive,* namely they allow for interleaved inputs and outputs where each output potentially depends on all prior inputs. Locating and routing information in a distributed system is an additional concern.

Second, different programs may be run by different entities with different interests, and one has to take into account the that the programs run other entities may not be known or correctly represented.

Third, distributed systems often consist of several different protocols (or several instances of the same protocol), all running "at the same time". These protocols may exchange information with each other during their execution, say by providing inputs and outputs to each other, making the separation between different executions more involved.

Fourth, modern systems allow programs to be generated dynamically within one device and "executed" remotely within another device.

The devised model is aimed to account for these and other aspects of distributed algorithms. We proceed in two main steps. First (Section 3.1.1), we define a *syntax,* or a "programming language" for protocols. This language, which extends the notion of *interactive Turing machine* [GMRa89], allows expressing instructions and constructs needed for operating in a distributed system. Next (Section 3.1.2), we define the *semantics* of a protocol, namely an execution model for distributed systems which consist of one or more protocols as sketched above. To facilitate readability, we postpone most of the motivating discussions to section 3.1.3. We point to the relevant parts of the discussion as we go along.

3.1.1 Interactive Turing Machines (ITMs)

An interactive Turing machine (ITM) extends the standard Turing machine formalism to capture a distributed algorithm (protocol). A definition of interactive Turing machines, geared towards capturing *pairs* of interacting machines is given in [GMRa89] (see also [G01, Vol I, Ch. 4.2.1]). That definition adds to the standard definition of a Turing machine a mechanism that allows a *pair* of machines to exchange information via writing on special "shared tapes". Here we extend this formalism to accommodate protocols written for systems with multiple computing elements, and where multiple concurrent executions of various protocols co-exist. For this purpose, we define a somewhat richer syntax (or, "programming language") for ITMs. The full meaning of the added syntax will become clear only in conjunction with the model of execution (Section 3.1.2). Still, we sketch here the main additions.[3]

First, we provide a mechanism for identifying a specific *addressee* for a given piece of transmitted information. Similarly, we allow a recipient of a piece of information to identify the source. Second, we provide a mechanism for distinguishing between different "protocol instances" within a multi-component system.

Third, we provide several types of "shared tapes", in order to facilitate distinguishing between different types of communicated information. The distinction proceeds on two planes: On one plane, we wish to facilitate distinguishing between communication "internal to a protocol instance" (namely, among the participants of the protocol instance) and communication "with other protocol instances". Furthermore, we wish to facilitate distinguishing between communication with "calling protocols" and communication with "subroutine protocols". On another plane, we wish to distinguish between information communicated over a "trusted medium", say locally within the same computing environment, and communication over an "untrusted medium", say across a network.

Definition 1 *An* interactive Turing machine (ITM) *M is a Turing machine (as in, say, [Si05]) with the following augmentations:*

Special tapes *(i.e., data structures)*:

- *An* identity tape. *The contents of this tape is interpreted as two strings. The first string contains a description, using some standard encoding, of the program of M (namely, its transition function). We call this description the* code *of M. The second string is called the* identity *of M. The identity of M together with its code is called the* extended identity *of M. This tape is "read only". That is, M cannot write to this tape.*

[3]As mentioned in the introduction, the use of Turing machines as the underlying computational 'device' is mainly due to tradition. Other computational devices (or, models) that allow accounting for computational complexity of algorithms, such as RAM or PRAM machines, boolean or arithmetic circuits can serve as a replacement. See additional discussion in Section 3.1.3.

- *An* outgoing message tape. *Informally, this tape holds the current outgoing message generated by M, together with sufficient addressing information for delivery of the message.*
- *Three* externally writable *tapes for holding inputs coming form other computing devices (or, processes):*
  - *An* input tape, *representing inputs from the "calling program(s)" or external user.*
  - *An* incoming communication tape, *representing information coming from other programs within the same "protocol instance".*
  - *A* subroutine output tape, *representing outputs coming from programs or modules that were created as "subroutines" of the present program.*
- *A one-bit* activation tape. *Informally, this tape represents whether the ITM is currently "in execution".*

New instructions:

- *An* external write *instruction. Roughly, the effect of this instruction is that the message currently written on the outgoing message tape is possibly written to the specified tape of the machine with the identity specified in the outgoing message tape. Precise specification is postponed to Section 3.1.2.*
- *A* read next message *instruction. This instruction specifies a tape out of* {input, incoming communication, subroutine output}. *The effect is that the reading head jumps to the beginning of the next message. (To implement this instruction, we assume that each message ends with a special* end-of-message (eom) *character.)*[4]

3.1.2 Executing Systems of ITMs

We specify the mechanics of executing a system that consists of multiple ITMs. Very roughly, the execution process resembles the sketch given in Section 2. However, some significant differences from that sketch do exist. First, here we make an explicit distinction between an ITM, which is a "static object", namely an algorithm or a program, and an *ITM instance (ITI),* which is a "run-time object", namely an instance of a program running on some specific data. In particular, the same program (ITM) may have multiple instances (ITIs) in an execution of a system. (Conceptually, an ITI is closely related to a *process* in a process calculus. We refrain from using this term to avoid confusion with other formalisms.)

Second, the model provides a concrete mechanism for addressing ITIs and exchanging information between them. The mechanism specifies how an addresee is determined, what information the recipient obtains on the sender, and the computational costs involved.

Third,the model allows the number of ITIs to grow dynamically in an unbounded way as a function of the initial parameters, by explicitly modeling the "generation" of new ITIs. Furthermore, new ITIs may have dynamically determined programs. Here the fact that programs of ITMs can be represented as strings plays a central role.

Fourth, we augment the execution model with a *control function,* which regulates the transfer of information between ITIs. Specifically, the control function determines which "external write"

[4]This intruction ins not needed if a RAM or PRAM machine is used as the underlying computing unit. See more discussion in Section 3.1.3.

instructions are "allowed". This added construct provides both clarity and flexibility to the execution model: All the model restrictions are expressed explicitly and in "one place." Furthermore, it is easy to define quite different execution models simply by changing the control function.

Systems of ITMs. Formally, a system of ITMs is a pair $S = (I, C)$ where I is an ITM, called the initial ITM, and $C : \{0, 1\}^* \to \{allow, disallow\}$ is a control function.

Executions of systems of ITMs. A configuration of an ITM M consists of the description of the control state, the contents of all tapes and the head positions. (Recall that the program, or the transition function of M is written on the identity tape, so there is no need to explicitly specify the code in a configuration.) A configuration is active if the activation tape is set to 1, else it is inactive.

An instance $\mu = (M, id)$ of an ITM M consists of the program (transition function) of M, plus an identity string $id \in \{0, 1\}^*$. We say that a configuration is a configuration of instance μ if the contents of the identity tape in the configuration agrees with μ, namely if the program encoded in the identity tape is M and the rest of the identity tape holds the string id. We use the acronym ITI to denote an ITM instance.

An activation of an ITI $\mu = (M, id)$ is a sequence of configurations that correspond to a computation of M starting from some *active configuration* of μ, until an inactive configuration is reached. (Informally, at this point the activation is complete and μ is waiting for the next activation.) If a special halt state is reached then we say that μ has halted; in this case, it does nothing in all future activations.

An execution of a system $S = (I, C)$ with input x consists of a sequence of activations of ITIs. The first activation starts from the configuration where the identity tape contains the code of I followed by the identity 0, the input tape contains the value x, and a sufficiently long random string is written on the random tape. In accordance, the ITI $(I, 0)$ is called the initial ITI in this execution.

An execution ends when the initial ITI halts (that is, when a halting configuration of the initial ITI is reached). An execution is accepting if the initial ITI halted in an accepting state. An execution prefix is a prefix of an execution.

To complete the definition of an execution, it remains to specify: (a) The effect of an external-write instruction, and (b) How to determine the first configuration in the next activation, once an activation is complete. These points are described next.

Writing to a tape of another ITI and invoking new ITIs. The mechanism that allows communication between ITIs is the external write instruction. The same instruction is used also for invoking new ITIs. More specifically, the effect of an external write instruction is the following. Let $\mu = (M, id)$ denote the ITI which performs the external write transition, The current contents of the outgoing message tape is interpreted (using some standard encoding) as consisting of μ, followed by an extended identity $\mu' = (M', id')$ of a "target ITI", a tape name out of {input, incoming communication, subroutine output}, and a string m called the message. Then:

1. If the control function C, applied to the current execution prefix, does not allow μ to write to the specified tape of μ' (i.e., it returns a *disallow* value) then the instruction is ignored.

2. If C allows the operation, and an ITI $\mu'' = (M'', id'')$ with identity $id'' = id'$ currently exists in the system (namely, one of the past configurations in the current execution prefix has identity id'), then:

   (a) If the target tape is the incoming communication tape, then the message m is written on the incoming communication tape of μ'', starting at the next blank space. (That is, a new configuration of μ'' is generated. This configuration is the last configuration of μ'' in this execution, with the new information written on the incoming communication tape.) It is stressed that the code M'' of μ'' need not equal the code M' specified in the external write request.

   This convention has the effect that an ITI does not necessarily know the code of the ITI it sends messages to using the communication tape. The recipient ITI learns neither the identity nor the code of the writing ITI. (Of course, this information may be included in the message itself, but the model provides no guarantees regarding the authenticity of this information.) The intuitive goal is to capture the effect of standard communication over an untrusted medium, such as a communication network.

   (b) If the target tape is the input tape or subroutine output tape, and $M'' = M'$, then the specified message is copied to the specified tape of μ'', *along with the code and identity of μ.* If $M' \neq M''$ then the message is not copied and μ transitions to a special error state.

   This convention has the effect that an ITI can verify the code of the ITI to whom it provides input or subroutine output. Furthermore, the recipient of an input or subroutine output knows both the identity and the code of the writing ITI. The intuitive goal here is to capture the effect of communication between processes within a trusted computing environment that allows verification of the code of receiver and sender.

3. If C allows the operation, and no ITI with identity id' exists in the system, then a new ITI μ' with code M' and identity id' is invoked. That is, a new configuration is generated, with code M', the value id' written on the identity tape, and a sufficiently long random string is written on the random input tape. Once the new ITI is invoked, the external-write instruction is carried out as in Step 2. In this case, we say that μ invoked μ'.

On the uniqueness of identities. Section 3.1.3 discusses several aspects of the external-write instruction, and in particular motivates the differences from the inter-component communication mechanisms provided in other frameworks. At this point we only observe that the above invocation rules for ITIs, together with the fact that the execution starts with a single ITI, guarantee that each ITI in a system has unique identity. That it, no execution of a system of ITIs has two ITIs with the same identity, regardless of their codes. This property makes sure that the present addressing mechanism is unambiguous.

Determining the next activated ITI. In order to simplify the process of determining the next ITI to be activated, we allow an ITI to execute at most a single external-write instruction per activation. That is, once an external-write instruction is executed, the activation of the writing ITI completes. The next ITI to be activated is the ITI μ'' whose tapes were written to. The first configuration in the next activation is the last configuration of μ'' in the current execution prefix, with the exception that the activation tape of μ'' is set to 1. If no external-write operation was carried out, or the external write operation was unsuccessful, then the initial ITI is the next one to be activated. See Section 3.1.3 for a discussion on the order of activations of ITMs.

Outputs of executions. We use the following notation. Let $\text{OUT}_{I,C}(x, id_0)$ denote the random variable describing the output of the execution of the system (I, C) of ITMs when I's input is x and identity is id_0. Here the probability is taken over the random choices of all the ITMs in the system.

Extended systems. The above definition of a system of ITMs makes sure that an ITI knows the code of the ITIs to whose input and subroutine output it wrtes, as well as the code of the ITIs that write to its own input and subroutine output taps. This provision is indeed important for the meaningfulness of the model. Still, to facilitate the definition of security, formulated in Section 4, we will need to provide a way to artificially modify the code of the target ITI specified in the external write request, as well as the "code of the source ITI" field in the data written to the target ITI. The mechanism we use to provide this extra flexibility is the control function. That is, we extend the definition of a control function so that it can also *modify* the external-write requests made by parties. Recall that in a system $S = (I, C)$ the control function C takes as input a sequence of external-write requests and outputs either *allowed* or *disallowed.* In an extended system the output of C consists of an entire external-write instruction, which may be different than the input request. The executed instruction is the output of C. We stress that, although the above definition of an extended system gives the control function complete power in modifying the external-write instructions, the extended systems considered in this work use control functions that only modify the *code* of the target and source ITIs as specified in the external write operation.

Subroutines, etc. When an ITI μ writes a message m to the incoming communication tape of ITI μ', we say that μ sends m to μ'. When μ writes a value x onto the input tape of μ', we say that μ passes input x to μ'. When μ' writes x to the subroutine-output tape of μ, we say that μ' passes output x (or simply outputs x) to μ. We say that μ' is a subroutine of μ if μ has passed input to μ' or μ' has passed output to μ in this execution. (Note that μ' may be a subroutine of μ even when μ' was invoked by an ITI other than μ.) If μ' is a subroutine of μ then we say that μ is a parent of μ'. μ' is a subsidiary of μ if μ' is a subroutine of μ or of another subsidiary of μ.

Protocols. A protocol is defined as a (single) ITM as in Definition 1. As already discussed, the goal is to capture the notion of an *algorithm* written for a distributed system where physically separated participants engage in a joint computation; namely, the ITM describes the program to be run by each participant in the computation. If the protocol specifies different programs for different participants, or "roles", then the ITM should describe all these programs. (Alternatively, protocols can be defined as sets, or sequences of machines, where different machines represent the code to be run by different participants. However, such a formulation would add unnecessary complication to the basic model, e.g. to the definition of protocol instances, considered next.)

Protocol instances. The notion of a *running instance* of a protocol has strong intuitive appeal. However, rigorously defining it in way that's both natural and reasonably general turns out to be tricky. Indeed, what would be a natural way to delineate, or isolate, a single instance of a protocol within an execution of a dynamic system where multiple parties run multiple pieces of code?

Traditionally, an instance of a protocol in a running system is defined as a fixed set of machines that run a predefined program, often with identities that are fixed in advance. Such a definitional approach, however, does not account for protocol instances where the number of participants, or perhaps even only their identities, are determined dynamically as the execution unfolds. It also doesnt account for instances of protocols where the code has been determined dynamically, rather than being fixed at the onset of the execution of the entire system. Thus, a more flexible definition is needed.

The definition proposed here attempts to formalize the following intuition: "A set of ITIs in an execution of a system belong to the same instance of some protocol π if they all run π, and in addition they were invoked with the intention of interacting with each other for a joint purpose." In fact, since different participants in an instance are typically invoked within different physical entities in a distributed system, the last condition should probably be rephrased to say: "...and in addition the invoker of each ITI in the instance intends that ITI to participate in a joint interaction with the other ITIs in that instance."

We provide a formal way for an invoker of an ITI to specify which protocol instance this ITI is to participate in. The construct we use for this purpose is the identity string. That is, we interpret (via some standard unambiguous encoding) the identity of an ITI as *two* strings, called the session identifier (SID) and the party identifier (PID). We then say that a set of ITIs in a given execution prefix of some system is an instance of protocol π if all these ITIs have the code π and all have the same SID. The PIDs are used to differentiate between ITIs within a protocol instance; they can also be used to associate ITIs with "clusters", such as physical computers in a network. More discussion on the SID/PID mechanism appears in Section 3.1.3.

Consider some execution prefix of some system of ITMs. Each ITI in a protocol instance in this execution is called a party of that instance. A sub-party of a protocol instance is a subroutine either of a party of the instance or of another sub-party of the instance. The extended instance of some protocol instance includes all the parties and sub-parties of this instance. If two protocol instances I and I' have the property that each party in instance I is a subroutine of a party in instance I' then we say that I is a subroutine instance of I'.

3.1.3 Discussion

This section contains more lengthy discussion that highlights and motivates the main aspects of the model. Indeed, several others general models of distributed computation with concurrently running processes exist in the literature, some of which explicitly aim at modeling security of protocols. A very incomplete list includes the CSP model of Hoare [H85], the CCS model and π-calculus of Milner [M89, M99] (that is based on the λ-calculus as its basic model of computation), the *spi*-calculus of Abadi and Gordon [AG97] (that is based on π-calculus), the framework of Lincoln et. al. [LMMS98] (that uses the functional representation of probabilistic polynomial time from [MMS98]), the I/O automata of Merritt and Lynch [Ly96], the probabilistic I/O automata of Lynch, Segala and Vaandrager [SL95, LSV03], and the Abstract Cryptography model of Maurer and Renner [MR11].

Motivating the use of ITMs. A first definitional choice is to use an explicit, "operational" formalism as the underlying computational model. That is, computation is represented as a sequence of mechanical steps (as in Turing machines) rather than in a functional way (as in the λ-calculus) or in a denotational way (as in Domain Theory). Indeed, while this operational model is less "elegant" and not as easily amenable to abstraction and formal reasoning, it most directly captures the *complexity* of computations. Furthermore, it provides a direct way of capturing the interplay between the complexity of local computation, communication, randomness, and resource-bounded adversarial activity. This interplay is often at the heart of the security of cryptographic protocols.

Moreover, the operational formalism faithfully represents the way in which existing computers operate in a network. Examples include the duality between data and code, which facilitates the modeling of dynamic code generation, transmission and activation ("download"), and the use of a small number of physical communication channels to interact with a large (in fact, potentially unbounded) number of other parties. It also allows considering "low level" complexity issues that are sometimes glossed over in other frameworks, such as the work spent on the addressing, sending, and receiving of messages as a function of the message length or the address space.

Another advantage of using an operational formalism that directly represent the complexity of computations is that it facilitates the modeling of adversarial yet computationally bounded scheduling of events in a distributed system.

Also, operational formalisms naturally allow both for concrete, parametric treatment of security as well as asymptotic treatment that meshes well with computational complexity theory.

Several "operational" models of computations exist in the literature, such as the original Turing machine model, several RAM and PRAM models, and arithmetic and logical circuits. Our choice of using Turing machines is mostly based on tradition, and is by no means essential. Any other "reasonable" model that allows representing resource-bounded computation together with adversarially controlled, resource bounded communication would do.

On the down side, we note that the ITM model, or "programming language" provides a relatively low level abstraction of computer programs and protocols. In contrast, practically all existing protocols are described in a much more high-level (and thus often informal) language. One way to bridge this gap is to develop a library of subroutines that will allow for more convenient representation of protocols as ITMs (or, say, interactice RAM machines). An alternative way is to demonstrate "security preserving correspondences" between programs written in more abstract models of computation and limited forms of the ITMs model, such as the correspondences in [AR00, MW04, CH11, C+05].

On the external-write mechanism. Traditionally, models of distributed computation (such as the ones mentioned above) allow the different components to exchange information via "dedicated named channels". That is, a component can, under various restrictions, write information to, and read information from a "channel name." Channel names are typically treated as fixed "system parameters", in the sense that they are not mutable by the programs running in the system. Furthermore, sending information on a channel is treated as an atomic operation regardless of the number of components in the system or the length of the message. Also, in some models the channel names to be used by each component have to be declared in advance.

This modeling of the communication is clean and elegant. It also facilitates reasoning about protocols framed within that model. In particular, it facilitates analytical operations that separate a system into smaller components by "cutting the channels", and re-connecting the components in different ways. However, this modeling is somewhat abstract and over-restrictive for our purposes. For one, The fact that the channel names are fixed in advance does not allow for effective addressing in settings where the number and makeup of components changes as the system evolves. Also, it does not allow for simple representation of protocols where the addressing mechanism is created within the protocol itself, as is often the case in realistic protocols. It also doesn't account for the cost of message addressing and delivery; in a dynamically growing systems this complexity may be an important factor. Finally, it does not account for dynamic generation of new programs.

The external-write mechanism, together with the control function, is aimed at providing a sufficiently low-level and flexible modeling that allows taking into account the above issues. First, they allow the analyzed protocols to generate the identities of participants in an algorithmic way, as part of the execution. They also provide a more dynamic and flexible extension of the static notions of "communication channels" or "ports" used in other models. Also, Requiring an ITI to explicitly write the recipient address and the message on a special tape allows considering the associated computational overhead. The only abstraction provided by this mechanism is the guarantee of global uniqueness of identities. Indeed, this guarantee is essential for unambiguous addressing. See more discussion on this point later on in this section. Below we highlight two additional aspects of the external write mechanism.

Letting the addressing depend on the code. When writing to input or subroutine output tapes, the external write mechanism provides the writing party with the guarantee that the message is written only if the target ITI runs the *code* (program) specified by the writing party. Furthermore, in these cases the recipient ITI learns the code of the writing party. Such an option is not explicitly given in other models. Indeed, at first glance it may seem odd: Why should the sender and recipient gain such information or control?

We first argue that, even when modeling systems that interact with untrusted entities, it is necessary for effective modeling to give a program the ability to verify (and sometimes specify) the program run by the processes it provides input to or gives output to. Similarly, it is necessary to give a program the ability to verify the program run by the entity that provides it with input or with "subroutine output". Indeed, such modeling is needed to capture situations where the partition of a program to subroutines is only logical, or when different programs run in a trusted computing environment.

Now, in models that only capture systems where the components and communication channels between them are fixed and known in advance, such a guarantee can be provided simply by allowing communication only over a pre-specified set of communication channels. However, in the present modeling, where components and identities are generated dynamically, and no static notions of "connections" or "channels" exist, an alternative mechanism is needed for verifying the code run by the communicating peer. The mechanism we employ here is to explicitly specify the code of the recipient ITI and disclose the code of the writing ITI.

It is stressed that in order to make the above mechanism meaningful, programs need to be written in ways that are "recognizable" by the peers. This can be done using standard encoding mechanisms. Indeed, a peer may accept one representation of a program, and reject another representation, even though the two might be functionally equivalent. Alternatively, programs may be written in a way that allows the peer to verify some desired properties. (For instance, a recipient may decide to ignore the received data unless the sending ITI's code provides some basic guarantees which appropriately restrict its behavior.)

See the formulation of $\mathcal{F}_{\text{AUTH}}$, the ideal authentication functionality, in Section 6.3, for an example of the use of this mechanism.

Invoking new ITIs. Allowing for dynamic invocation of new ITIs is important for modeling realistic situations where parties may join a computation as it unfolds, and where the number of protocol instances that run in the systems is not known in advance. (Indeed, such situations impose security requirements on protocols, that are not naturally expressible in a model where the number of components is fixed in advance. One example is the study of concurrent Zero-Knowledge, where the number of sessions depends on the adversary and cannot be bounded by any fixed polynomial. See e.g. [R06].) Similarly, we would like to be able to model commonplace situations where programs are generated automatically, "downloaded", and incorporated in a computation "on the fly". The external write mechanism provides a flexible way for incorporating in the computation new programs, as well as new instances of existing programs.

Two additional remarks are in order here. First, it may seem at first glance that the ability to dynamically generate arbitrary new code is not needed, since it suffices to consider only ITIs that run the code of a "universal Turing machine", and then provide these ITIs with appropriate code to be run as part of the input. However, we argue that if such a generic convention were adopted then ITIs would not be able to effectively verify that their communicating peers are running some acceptable programs; the necessity of such a mechanism was argued in the previous remark. It is thus important to allow newly generated ITIs to have code with some verifiable properties.

Second, recall that the invocation of a new ITI is implicit, i.e. it occurs only when an existing ITI attempts to write to a tape of a non-existing ITI. Furthermore, the writing ITI does not learn whether a new ITI was invoked. We adopt this convention since it seems natural; furthermore, it simplifies the model, the definition of security, and subsequently the presentation and analysis of protocols. Still, it is not essential: Once could, without significant effect on the expressibility of the model, add an explicit "instance invocation" operation and require that an ITI is invoked before it is first activated. In this case, however, a different mechanism for guaranteeing uniqueness of identities would be needed.

On the distinction between input, communication, and subroutine output tapes. The definition of ITMs provides three syntactically different methods to transfer information from one ITI to another: either via the input tape, via the incoming communication tape, or via the subroutine output tape. This distinction is used for a number of different purposes. A first use is to facilitate the distinction between various types of incoming information in a multi-party, multi-protocol, multi-instance system, and in particular to make explicit the notion of subroutines (which is central for the universal composition operation and theorem). Here the input tape is used to model information coming from a "calling program," namely a program that uses the present program as a subroutine. The communication tape is used to model information coming from "peers", or other programs within the same protocol instance. The subroutine output tape is used to model information coming from "subroutines" of the present program. This interpretation of the different tapes is depicted in Figure 2.1 on page 9.

A second use (discussed earlier in the context of the external write instruction) is to facilitate the distinction between "trusted" and "untrusted" communication, as expressed in the different semantics of the external write instruction to different tapes. (See also the discussion on

A third use is made by the definition of resource-bounded computation (Section 3.2). There, only information written on the input tape is used in the calculation of the allowed runtime.

These three interpretations, or "attributes" of the different communication tapes are, in principle, orthogonal to each other. The specific "bundling" of attributes with tapes is rather ad hoc, and is done for convenience only. For instance, the fact that the same tape is used to represent communication with peers and also to represent an untrusted medium is not essential. It only makes sense since in typical systems the various parties in a protocol instance are physically separated and connected via a network. Similarly, the fact that the tape used to represent information coming from a subroutine program also models trusted communication is not essential, and merely represents typical systems of interest. A more general modeling would allow for full separation of these interpretations, or "attributes" of tapes. That is, each external write instruction would specify, in addition to the tape to be written to, whether the code of the receiver is to be verified as a precondition to the writing operation, whether the identity and code of the sender is to be included, and whether the incoming data should be used towards determining the allowed runtime of the recipient. In the interest of simplicity, we choose not to provide such level of generality.

Jumping to the next received message. Recall that Definition 1 allows an ITM to move, in a single instruction, the reading head on each of the three incoming data tapes to the beginning of the next incoming message. At first, this instruction seems superfluous: Indeed, why not let the ITM simply move the head in the usual way, namely cell by cell?

The reason is that such an instruction becomes necessary in order to maintain a reasonable notion of resource-bounded computation in a heterogeneous and untrusted network, where the computational powers of participants vary considerably, and in addition some participants may be adversarial. In such a system, powerful participants may try to "overwhelm" less powerful participants by simply sending them very long messages. In reality, such an "attack" can be easily thwarted by having parties simply "drop" long messages, namely abort attempt to interpreted incoming messages that become too long. However, without a "jump to the next message" instruction, the ITM model does not allow such an abortion, since the reading head must be moved to the next incoming message in a cell-by-cell manner. (There are of course other ways in which powerful parties may try to "overwhelm" less powerful ones. But, with respect to these, the ITM model seems to adequately represent reality.)

We remark that the above discussion exemplifies the subtleties involved with modeling systems of ITMs. In particular, the notions of security in subsequent sections would have different technical meaning without the ability to jump to the beginning of the next incoming message. (In contrast, in a RAM machine model, such a provision would not be necessary.) A similar phenomenon has been independently observed in [P06] in the context of Zero-Knowledge protocols.

Making the identities available to the code. In contrast with other formalisms (such as [DKMR05, K06, KT13]), The present formalism allows ITMs to read and use their identities, which are guaranteed by the model to be globally unique. Indeed, providing parties with identities that are guaranteed to be unique is a strong and potentially unrealistic guarantee. Still, in some cases having unique identities available to the protocol is *essential* for a meaningful solution. (This fact is exemplified in [LLR02] for the basic tasks of broadcast and Byzantine agreement.)

We provide unique identities in order to facilitate representing protocols that used identities. Still, it is of course possible to study within the present framework protocols that do not use the identities given in the model, by explicitly considering only protocols that ignore the identities.

Finally note that there are a number of practical methods for guaranteeing global uniqueness of identities. One such way is to have each identity contain a component that's chosen at random from a large enough domain. Alternatively, one can use a hierarchical encoding where each new identity is pair (invoker ID, new ID). (Indeed, such a mechanism is mandated in [HS11].) These methods can be regarded as ways to implement the abstraction of unique identities.

On the SID mechanism. As argued in the preamble to the definition of protocol instances (Section 3.1.2), the SID mechanism provides a relatively simple and flexible way to delineate protocol instances in a dynamically changing system. It also allows capturing, within the formal model, the act of "creating an instance of a protocol" in a dynamic and distributed way.

One basic aspect of this act is that some sort of agreement or coordination between the entities that invoke the participants of a protocol instance is needed. The SID mechanism postulates that this agreement take the form of agreeing on a joint identifier, namely the SID. We briefly sketch three common methods for reaching such agreement, and then point out some possible relaxations.

One method is to have the SID of the instance determined in advance (potentially as a function of the SID of the calling protocol and other run-time parameters). This method is natural when there is already some prior coordination between the entities that invoke the participants of a protocol instance, say when the protocol instance in question is a subroutine in a larger protocol.

A second method is to design the protocol so that all the ITIs in a protocol instance (except for the first one) are invoked via incoming messages from other ITIs in that instance itself, rather than via inputs from other protocol instances. That is, the first activation of each party occurs due to a message from another party of that protocol instance, rather than due to an input or a message from another protocol instance. Furthermore, the invoking ITI will set the SID of the invoked ITI to be the same as its own SID. This way, a multi-party instance of a protocol is created *without any prior coordination.* (The ideal authentication and secure message transmission functionalities from Section 1.3 are written in this manner, thus eliminating the need for prior agreement on the SID, and allowing realization by non-interactive protocols.) We note that real-life implementation of protocols that are written in this manner would require each participant to locally make sure that no two protocol sessions it participates in have the same SID.

A third alternative is to run some agreement protocol among the parties in order to determine a joint SID. This method is viable in situations where there is no prior coordination among the entities that invoke the various ITIs in a protocol instance, and the parties wish to jointly determine the SID. (See [BLR04, B$^+$11] for a protocol and more discussion.) In this case, one convenient way to determine the SID of a protocol instance is to let it be the concatenation of the PIDs of some or all of the parties in this instance.

We remark that it is possible to formulate alternative conventions that allow the SIDs of the parties in a protocol instance to be related in some other way, rather than being equal. Such a more general convention may allow more loose coordination between the ITIs in a protocol instance. (For instance, one may allow the participants to have different SIDs, and only require that there exists some global function that, given a state of the system and a pair of SIDs, determines whether these SIDs belong to the same instance.) Also, SIDs may be allowed to change during the course of the execution. However, such mechanisms would further complicate the model, and the extra generality obtained does not seem essential for our treatment.

Finally we note that other frameworks, such as [HS11] put additional restrictions on the format of the SIDs. Specifically, the SID of a protocol instance is required to include the SID of the calling protocol instance. While this is a convenient convention in many cases, it is rather limiting in others. Furthermore, the main properties of the model hold regardless of whether this convention is adhered to.

On "true concurrency" and the order of activations. Recall the order of activations of ITMs in an execution of a system: Once an ITI μ completes its activation, the (single) ITI on whose tapes μ wrote is activated next; if μ didn't write to any other ITI then the initial ITI is activated.

One might wonder whether this simple and sequential order adequately represents concurrent systems. Indeed, the model stands in contrast with the physical nature of distributed systems, where computations take place in multiple physically separate places at the same time. It also stands in contrast with other models of concurrent and distributed computation, which represent concurrent (typically non-deterministic) scheduling already within the basic model of computation.

We claim however that this order does provide a sound basis for the study of concurrent systems in general, and their security properties in particular. First we claim that, as long as individual activations represent computations that are relatively short comparing to the communication time, this sequential execution model actually provides "pseudo-concurrency" that is a reasonable approximation of "true concurrency".

More substantially, within the present framework, the "unpredictable" or "non-deterministic" nature of the communication among concurrently running physically separate processes is captured only at a higher level, namely as part of the actual model of protocol execution with an environment and adversary. (This model was sketched in Section 2 and will be described in more detail in Section 4.) That is, the requirement that protocols should withstand multiple interleavings of local computations is expressed within the context of adversarial scheduling of (small execution pieces of) such computations.

In particular, in spite of its simplicity, the present model is actually sufficient for representing liveness, synchrony and fairness properties. This is exemplified in Section 6. Furthermore, combining the scheduling together with the other adversarial activities allows for representing adversarial scheduling which, on the one hand, is computationally bounded, and on the other hand can adaptively depend on the adversarial view of the execution so far. Such modeling is essential for capturing the security of cryptographic protocols.

Some models that combine non-deterministic treatment of concurrency with cryptographic modeling are discussed in the Appendix.

The control function as an ITM. The control function is a convenient abstraction: First, as argued earlier, the control function can be regarded as a generalization of the more traditional notion of fixed "communication channels" or "ports". Second, the control function allows separating the definition of the basic communication model from definitions of security which are built on top of such a model. In particular, the definition of security (Section 4) defines the model for protocol execution by specifying a specific control function. Third, the control function can be used to represent models where the order of activations is different than here, such as [BPW04, DKMR05, CV12].

We note that an alternative and equivalent formulation of a system of ITMs might replace the control function by a special-purpose "router ITM" C that controls the flow of information between ITIs. Specifically, in this formulation the external input to the system is written on the input tape of C. Once activated for the first time, C copies its input to the input tape of the initial ITM I. From now on, all ITIs are allowed to write only to the incoming communication tape of C, and C is allowed to write to any externally writable tape of anther ITI. In simple (non-extended) systems, C always writes the requested value to the requested tape of the requested recipient, as long as the operation is allowed. In extended systems, C may write arbitrary values to the externally writable tapes of ITIs.

Deleting ITIs. The definition of a system of ITMs does not provide any means to "delete" an ITI from the system. That is, once an ITI is invoked, it remains present in the system for the rest of the execution, even after it has halted. In particular, its identity remains valid and "reserved" throughout. If a halted ITI is activated, it performs no operation and the initial ITI is activated next. The main reason for this convention is to avoid ambiguities in addressing of messages to ITIs.

3.2 Polynomial time ITMs and systems; Parameterized systems

We adapt the standard notions of "resource bounded computation" to the distributed setting considered in this work. This requires accommodating systems with dynamically changing number of components and communication patterns, and where multiple protocols and instances thereof co-exist. As usual in cryptography, where universal statements on the capabilities of *any feasible computation* are key, notions of security depend in a strong way on the precise formulation of resource bounded computation. As we'll see, current formulations do not behave well in a dynamically changing distributed setting such as the one considered in this work. We thus propose an extension that seems adequate within the present model.

Before proceeding with the definition itself, we first note that the notion of "resource bounded computation" is typically used for two quite different purposes. One is the study of *efficient algorithms.* Here we'd like to examine the number of steps required as a function of the complexity of the input, often interpreted as the input length. Another purpose is bounding the power of *feasible computation,* often for the purpose of security. Here we typically do not care whether the computation is using "efficient algorithms"; we are only concerned with what can be done within the given resource bounds.

At first glance it appears that for security we should be primarily interested in the second interpretation. However, recall that to argue security we often prove an algorithmic reduction that translates an attacker against the scheme in question to an attacker against some underlying construct that's assumed to be secure. This reduction should be efficient *in the former, algorithmic sense.* Furthermore, the very definition of security, formulated later, will require presenting an *efficient transformation* from one *feasible computation* to another. We conclude that a good model should allow capturing both interpretations.

Let $T : \mathbf{N} \to \mathbf{N}$. Traditionally, a Turing machine M is said to be T-bounded if, given any input of length n, M halts within at most $T(n)$ steps. There are several ways to generalize this notion to the case of ITMs. One option is to require that each activation of the ITM completes within $T(n)$ steps, where n is either the length of the current incoming message, or, say, the overall length of incoming messages on all externally writable tapes to the ITM. However, this option does not bound the overall number of activations of the ITM; this allows a system of ITMs to have unbounded executions, thus unbounded "computing power", even when all its components are resource bounded. This does not seem to capture the intuitive concept of resource bounded distributed computation.

Another alternative is then to let $T(n)$ bound the overall number of steps taken by the ITM since its invocation, regardless of the number of activations. But what should n be, in this case? One option is to let n be the overall length of incoming messages on all externally writable tapes of the

ITM. However, this would still allow a situation where a system of ITMs, all of whose components are $T(n)$-bounded, consumes an unbounded number of resources. This is so since ITIs may send each other messages of ever increasing lengths. In [GMRa89] this problem was solved by setting n to be the length of the input only. Indeed, in the [GMRa89] setting, where ITMs cannot write to input tapes of each other, this solution is adequate. However, in our setting no such restrictions exist; thus, when n is set to the overall length of the input received so far, infinite runs of a systems are possible even if all the ITIs are $T(n)$-bounded. Furthermore, infinite "chains" of ITIs can be created, where each ITI in the chain invokes the next one, again causing potentially infinite runs.

We prevent this "infinite runs" problem via the following simple mechanism: We define n to be the overall length of the input received so far, i.e. the number of bits written on the ITI's input tape, *minus the overall number of bits written by the ITI to the input tapes of other ITIs.* As we'll see, this provision allows guaranteeing that, for all "reasonable" functions T, the overall number of steps taken in a system of ITMs which are all T-bounded is finite. In fact, this number is bound by $T(n)$, where n is the length of the initial input to the system. Intuitively, this provision treats the characters written on the input tape as "tokens" that "buy" runtime. An ITI receives tokens on its input tape, and gives out tokens to other ITI by writing on their input tapes. This way, it is guaranteed that the number of tokens in the system remains unchanged, even if ITIs are generated dynamically and write on the input tapes of each other.

In Section 3.2.1 we briefly mention some other potential formulations and their shortcomings.

Definition 2 (T-bounded, PPT) *Let $T : \mathbf{N} \to \mathbf{N}$. An ITM M is* locally T-bounded *if, at any point during an execution of M (namely, in any configuration of M), the overall number of computational steps taken by M so far is at most $T(n)$, where $n = n_I - n_O$, n_I is the overall number of bits written so far on M's input tape, and n_O is the number of bits written by M so far to input tapes of ITM instances.*

If M is locally T-bounded, and in addition either M does not make external write requests, or each external write request specifies a recipient ITM which is T-bounded, then we say that M is T-bounded.

M is PPT *if there exists a polynomial p such that M is p-bounded. A protocol is PPT if it is PPT as an ITM.*

Recall that $T : \mathbf{N} \to \mathbf{N}$ is super-additive if $T(n + n') \geq T(n) + T(n')$ for all n, n'. We have:

Proposition 3 *Let $T : \mathbf{N} \to \mathbf{N}$ be a super-additive increasing function. If the initial ITM in a system (I, C) of ITMs is T-bounded, and in addition the control function C is computable in time $T'(\cdot)$, then an execution of the system can be simulated on a single (non-interactive) Turing machine M, which takes for input the initial input x, and runs in time $O(T(|x|)T'(T(|x|)))$. In particular, if both I and C are PPT then so it M. The same holds also for extended systems of ITMs, as long as all the ITMs invoked are T-bounded.*

Proof: We first claim that the overall number of configurations in an execution of a system (I, C) where I is T-bounded is at most $T(|x|)$, where x is the initial input of I. As mentioned above, this can be seen by treating the bits written on the input tapes of ITIs as "tokens" that give runtime. Initially, there are $|x|$ tokens in the system. The tokens are "passed around" between ITIs, but their number remains unchanged throughout. More formally, recall that an execution of a system of ITMs consists of a sequence of activations, where each activation is a sequence of configurations of the active ITI. Thus, an execution is essentially a sequence of configurations of ITIs. Consider an execution, and et m_i be the set of ITIs that were active up till the ith configuration in the execution. For each $\mu \in m_i$ let $n_{\mu,i}$ be number of bits written on the input tape of ITI μ at the last configuration where it was active before the ith configuration in the execution, minus the overall number of bits written by μ to other ITIs in all previous configurations. Since I is T-bounded we have that μ is also T-bounded, namely for any i, the number of steps taken by each $\mu \in m_i$ is at most $T(n_{\mu,i})$. It follows that $i = \sum_{\mu \in m_i}(\#\text{ steps taken by }\mu) \leq \sum_{\mu \in m_i} T(|n_{\mu,i}|)$. By super-additivity of T we have that $i \leq \sum_{\mu \in m_i} T(n_{\mu,i}) \leq T(\sum_{\mu \in m_i} n_{\mu,i})$. However, $\sum_{\mu \in m_i} n_{\mu,i} \leq |x|$. Thus $i \leq T(|x|)$.

The machine M that simulates the execution of the system (I, C) simply writes all the configurations of $I, C)$ one after the other, until it reaches a halting configuration of I. It then accepts if this configuration accepts. To bound the runtime of M, it thus remains to bound the time spent on evaluating the control function C. However, C is evaluated at most $T(|x|)$ times, on inputs of length at most $T(|x|)$ each. The bound follows. □

We note that the control functions of all the systems in this work run in linear time.

Parameterized systems. The definition of T-bounded ITMs guarantees that an execution of a system of bounded ITMs completes in bounded time. However, it does not provide any guarantee regarding the relative computing times of different ITMs in a system. To define security of protocols we will want to bound the variability in computing power of different ITMs. To do that, we assume that all parties know a common value, called the security parameter, that will be taken into account when determining the allowed runtime. More specifically, we say that a system of ITMs is parameterized with security parameter k if the following two conditions hold: First, at any external write operation where a new ITI is invoked, the value 1^k is first written on the input tape of the invoked ITI. (This guarantees that each invoked ITI has some minimum runtime. It also This provides some basic coordination between the parties regarding the desired "level of security".)

Finally we note that subsequent sections will concentrate on the behavior of systems when the length of the initial input is at most some function of (specifically, polynomial in) the security parameter.

3.2.1 Discussion

We discuss some aspects of Definition 2 and mention some related definitional approaches.

Recognizing PPT ITMs. One general concern regarding notions of PPT Turing machines is how to decide whether a given ITM is PPT. Of course, it is in general undecidable whether a given ITM is PPT. The standard way of getting around this issue is to specify a set of rules on encodings of ITMs such that: (a) it is easy to verify whether a given string obeys the rules, (b) all strings obeying these rules encode PPT ITMs, and (c) for essentially any PPT ITM there is a string that encodes it and obeys the rules. If there exists such a set of rules for a given notion of PPT, then we say that the notion is *efficiently recognizable.*

It can be readily seen that the notion of PPT in Definition 2 is efficiently recognizable. Specifically, an encoding σ of a *locally* PPT ITM will first specify an exponent c. It is then understood that the ITM encoded in σ counts its computational steps and halts after n^c steps. An encoding of a PPT ITM will guarantee in addition that that all the codes specified by the external write operations are also $n^{c'}$-bounded with an exponent $c' \leq c$. These are simple conditions that are straightforward to recognize. We note that other notions of PPT protocols, such as those in [HMU09, HS11] are not known to be efficiently recognizable.

Imposing an overall bound on the running time. Recall that it does not suffice in of itself to simply bound the runtime of each individual activation of an ITI by some function of the length of the contents of the externally writable tapes. This is so since, as discussed prior to Definition 2, we might still have unbounded executions of systems even when all the ITMs are bounded. Definition 2 gets around this problem by making a restriction on the *overall* number of steps taken by the ITI so far. An alternative approach might be to directly impose an overall bound on the runtime of the system. For instance, one can potentially bound the overall number of bits that are externally written in the execution. This approach seems attractive at first since it is considerably simpler; it also avoids directly "linking" the runtime in an activation of an ITM to the run-times in previous activations of this ITM. However this approach has a severe drawback: It causes an execution of a system to halt at a point which is determined by the overall number of steps taken by the system, rather than by the local behavior of the last ITI to be activated (namely the initial ITI). This provides an "artificial" way for the initial ITI to obtain global information on the execution via the timing in which the execution halts. (For instance, the initial ITI I can start in a rejecting state, and then pass control to another ITI μ. If I ever gets activated again, it moves to an accepting state. Now, whether I gets activated again depends only on whether the computation carried out by μ, together with the ITIs that μ might have invoked, exceeds the allotted number of steps, which in turn may be known to I. Thus, we have that whether I accepts depends on information that should not be "legitimately available" to I in a distributed system.) Jumping ahead, we note that this property would cause the notions of security considered in the rest of this work to be artificially restrictive. Specifically, the environment would now be able to distinguish between two executions as soon as the overall number of steps in the two executions differs even by one operation. In contrast, we would like to consider two systems equivalent from the point of view of the environment even in cases where the overall number of computational steps and communicated bits in the two systems might differ by some polynomial amount.

Bounding the runtime by a function of the security parameter alone. Another alternative way to define resource bounded ITMs is to consider parameterized systems as defined above, and then restrict the number of steps taken by each ITI in the computation by a function of the security parameter *alone.* That is, let the overall number of steps taken by each ITI in the system be bounded by $T(k)$, where k is the security parameter. This formulation is actually quite popular; In particular, it is the notion of choice in earlier versions of this work as well as in [C00, PW00, BPW04, BPW07, MMS03, C+05].

Bounding the runtime this way is simpler than the method used here. It also allows proving a proposition akin to Proposition 3. However, it has a number of drawbacks. First, it does not allow capturing algorithms and protocols which work for any input size, or alternatively work for any number of activations. For instance, any signature scheme that's PPT in the security parameter alone can only sign a number of messages that's bounded by a fixed polynomial in the security parameter. Similarly, it can only sign messages whose length is bounded by a fixed polynomial in the security parameter. In contrast, standard definitions of cryptographic primitives such as signature schemes, encryption schemes, or pseudorandom functions require schemes to handle a number of activations that's determined by an arbitrary PPT adversary, and thus cannot be bounded by any specific polynomial in the security parameter. Consequently, bounding the runtime by a fixed function of the security parameter severely restricts the set of protocols and tasks that can be expressed and analyzed within the framework.[5]

Furthermore, when this definition of bounded computation is used, security definitions are inevitably weaker, since the standard quantification over "all PPT adversaries" fails to consider those adversaries that are polynomial in the length of their inputs but not bounded by a polynomial in the security parameter. In fact, there exist protocols that are secure against adversaries that are PPT in the security parameter, but insecure against adversaries that are PPT in the length of their inputs (see e.g. the separating example in [HU05]).

Another drawback of bounding the runtime by a fixed function of the security parameter is that it does not allow taking advantage of the universality of computation and the duality between machines and their encodings. Let us elaborate, considering the case of PPT ITMs: When the runtime can vary with the length of the input, it is possible to have a single PPT ITM U that can "simulate" the operation of *all* PPT ITMs, when given sufficiently long input. (As the name suggests, U will be the universal Turing machine that receives the description of the ITM to be simulated, plus sufficiently long input that allows completing the simulation.) This universality is at the heart of the notion of "feasible computation". Also, this property turns out to be useful in gaining assurance in the validity of the definition of security, defined later in this work.

Bounding the runtime of ITMs by a function of the security parameter alone does not seem to allow for such a natural property to hold. Indeed, as discussed in Section 4.4, some of the properties of the notion of security defined here no longer hold when the runtime of ITMs is bounded this way.

A more general notion. The present notion of resource bounded ITMs treats all the bits on the input tape, and only those bits, as "resource tokens". A more general notion of resource bounded ITMs may allow *any* external write operation to provide any number of "runtime tokens" to the target ITI by explicitly specifying as much. (Of course, any "runtime tokens" transferred to the target ITI would be deducted from the writing ITI.) This more general notion gives somewhat more expressibility to the model. However, we stick with the present notion since it is simpler, and furthermore it is closer to existing notions of resource bounded computation (which also measure runtime as a function of the input length).

Thanks. We thank Oded Goldreich, Dennis Hofheinz, Ralf Küsters, Yehuda Lindell, Jörn Müller-Quade, Rainer Steinwandt and Dominic Unruh for very useful discussions on modeling PPT ITMs and systems, and for pointing out to us shortcomings of the definition of PPT ITMs in earlier versions of this work and of some other definitional attempts. Discussions with Dennis were particularly instructive.

[5]We remark that the difference is not only "cosmetic." For instance, pseudorandom functions with respect to a number of queries that is bounded by a fixed polynomial in the security parameter can be constructed without computational assumptions, whereas the standard notion implies one-way functions.

4 Defining security of protocols

This section presents the main definition of this work, namely the definition of protocols that securely realize a given ideal functionality, as outlined in Section 2.2. Section 4.1 presents the basic computational model for executing distributed protocols. The general notion of protocol emulation and some variants are presented in Section 4.2. Ideal functionalities and the ideal protocol for carrying out a given functionality are presented in Section 4.3, followed by the definition of securely realizing an ideal functionality. Finally, Section 4.4 presents several alternative formalizations of the definition of protocol emulation and demonstrates their equivalence to the main one.

4.1 The model of protocol execution

The model of protocol execution is defined in terms of a system of ITMs, as defined in Section 3. The model gives the adversary complete control over the communication between parties. Ths modeling is not aimed at representing some specific realistic setting for distributed computation. Rather, the goal is to provide a basic platform on top of which one can define various communication models that correspond to realistic settings. Several such communication models are defined in Section 6.

The model presented here differs from the informal model of protocol execution in Section 2.2 in two main ways. First, here we do not model party corruption within the basic model; instead, party corruptions are modeled at a later stage via special protocol conventions, thus simplifying the basic model and providing extra flexibility in defining various types of corruption. Second, here we need to accommodate the fact that parties (ITIs) expect to know the identity and code of the ITIs they provide input and output to, and obtain input or subroutine output from. This makes the incorporation of the environment machine in the model a bit more tricky: ITIs representing computing elements in an actual system never give output to or get input from an "environment ITI"; they only get input or give output to actual other ITIs. Thus the model should "hide" the existence of the environment from the ITIs representing the protocol. Similarly, it needs to allow substituting one protocol for another without the environment knowing that such substitution occurred. (Jumping ahead, we note that the universal composition operation cannot be meaningfully defined without appropriate modeling of this fact.) These issues are handled via the control function, as described below.

The model is parameterized by three ITMs: the protocol π to be executed, an environment $\mathcal{E}$ and an adversary $\mathcal{A}$. That is, given $\pi, \mathcal{E}, \mathcal{A}$, the model for executing π is the following extended, parameterized system of PPT ITMs $(\mathcal{E}, C^{\pi,\mathcal{A}}_{\text{EXEC}})$, as defined in Section 3. The initial ITM in the system is the environment $\mathcal{E}$. The control function $C^{\pi,\mathcal{A}}_{\text{EXEC}}$ is defined in the paragraphs below. Figure 5 presents a summary of the model. A graphical depiction appears in Figure 2 on page 11.

The input of the initial ITM $\mathcal{E}$ represents some initial state of the environment in which the protocol execution takes place. In particular, it represent all the external inputs to the system, including the local inputs of all parties. The first ITI to be invoked by $\mathcal{E}$ is set by the control function to be $\mathcal{A}$.

In addition, as the computation proceeds, $\mathcal{E}$ can invoke and pass inputs to an unlimited number of ITIs, subject to the restriction that all these ITIs have the same SID (which is chosen by $\mathcal{E}$). More precisely, external write operations by $\mathcal{E}$ are required to be of the form $(v, id_{source}, id_{target})$, where v is an input value, $id_{source} = (pid_{source}, sid_{source})$ is an extended identity representing the claimed source of the input value, and $id_{target} = (pid_{target}, sid_{target})$ is an identity for the target

Execution of protocol π with environment $\mathcal{E}$ and adversary $\mathcal{A}$

An execution of protocol π with adversary $\mathcal{A}$ and environment $\mathcal{E}$ is a run of an extended, parameterized system of ITMs as specified in Section 3, with initial ITM $\mathcal{E}$, and a control function that enforces the following restrictions:

1. $\mathcal{E}$ may only *pass inputs* to other parties. The first ITI that $\mathcal{E}$ passes input to is set to be the adversary: the identity of this ITI is required to have a special value, say $id = 1$, and the code of that ITI is set (by the control function) to be $\mathcal{A}$.

   All the other ITIs that $\mathcal{E}$ passes input to are required to be an instance of protocol π. More precisely, any external write operation by $\mathcal{E}$ (other than the first one) is required to be of the form $(v, id_{source}, id_{target})$, where v is an input value, id_{source} is an extended identity representing the claimed source of the input value, and id_{target} is an identity for the target ITI. If the SID s in id_{target} is the same as the SID of the target identity in previous external write instructions of $\mathcal{E}$, and the SID in id_{source} is different than s, then the message m with extended source id_{source} is written to the input tape of an ITI with code π and identity id_{target}. If no such ITI exists then one is created.

2. The adversary $\mathcal{A}$ can write to any tape of any ITI. There are no restrictions on the contents and sender identities of delivered messages.

3. ITIs other than $\mathcal{E}$ or $\mathcal{A}$ (including the parties invoked by $\mathcal{E}$ and $\mathcal{A}$ and their subsidiaries) can send messages to $\mathcal{A}$, and also pass inputs and outputs to any ITI other than $\mathcal{A}$, $\mathcal{E}$. If the extended identity of the target ITI of an external write request coincides with the claimed source identity value id_s in one of the previous external write requests of $\mathcal{E}$, and the target tape is the subroutine output tape, then the control function writes the requested message on the subroutine output tape of $\mathcal{E}$, with the code of the writing ITI removed. (That is, $\mathcal{E}$ sees the identity but not the code of the ITI that writes to its subroutine output tape.)

Figure 5: A summary of the model for protocol execution

ITI. The control function then verifies that sid_{target}, the SID in id_{target}, is the same as the SID of the target ITI in previous external write instructions of $\mathcal{E}$, and that sid_{source}, the SID in id_{source}, is different than sid_{target}. It then writes message m with extended source id_s to the input tape of an ITI with code π and identity $id_t = (p, s)$. If no such ITI exists then one is created.[6]

The adversary $\mathcal{A}$ can write to any tape of any ITI in the system. If $\mathcal{A}$ writes to the incoming communication tape of an ITI then we say that $\mathcal{A}$ delivers this message. We use a different term for the delivery operation to stress the fact that sending by the adversary models actual delivery of the message to the recipient. Note that there need not be any correspondence between the messages

[6]Jumping ahead, we note that restricting the environment to invoke only a single instance of π considerably simplifies the analysis of protocols, since it restricts attention to a single, "stand-alone" protocol execution. An alternative formulation would allow $\mathcal{E}$ to invoke arbitrary ITIs with multiple different SIDs. This more general (but more complex) formulation captures a meaningful extension of the model. See more details in [CDPW07].

We also note that revious version of this work did not provide adequate treatment of the translation, done by the control function, from inputs provided by the environment to the values received by protocol instance, and from the outputs provided by the protocol instance to the values received by environment. Jumping ahead, this flaw resulted in imprecision in the composition theorem and the specification of protocols and ideal functionalities. We thank Margarita Vald for pointing out this flaw.

sent by the parties and the messages delivered by the adversary.[7]

Any ITI other than $\mathcal{E}$ and $\mathcal{A}$, namely the parties and sub-parties of the main instance of π, as well as the ITIs invoked by $\mathcal{A}$, are allowed to pass inputs and outputs to any other ITI other than $\mathcal{E}$ and $\mathcal{A}$, with the exception that only the main parties of this instance of π can pass inputs to (other) main parties of this instance. In addition, they can send messages to $\mathcal{A}$. (These messages may indicate an identity of an intended recipient ITI; but the adversary is not obliged to respect these indications.)

The control function responds to these external-write operations as so: If the extended identity of the target ITI of an external write request coincides with the claimed source extended identity value id_{source} in one of the previous external write operations of $\mathcal{E}$, and the target tape it the subroutine output tape, then the control function writes the requested message on the subroutine output of $\mathcal{E}$, with the code of the writing ITI removed. That is, $\mathcal{E}$ sees the identity — but not the code — of the ITI that writes to its subroutine output tape. (If some ITI M performs an external write request to an ITI with extended identity id'_{source} that differs from id_{source} only in the code of the target ITI, then the operation is not performed and M transitions to an error state, in the same way as described in the specification of the external write instruction in Section 3.1.[8])

4.2 Protocol emulation

This section formalizes the general notion of emulating one protocol via another protocol, as sketched in Section 2.2. In preparation for the formal definition, we first recall the notions of probability ensembles and indistinguishability, and define a restricted class of environments, called *balanced environments.* We then define UC-emulation in its general form, followed by some variants, including statistical, perfect, and uniform-complexity emulation. Next we formulate several more quantitative notions of emulation, and assert a somewhat surprising quantitative property of UC emulation. The section concludes with a discussion of the transitivity properties of UC emulation.

Distribution ensembles and indistinguishability. A probability distribution ensemble $X = \{X(k,z)\}_{k\in\mathbf{N},x\in\{0,1\}^*}$ is an infinite set of probability distributions, where a distribution $X(k,z)$ is associated with each $k \in \mathbf{N}$ and $z \in \{0,1\}^*$. The ensembles considered in this work describe outputs of computations where the parameter z represents input, and the parameter k represents the security parameter. As we'll see, it will suffice to restrict attention to binary distributions, i.e. distributions over $\{0,1\}$.

Definition 4 *Two binary probability distribution ensembles X and Y are* indistinguishable *(written $X \approx Y$) if for any $c,d \in \mathbf{N}$ there exists $k_0 \in \mathbf{N}$ such that for all $k > k_0$ and all $z \in \cup_{\kappa \le k^d}\{0,1\}^\kappa$ we have:*

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

[7]In previous versions of the framework, $\mathcal{A}$ was restricted to delivering messages and passing outputs to $\mathcal{E}$. However, it will sometimes be convenient for modeling purposes to allow the adversary to pass input or outputs to other special ITIs, thus we do not disallow these operations. Still,"standard" protocols will typically ignore input or subroutine output coming directly from the adversary.

[8]Note that the above "redirection of outputs" to the environment takes place whenever $\mathcal{E}$ assumes source identity id_{source} — even when id_{source} is the extended identity of an existing ITI. This captures the concept that, while a protocol might specify an intended recipient of its output values, it has no control over where these outputs are redirected to. In particular, a "well constructed" protocol should make sure that no main party of the protocol will pass output to an ITI which is a subsidiary of this protocol.

The probability distribution ensembles considered in this work represent outputs of systems of ITMs, namely outputs of environments. More precisely, we consider ensembles of the form $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}} \overset{\text{def}}{=} \{\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(k,z)\}_{k\in\mathbf{N},z\in\{0,1\}^*}$. It is stressed that Definition 4 considers the distributions $X(k,z)$ and $Y(k,z)$ only when the length of z is polynomial in k. This essentially means that we consider only environments that set the security parameter to be some polynomial fraction of the length of their input.

Balanced environments. In order to keep the notion of protocol emulation from being unnecessarily restrictive, we need to restrict attention to environments that satisfy some basic conditions regarding the lengths of inputs given to the parties. Recall that we have already restricted ourselves to parameterized systems where the length of input to each party must be at least the security parameter, and where the security parameter is a polynomial fraction of the length of input to the environment. However, this restriction does not limit the relative lengths of the inputs that the environment provides to the adversary and the parties; the difference can be any arbitrary polynomial in the security parameter, and the ratio can be arbitrary. Consequently, the model still allows the environment to create situations where the input length to the protocol, hence the protocol's complexity and communication complexity, are arbitrarily large relative to the input length and complexity of the adversary. Such situations seem unnatural; for instance, with such an environment no polytime adversary can deliver even a fraction of the protocol's communication. Indeed, it turns out that if we allow such situations then the definition of protocol emulation below (Definition 5) becomes overly restrictive.[9]

To avoid such situations, we wish to consider only environments where the amount of resources given to the adversary (namely, the length of the adversary's input) is at least some fixed polynomial fraction of the amount of resources given to the protocol. To be concrete, we consider only environments where, at any point in time during the execution, the overall length of the inputs given by $\mathcal{E}$ to the parties of the main instance of π is at most k times the the length of input to the adversary. We call such environments balanced. It is stressed that the input of the adversary can still be arbitrarily (but polynomially) long relative to the input of the parties.

We finally turn to the definition of protocol emulation:

Definition 5 (Protocol Emulation) *Let π and ϕ be PPT protocols. We say that π UC-emulates ϕ if for any PPT adversary $\mathcal{A}$ there exists a PPT adversary $\mathcal{S}$ such that for any balanced PPT environment $\mathcal{E}$ we have:*

$$\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}} \approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}. \qquad (1)$$

On statistical and perfect emulation. Definition 5 can be extended to the standard notions of statistical and perfect emulation (as in, say, [C00]). That is, when $\mathcal{A}$ and $\mathcal{E}$ are allowed unbounded complexity, and the simulator $\mathcal{S}$ is allowed to be polynomial in the complexity of $\mathcal{A}$, we say that π statistically UC-emulates ϕ. If in addition the two sides of (1) are required to be identical then we say that π perfectly UC-emulates ϕ. Another variant allows $\mathcal{S}$ to have unlimited computational power,

[9] The definition will require that any adversary be "simulatable" given a comparable amount of resources, in the sense that no environment can tell the simulated adversary from the real one. When this requirement is applied to environments that give the adversary much less resources than to the protocol, the simulation may not even be able to run the code of the protocol in question.

regardless of the complexity of $\mathcal{A}$; however, this variant provides a weaker security guarantee, see discussion in [c00].

On security with respect to closed environments. Definition 5 considers "open environments", namely environments that take input (of some polynomial length) that was generated in an arbitrary way, perhaps not even recursively. Alternatively, one may choose to consider only "closed environments", namely environment that do not receive meaningful external input. Here the inputs given to the protocol are the result of some uniform, polynomial time process. This notion of security can be captured by considering only environments whose external input contains no information other than its length, e.g. it is 1^n for some n. Such environments would choose the inputs of the parties using some internal stochastic process.

We note that the former notion corresponds naturally to considering environments and adversaries that are non-uniform polynomial time, whereas the latter notion corresponds to considering uniform polynomial time environments and adversaries. Here it is guaranteed that the simulator gets exactly the same advice as the adversary.

More quantitative notions of emulation. The notion of protocol emulation as defined above only provides a "qualitative" measure of security. That is, it essentially only gives the guarantee that "any feasible attack against π can be turned into a feasible attack against ϕ," where "feasible" is interpreted broadly as "polynomial time". We formulate more quantitative variants of this definition. We note that, besides being informative in of itself, the material here will prove instrumental in later sections.

We quantify two parameters: the emulation slack, meaning the probability by which the environment distinguishes between the interaction with π from the interaction with ϕ, and the simulation overhead, meaning the difference between the complexity of the given adversary $\mathcal{A}$ and that of the constructed adversary $\mathcal{S}$. Recall that an ITM is T-bounded if the function bounding its running time is $T(\cdot)$ (see Definition 2), and that a functional is a function from functions to functions. Then:

Definition 6 *Let π and ϕ be protocols and let ϵ, g be functionals. We say that π UC-emulates ϕ with* emulation slack ϵ *and* simulation overhead g *(or, in short, π (e,g)-*UC-emulates *ϕ), if for any polynomial $p_{\mathcal{A}}(\cdot)$ and any $p_{\mathcal{A}}$-bounded adversary $\mathcal{A}$, there exists a $g(p_{\mathcal{A}})$-bounded adversary $\mathcal{S}$, such that for any polynomial $p_{\mathcal{E}}$, any $p_{\mathcal{E}}$-bounded environment $\mathcal{E}$, any large enough value $k \in \mathbf{N}$ and any input $x \in \{0,1\}^{p_{\mathcal{E}}(k)}$ we have:*

$$|\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}}(k,x) - \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(k,x)| < \epsilon(p_A, p_{\mathcal{E}})(k).$$

Including the security parameter k is necessary when the protocol depends on it. Naturally, when k is understood from the context it can be omitted. A more concrete variant of Definition 6 abandons the asymptotic framework and instead concentrates on a specific value of the security parameter k:

Definition 7 *Let π and ϕ be protocols, let $k \in \mathbf{N}$, and let $g, \epsilon : \mathbf{N} \to \mathbf{N}$. We say that π (k,e,g)-UC-emulates ϕ if for any $t_{\mathcal{A}} \in \mathbf{N}$ and any adversary $\mathcal{A}$ that runs in time $t_{\mathcal{A}}$ there exists an adversary $\mathcal{S}$ that runs in time $g(t_{\mathcal{A}})$ such that for any $t_{\mathcal{E}} \in \mathbf{N}$, any environment $\mathcal{E}$ that runs in time $t_{\mathcal{E}}$, and any input $x \in \{0,1\}^{t_{\mathcal{E}}}$ we have:*

$$|\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}}(k,x) - \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(k,x)| < \epsilon(k, t_A, t_{\mathcal{E}}).$$

It is stressed that Definition 7 still quantifies over all environments and adversaries of all complexities. One can potentially formulate a definition that parameterizes also the run-times of the environment, adversary and simulator. That is, the the quantifies only over environments and adversaries that have specific complexity. It should be noted, however, that such a definition would be considerably weaker than definition 7, since it guarantees security only for adversaries and environments that are bounded by specific run-times. Furthermore, both the protocols and the simulator can depend on these run-times. In contrast, Definition 7 bounds the specified parameters for any arbitrarily complex environment and adversary. Consequently, while the UC theorem can be easily adapted to the formulations of Definitions 6 and 7, a definition that parameterizes complexity of the environment and adversary does not seem to guarantee universal composability.

The simulation overhead is always additive. An interesting property of the notion of UC-emulation is that the simulation overhead can be always bounded by an additive polynomial factor that depends only on the protocols in question, and is independent of the adversary and environment. That is:

Claim 8 *Let π and ϕ be protocols such that π UC-emulates ϕ with emulation slack ϵ and simulation overhead g, as in Definition 6. Then there exists a polynomial α such that π UC-emulates ϕ with emulation slack ϵ and simulation overhead $g'(p_{\mathcal{A}})(\cdot) = p_{\mathcal{A}}(\cdot) + \alpha(\cdot)$.*

Said otherwise, if π (ϵ, g)-UC-emulates ϕ then it is guaranteed that the overhead of running $\mathcal{S}$ rather than $\mathcal{A}$ can be made to be at most an additive polynomial factor $\alpha(\cdot)$ that depends only on π and ϕ. Furthermore, this can be done with no increase in the emulation slack. We call $\alpha(\cdot)$ the intrinsic simulation overhead of π with respect to ϕ.

The proof of Claim 8, while not immediate from the definition of the simulation overhead, follows as an easy corollary from the proof of Claim 10 below. We thus postpone the proof of Claim 8 till after the proof of Claim 10.

On the transitivity of emulation. It is easy to see that if protocol π_1 UC-emulates protocol π_2, and π_2 UC-emulates π_3, then π_1 UC-emulates π_3. Moreover, if π_1 (e_1, g_1)-UC-emulates π_2, and π_2 (e_2, g_2)-UC-emulates π_3, then π_1 $(e_1+e_2, g_2 \circ g_1)$-UC-emulates π_3. (Here e_1+e_2 is the functional that output the sum of the outputs of e_1 and e_2, and $\circ$ denotes composition of functionals.) Transitivity for any number of protocols $\pi_1, ..., \pi_n$ follows in the same way. Note that if the number of protocols is not bounded by a constant then the complexity of the adversary may no longer be bounded by a polynomial. Still, when there is an overall polynomial bound on the intrinsic simulation overheads of each π_i w.r.t. π_{i+1}, Claim 8 implies that the simulation overhead remains polynomial as long as the number of protocols is polynomial. Similarly the emulation slack remains negligible as long as the number of protocols is polynomial. Finally, we stress that the question of transitivity of emulation should not be confused with the question of multiple *nesting* of protocols, which is discussed in Section 5.3.

4.3 Realizing ideal functionalities

We now turn to applying the general machinery of protocol emulation towards one of the main goals of this work, namely defining security of protocols via realizing ideal functionalities.

Ideal functionalities. As discussed in Section 2, an ideal functionality represents the expected functionality of a certain task. This includes both "correctness", namely the expected input-output relations of uncorrupted parties, and "secrecy", or the acceptable leakage of information to the adversary. Technically, an ideal functionality $\mathcal{F}$ is simply an ITM as in Definition 1. In typical use, the input tape of an ideal functionality is expected to be written to by a number of ITIs; it also expects to write to the subroutine output tapes of multiple ITIs. In other words, an ideal functionality behaves like a subroutine machine for a number of different ITIs (which are thought of as parties of some protocol instance). Next, the PID of an ideal functionality is typically meaningless, and set to $\perp$. Often ideal functionalities will have some additional structure, such as specifying the response to party corruption requests by the adversary, or verifying the identity and code of the ITIs that pass input to or receive subroutine output from the ideal functionality. However, to avoid cluttering the basic definition with unnecessary details, further restrictions and conventions regarding ideal functionalities are postponed to Section 6.

Ideal protocols. In order to facilitate the use of ideal functionalities as subroutines within other protocols, we provide a "wrapper" that makes an instance of an ideal functionality look, syntactically, like an instance of multi-party protocol. This mechanism takes the form of "dummy parties". These are simple ITIs that act as placeholders for "local subroutines" of the individual ITIs of the calling protocol instance, while relaying all inputs and outputs between the calling ITIs and the ideal functionality. In accordance, the dummy parties of an instance of an ideal protocol will have the same session ID as the ideal functionality, and party IDs that are specified by their calling ITIs.

More formally, the ideal protocol $\text{IDEAL}_{\mathcal{F}}$ for ideal an functionality $\mathcal{F}$ is defined as follows:

1. When activated with input (v, eid_c, id), where v is the actual input value, eid_c is the extended identity of the calling ITI, and $id = (s, p)$ is the local identity, pass input (v, eid_c) to an instance of $\mathcal{F}$ with identity $(s, \perp)$. If the input operation was not successful then pass a failure output to eid_c.

2. When activated with subroutine output (v, id, eid_t) from an ITI with code $\mathcal{F}$ and identity $(s, \perp)$, where id is the local identity and v is the actual output value, pass output v to the ITI with extended identity eid_t. If the output operation was not successful then pass a failure input to $\mathcal{F}$, namely to the ITI $(\mathcal{F}, s, \perp)$.

3. Messages delivered by $\mathcal{A}$, including corruption messages, are ignored. (The intention here is that, in the ideal protocol, the adversary should send corruption messages directly to the ideal functionality. See more discussion in Section 6.2.)

Notice that the dummy parties pass the code of the calling ITIs to the ideal functionality, and allow the ideal functionality to verify the code of recipient ITIs. This is essential for modeling tasks such as entity and message authentication. The present structure also allows an ideal functionality to create a recipient ITI. It does not require that such an ITI (with appropriate SID and PID) be created ahead of time via an external mechanism. Also, if $\mathcal{F}$ is PPT then so is $I_{\mathcal{F}}$. Finally note that a dummy party makes no use of its PID. Indeed, the PID is there only as a 'placeholder' to be used by protocols that realize the ideal protocol. We use the term $\mathcal{F}$-hybrid protocols to denote protocols that make subroutine calls to $I_{\mathcal{F}}$.[10]

[10]The above description of $I_{\mathcal{F}}$ is not necessarily PPT as in Definition 2. This is so since a dummy party of $I_{\mathcal{F}}$ may

Realizing an ideal functionality. Protocols that realize an ideal functionality are defined as protocols that emulate the ideal protocol for this ideal functionality:

Definition 9 *Let $\mathcal{F}$ be an ideal functionality and let π be a protocol. We say that π* UC-realizes *$\mathcal{F}$ if π UC-emulates the ideal protocol for $\mathcal{F}$.*

4.4 Alternative formulations of UC emulation

We discuss some alternative formulations of the definition of protocol emulation (Definition 5) and show that they are all equivalent to that formulation.

On environments with non-binary outputs. Definition 5 quantifies only over environments that generate binary outputs. One may consider an extension to the models where the environment has arbitrary output; here the definition of security would require that the two output ensembles $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$ and $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}}$ (that would no longer be binary) be *computationally indistinguishable,* as defined by Yao [Y82] (see also [G01]). It is easy to see, however, that this extra generality results in a definition that is equivalent to Definition 5. We leave the proof as an exercise.

On deterministic environments. Since we consider environments that receive an arbitrary external input of polynomial length, it suffices to consider only deterministic environments. That is, the definition that quantifies only over deterministic environments is equivalent to Definition 5. Again, we leave the proof as an exercise. Note however that this equivalence does *not* hold for the case of closed environments, where the environment only receives inputs of the form 1^n.

4.4.1 Emulation with respect to the dummy adversary

We show that Definition 5 can be simplified as follows. Instead of quantifying over all possible adversaries $\mathcal{A}$, it suffices to require that the ideal-protocol adversary $\mathcal{S}$ be able to simulate, for any environment $\mathcal{E}$, the behavior of a specific and very simple adversary. This adversary, called the "dummy adversary", only delivers to parties messages generated by the environment, and delivers to the environment all messages generated by the parties. Said otherwise, we essentially show that the dummy adversary is "the hardest adversary to simulate", in the sense that simulating this adversary implies simulating all adversaries. Intuitively, the reason that the dummy adversary is the "hardest to simulate" is that it gives the environment full control over the communication. It thus leaves the simulator with very little "wiggle room."

receive little or no input from $\mathcal{E}$, while receiving long incoming communication from some other party. One way to resolve this issue is to demonstrate that Proposition 3 holds as long as $\mathcal{F}$ is PPT, even when $I_{\mathcal{F}}$ is not. We take an alternative route, and modify $I_{\mathcal{F}}$ to make it PT. We do this in two steps. First, we bound the runtime of $I_{\mathcal{F}}$ by some arbitrary polynomial, say it is allowed some constant times the number of operations as the length of its input. By itself, however, this stipulation is over-restrictive since it means that a dummy party cannot generate outputs prior to receiving input of comparable length. We thus let a dummy party distinguish between two types of input: One type of input is the one described above, to be forwarded to $\mathcal{F}$. The other type of input is of the form 1^m for some m. This input is ignored by the dummy party; it is used only to allow the calling ITI to provide the dummy party with extra runtime.

We note that previous formulations of the ideal protocol (and also of the dummy adversary in Claim 10) ignored this aspect, thus resulting in an incorrect proof of Claims 10 and 11. We thank Ralf Küsters for pointing out these errors.

More specifically, the dummy adversary, denoted $\mathcal{D}$, proceeds as follows. When activated with an incoming message m on its incoming communication tape, adversary $\mathcal{D}$ passes m as output to $\mathcal{E}$, along with the extended identity of the sender. When activated with an input (m, id, c) from $\mathcal{E}$, where m is a message, id is an identity, and c is a code for a party, $\mathcal{D}$ delivers the message m to the party with identity id and code c. (Jumping ahead, this in particular means that $\mathcal{D}$ corrupts parties when instructed by $\mathcal{E}$, and passes all gathered information to $\mathcal{E}$.) Finally, $\mathcal{D}$ ignores any prefix of its input of the form 1^*. This allows $\mathcal{E}$ to provide $\mathcal{D}$ with additional runtime without specifing any value to be delivered.

Say that protocol π UC-emulates protocol ϕ with respect to the dummy adversary if there exists an adversary $\mathcal{S}$ such that for any environment $\mathcal{E}$ we have $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}} \approx \text{EXEC}_{\pi,\mathcal{D},\mathcal{E}}$. We show:

Claim 10 *Let π, ϕ be protocols. Then π UC-emulates ϕ according to Definition 5 if and only if it UC-emulates ϕ with respect to the dummy adversary.*

Proof: Clearly if π UC-emulates ϕ according to Definition 5 then it UC-emulates ϕ with respect to dummy adversaries. The idea of the derivation in the other direction is that, given direct access to the communication sent and received by the parties, the environment can run any adversary by itself. Thus quantifying over all environments essentially implies quantification also over all adversaries. More precisely, let π, ϕ be protocols and let $\mathcal{S}_{\mathcal{D}}$ be the adversary guaranteed by the definition of emulation with respect to dummy adversaries (that is, $\mathcal{S}_{\mathcal{D}}$ satisfies $\text{EXEC}_{\phi,\mathcal{S}_{\mathcal{D}},\mathcal{E}} \approx \text{EXEC}_{\pi,\mathcal{D}_{c_\pi},\mathcal{E}}$ for all $\mathcal{E}$.) We show that π UC-emulates ϕ according to Definition 5. For this purpose, given an adversary $\mathcal{A}$ we construct the adversary $\mathcal{S}$ as follows. $\mathcal{S}$ runs simulated instances of $\mathcal{A}$ and $\mathcal{S}_{\mathcal{D}}$. In addition:

1. $\mathcal{S}$ forwards any input from the environment to the simulated $\mathcal{A}$, and passes any output of $\mathcal{A}$ to the environment.
2. When the simulated $\mathcal{A}$ delivers a message m to an ITI with identity id and code c, $\mathcal{S}$ activates $\mathcal{S}_{\mathcal{D}}$ with input (m, id, c). Similarly, any output generated by $\mathcal{S}_{\mathcal{D}}$ is copied to the incoming communication tape of $\mathcal{A}$.
3. Whenever $\mathcal{S}_{\mathcal{D}}$ writes a message m on some tape of some ITI, $\mathcal{S}$ writes m to that tape of that ITI. Finally, when $\mathcal{S}$ obtains a message m on its incoming communication tape, it proceeds as follows: It first writes 1^m on the input tape of $\mathcal{S}_{\mathcal{D}}$; next (i.e., in the next activation) it writes m on the incoming communication tape of $\mathcal{S}_{\mathcal{D}}$.

A graphical depiction of the operation of $\mathcal{S}$ appears in Figure 6.

Analysis of $\mathcal{S}$. We first argue that $\mathcal{S}$ is PPT. The running time of $\mathcal{S}$ is dominated by the runtime of the $\mathcal{A}$ module plus the runtime of the $\mathcal{S}_{\mathcal{D}}$ module. When $\mathcal{S}$ has input of length n, the runtime of the $\mathcal{A}$ module is bounded by $p_{\mathcal{A}}(n)$, where $p_{\mathcal{A}}$ is the polynomial bounding the running time of $\mathcal{A}$. To bound the runtime of $\mathcal{S}_{\mathcal{D}}$, recall that $\mathcal{S}_{\mathcal{D}}$ gets inputs both from $\mathcal{A}$ and from $\mathcal{S}$ itself. The overall length of the first input is bounded by $p_{\mathcal{A}}(n)$. The length of the second input is bounded by the overall communication generated by ϕ. This quantity may in principle be unrelated to n, the length of $\mathcal{S}$'s input. However, since the environment is balanced, we have that the communication generated by π is at most $n'p_\pi(kn)$, where p_π is the polynomial bounding the complexity of π, where n' is the number of ITIs in the main instance of π. In addition, without loss of generality

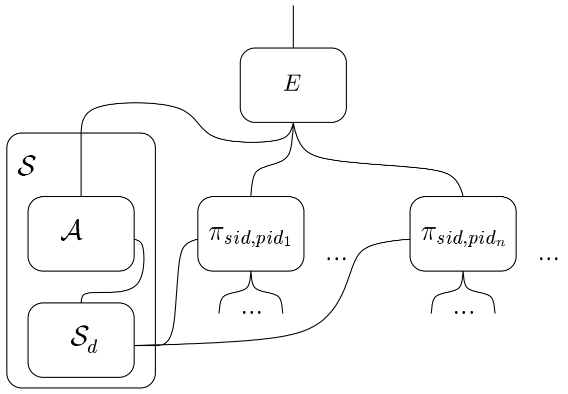

Figure 6: The operation of simulator $\mathcal{S}$ in the proof of Claim 10: Both $\mathcal{A}$ and $\mathcal{S}_{\mathcal{D}}$ are simulated internally by $\mathcal{S}$. The same structure represents also the operation of the shell adversary in the definition of black-box simulation (see Section 4.4.2).

the length of the input coming from $\mathcal{A}$ can also be bounded by $n'p_\pi(kn)$. We conclude that the polynomial bounding the runtime of $\mathcal{S}$ is at most

$$p_{\mathcal{S}}(a) = p_{\mathcal{A}}(a) + p_{\mathcal{S}_{\mathcal{D}}}(n'p_\pi(ka))$$

Note that the right hand side summand is a polynomial that depends only on π and ϕ; this fact is used in the proof of Claim 8 below.

Next we assert the validity of $\mathcal{S}$. Assume for contradiction that there is an adversary $\mathcal{A}$ and a balanced environment $\mathcal{E}$ such that $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}} \not\approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$. We construct a balanced environment $\mathcal{E}_{\mathcal{D}}$ such that $\text{EXEC}_{\pi,\mathcal{S}_{\mathcal{D}},\mathcal{E}_{\mathcal{D}}} \not\approx \text{EXEC}_{\pi,\mathcal{D},\mathcal{E}_{\mathcal{D}}}$. Environment $\mathcal{E}_{\mathcal{D}}$ runs an interaction between simulated instances of $\mathcal{E}$ and $\mathcal{A}$. In addition:

1. All the inputs generated by $\mathcal{E}$ to the adversary are forwarded to $\mathcal{A}$, and all of $\mathcal{A}$'s outputs are forwarded to $\mathcal{E}$.

2. Whenever $\mathcal{E}_{\mathcal{D}}$ receives an output value v from its adversary, $\mathcal{E}_{\mathcal{D}}$ passes v to the simulated $\mathcal{A}$. Similarly, whenever the simulated $\mathcal{A}$ delivers a message m to some ITI, $\mathcal{E}_{\mathcal{D}}$ instructs the external adversary to deliver message m to that ITI.

3. All inputs from $\mathcal{E}$ to the parties of π are forwarded to the external parties, and all the outputs coming from the external parties are forwarded to $\mathcal{E}$ as coming from the parties of π.

4. In addition, whenever $\mathcal{E}_{\mathcal{D}}$ passes input of length m to some party, it first passes input $1^{p(m)}$ to the external adversary, where $p()$ is the maximum between the polynomials bounding the run times of ϕ and π. This makes sure that $\mathcal{E}_{\mathcal{D}}$ is balanced.

5. Finally, $\mathcal{E}_{\mathcal{D}}$ outputs whatever the simulated $\mathcal{E}$ outputs.

It can be readily verified that the ensembles $\text{EXEC}_{\pi,\mathcal{D},\mathcal{E}_{\mathcal{D}}}$ and $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$ are identical. In particular, $\mathcal{E}_{\mathcal{D}}$ makes sure that $\mathcal{D}$ never halts due to insufficient runtime. Similarly, ensembles $\text{EXEC}_{\phi,\mathcal{S}_{\mathcal{D}},\mathcal{E}_{\mathcal{D}}}$ and $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}}$ are identical. □

Discussion. From a technical point of view, emulation with respect to the dummy adversary is an easier definition to work with, since it involves one less quantifier, and furthermore it restricts the interface of the environment with the adversary to be very simple. Indeed, we almost always prefer to work with this notion. However, we chose not to present this formulation as the main notion of protocol emulation, since we feel is is less intuitively appealing than Definition 9. In other words, we find it harder to get convinced that this definition captures the security requirements of a given task. In particular, it looks farther away from the the basic notion of security in, say, [C00]. Also, it is less obvious that this definition has some basic closure properties such as transitivity.[11]

Proof of Claim 8. Claim 8 states that if a protocol π UC-emulates protocol ϕ then there exists a polynomial $\alpha(\cdot)$ such that, for any adversary $\mathcal{A}$ whose running time is bounded by the polynomial $p_{\mathcal{A}}(\cdot)$, there is a simulator whose running time is bounded by $p_{\mathcal{A}}(\cdot) + \alpha(\cdot)$. The claim follows from the proof of Claim 10. Indeed, the proof of Claim 10 shows how to construct, for any adversary $\mathcal{A}$, a valid simulator $\mathcal{S}_{\mathcal{D}}$, whose complexity is bounded by $p_{\mathcal{A}}(n) + \alpha(n)$, where $p_{\mathcal{A}}$ is the polynomial bounding the running time of $\mathcal{A}$ and $\alpha(\cdot)$ is a polynomial that depends only on π and ϕ. $\square$

4.4.2 Emulation with respect to black box simulation

Another alternative formulation of Definition 5 imposes the following technical restriction on the simulator $\mathcal{S}$: Instead of allowing a different simulator for any adversary $\mathcal{A}$, let the simulator have "black-box access" to $\mathcal{A}$, and require that the code of the simulator remains the same for all $\mathcal{A}$. Restricting the simulator in this manner does not seem to capture any tangible security concern. Still, in other contexts, e.g. in the classic notion of Zero-Knowledge, this requirement results in a strictly more restrictive notion of security than the definition that lets $\mathcal{S}$ depend on the description of $\mathcal{A}$, see e.g. [GK88, B01]. We show that in the UC framework security via black-box simulation is *equivalent* to the standard notion of security.

We formulate black box emulation in a way that keeps the overall model of protocol execution unchanged, and instead imposes restrictions on the operation of the simulator. Specifically, an adversary $\mathcal{S}$ is called a *shell simulator* if it operates as follows, given an ITM $\hat{\mathcal{S}}$ (called a black-box simulator) and a PPT adversary $\mathcal{A}$. $\mathcal{S}$ first internally invokes an instance of $\mathcal{A}$ and an instance of $\hat{\mathcal{S}}$. Next:

- Upon receiving an input from the environment, $\mathcal{S}$ forwards this input to $\mathcal{A}$. Any outgoing message generated by $\mathcal{A}$ is given as input to $\hat{\mathcal{S}}$. Instructions of $\hat{\mathcal{S}}$ regarding delivering messages to parties are carried out.
- Upon receiving an incoming message from some party, $\mathcal{S}$ forwards this incoming message to $\hat{\mathcal{S}}$. Outputs of $\hat{\mathcal{S}}$ are forwarded as incoming messages to $\mathcal{A}$, and outputs of $\mathcal{A}$ are outputted to $\mathcal{E}$.

Observe that the structure of $\mathcal{S}$ is the same as the structure of the simulator $\mathcal{S}$ in the proof of Claim 10, where $\hat{\mathcal{S}}$ plays the role of $\mathcal{S}_{\mathcal{D}}$. Indeed, Figure 6 depicts the operation of a black-box simulator (substitute $\hat{\mathcal{S}}$ for $\mathcal{S}_{\mathcal{D}}$).

[11]One might be tempted to further simplify the notion of emulation with respect to the dummy adversary by removing the dummy adversary altogether and letting the environment interact directly with the ITIs running the protocol. We note however that this definition would be over-restrictive, unless the environment is required to be balanced. See discussion in Footnote 9.

Let $\text{EXEC}_{\phi,\mathcal{S}^{\hat{\mathcal{S}},\mathcal{A}},\mathcal{E}}$ denote the output of $\mathcal{E}$ from an interaction with protocol ϕ and a shell adversary $\mathcal{S}$ that runs a black-box simulator $\hat{\mathcal{S}}$ and an adversary $\mathcal{A}$. Say that a protocol π UC-emulates protocol ϕ with black-box simulation if there exists a black-box simulator $\hat{\mathcal{S}}$ such that for any PPT adversary $\mathcal{A}$, the resulting shell adversary $\mathcal{S}$ is PPT, and furthermore, and any PPT environment $\mathcal{E}$, we have $\text{EXEC}_{\phi,\mathcal{S}^{\hat{\mathcal{S}},\mathcal{A}},\mathcal{E}} \approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$. (It is stressed that $\hat{\mathcal{S}}$ need not necessarily be PPT as an ITM; only $\mathcal{S}$ needs to be PPT.) We show:

Claim 11 *Let π, ϕ be PPT multiparty protocols. Then π UC-emulates ϕ according to Definition 5 if and only if it UC-emulates ϕ with black-box simulation.*

Proof: The 'only if' direction follows from the definition. For the 'if' direction, notice that the simulator $\mathcal{S}$ in the proof of Claim 10 can be cast as a shell adversary with adversary $\mathcal{A}$ and the following black-box simulator $\hat{\mathcal{S}}$: $\hat{\mathcal{S}}$ runs $\mathcal{S}_{\mathcal{D}}$; in addition, before delivering a message of length m to the input tape of $\mathcal{S}_{\mathcal{D}}$, $\hat{\mathcal{S}}$ writes 1^m on the input tape of $\mathcal{S}_{\mathcal{D}}$. (Indeed, $\hat{\mathcal{S}}$ is not PPT as an ITM by itself. However, as argued in the proof of Claim 10, for any PPT adversary $\mathcal{A}$, the shell simulator $\mathcal{S}$ is PPT with runtime bounded by $p_{\mathcal{A}}(\cdot) + p_{\mathcal{S}_{\mathcal{D}}}(p_{\phi}(\cdot))$.) □

Discussion. The present formulation of security via black-box simulation is considerably more restrictive than that of standard cryptographic modeling of black-box simulation. In particular, in the standard modeling S may query $\mathcal{A}$ in arbitrary ways. In contrast, here the communication between $\hat{\mathcal{S}}$ and $\mathcal{A}$ is restricted, in that $\hat{\mathcal{S}}$ cannot "reset" or "rewind" $\mathcal{A}$. Still, the present definition is equivalent to the general (non black-box) notion of security.

We remark that the present formulation of black-box simulation is reminiscent of the notions of strong black-box simulation in [DKMR05] and in [PW00] (except for the introduction of the shell adversary). However, in these works this notion is not equivalent to the standard one, due to different formalizations of probabilistic polynomial time.

4.4.3 Letting the simulator depend on the environment

Consider a seemingly weaker variant of Definition 5, where the simulator $\mathcal{S}$ can depend on the code of the environment $\mathcal{E}$. That is, for any $\mathcal{A}$ and $\mathcal{E}$ there should exist a simulator $\mathcal{S}$ that satisfies $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}} \approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$. Following [L03], we call this variant security with respect to specialized simulators. We demonstrate that this variant is equivalent to the main definition (Definition 5).

Claim 12 *A protocol π UC-emulates protocol ϕ according to Definition 5 if and only if it UC-emulates ϕ with respect to specialized simulators.*

Proof: Clearly, if π UC-emulates ϕ as in Definition 5 then UC-emulates ϕ with respect to specialized simulators. To show the other direction, assume that π UC emulates ϕ with respect to specialized simulators. That is, for any PPT adversary $\mathcal{A}$ and PPT environment $\mathcal{E}$ there exists a PPT simulator $\mathcal{S}$ such that (1) holds. Consider the "universal environment" $\mathcal{E}_u$ which expects its input to consist of $(\langle\mathcal{E}\rangle, z, 1^t)$, where $\langle\mathcal{E}\rangle$ is an encoding of an ITM $\mathcal{E}$, z is an input to $\mathcal{E}$, and t is a bound on the running time of $\mathcal{E}$. Then, $\mathcal{E}_u$ runs $\mathcal{E}$ on input z for up to t steps, outputs whatever $\mathcal{E}$ outputs, and halts. Clearly, machine $\mathcal{E}_u$ is PPT. (in fact, it runs in linear time in its input length). We are thus guaranteed that there exists a simulator $\mathcal{S}$ for $\mathcal{E}_u$ such that (1) holds. We claim that $\mathcal{S}$ satisfies

(1) with respect to *any* balanced PPT environment $\mathcal{E}$. To see this, fix a PPT machine $\mathcal{E}$ as in Definition 2, and let c be the constant exponent that bounds $\mathcal{E}$'s running time. For each $k \in \mathbf{N}$ and $z \in \{0,1\}^*$, the distribution $\textsc{exec}_{\phi,\mathcal{S},\mathcal{E}}(k,z)$ is identical to the distribution $\textsc{exec}_{\phi,\mathcal{S},\mathcal{E}_u}(k,z_u)$, where $z_u = (\langle\mathcal{E}\rangle, z, 1^{c\cdot|z|})$. Similarly, the distribution $\textsc{exec}_{\pi,\mathcal{A},\mathcal{E}}(k,z)$ is identical to the distribution $\textsc{exec}_{\pi,\mathcal{A},\mathcal{E}_u}(k,z_u)$. Consequently, for any $d \in \mathbf{N}$ we have:

$$\begin{aligned}\{\textsc{exec}_{\phi,\mathcal{S},\mathcal{E}}(k,z)\}_{k\in\mathbf{N},z\in\{0,1\}^{\le k^d}} &= \{\textsc{exec}_{\phi,\mathcal{S},\mathcal{E}_u}(k,z_u)\}_{k\in\mathbf{N},z_u=(\langle\mathcal{E}\rangle,z\in\{0,1\}^{\le k^d},1^{c\cdot|z|})}\\ &\approx \{\textsc{exec}_{\pi,\mathcal{A},\mathcal{E}_u}(k,z_u)\}_{k\in\mathbf{N},z_u=(\langle\mathcal{E}\rangle,z\in\{0,1\}^{\le k^d},1^{c\cdot|z|})}\\ &= \{\textsc{exec}_{\pi,\mathcal{A},\mathcal{E}}(k,z)\}_{k\in\mathbf{N},z\in\{0,1\}^{\le k^d}}.\end{aligned}$$

In particular, as long as $|z|$ is polynomial in k, we have that $|z_u|$ is also polynomial in k (albeit with a different polynomial). Consequently, $\textsc{exec}_{\phi,\mathcal{S},\mathcal{E}} \approx \textsc{exec}_{\pi,\mathcal{A},\mathcal{E}}$. (Notice that if $|z_u|$ were not polynomial in k then the last derivation would not hold.) □

Remark: Claim 12 is an extension of the equivalence argument for the case of computationally unbounded environment and adversaries, discussed in [c00], in the context of computationally unbounded adversaries. A crucial element in the proof of this claim is the fact that the class of allowed environment permits existence of an environment $\mathcal{E}_u$ that is universal with respect to all allowed environments. In the context of computationally bounded environments, this feature becomes possible when using a definition of PPT ITMs where the running time may depend not only on the security parameter, but also on the length of the input. Indeed, in [c00] and in previous versions of this work, which restrict ITMs to run in time that is bound by a fixed polynomial in the security parameter, standard security and security with respect to specialized simulators end up being different notions (see, e.g., [l03, hu05]). Similarly, the proof of Claim 12 does *not* hold for the notion of security with respect to closed environments, i.e. environments that take inputs only of the form 1^n for some n.

Finally we note that the the current proof of the UC composition theorem does *not* hold for UC emulation with respect to specialized simulators.

5 Universal composition

This section states and proves the universal composition theorem. Section 5.1 defines the composition operation and states the composition theorem. Section 5.2 presents the proof. Section 5.3 discusses and motivates some aspects of the theorem, and sketches some extensions. Additional discussion on the implications of the UC operation and theorem appears in [c06, c13].

5.1 The universal composition operation and theorem

While the main intended use of universal composition is for replacing an ideal functionality $\mathcal{F}$ with a protocol that securely realizes $\mathcal{F}$, we define universal composition more generally, in terms of replacing one subroutine protocol with another. This both simplifies the presentation and makes the result more powerful.

Universal composition. We present the composition operation in terms of an operator on protocols. This operator, called the universal composition operator $\textsc{uc}()$, is defined as follows. Given a protocol ϕ, a protocol ρ (that presumably makes subroutine calls to ϕ), and a protocol π (that presumably UC-emulates ϕ), the composed protocol $\rho^{\phi\to\pi} = \textsc{uc}(\rho, \pi, \phi)$ is identical to protocol ρ, with the following modifications.

1. Wherever ρ contains an instruction to pass input x to an ITI running ϕ with identity (sid, pid), then $\rho^{\phi\to\pi}$ contains instead an instruction to pass input x to an ITI running π with identity (sid, pid).

2. Whenever $\rho^{\phi\to\pi}$ receives an output passed from $\pi_{(sid,pid')}$ (i.e., from an ITI running π with identity (sid, pid'), it proceeds as ρ proceeds when it receives an output passed from $\phi_{(sid,pid')}$.

In other words, the program of $\rho^{\phi\to\pi}$ can be thought of as consisting of an internal "core" part that's identical to ρ, and a separate "shell" part that performs the translation between calling ϕ and calling π. When protocol ϕ is the ideal protocol $\textsc{ideal}_{\mathcal{F}}$ for some ideal functionality $\mathcal{F}$, we denote the composed protocol by $\rho^{\pi/\mathcal{F}}$. Also, when ϕ is understood from the context we use the shorthand ρ^{π} instead for $\rho^{\phi\to\pi}$. See a graphical depiction in Figure 4 on page 14.

We remark that the composition operation can alternatively be defined as a model operation where the protocols remain unchanged, and the only change is that the control function invokes instances of ρ instead of instances ϕ. We find the present formulation, where the protocol determines the code run by its subroutines, intuitively appealing. It is also more expressive, allowing to capture protocols where the identities and code of the subroutines are not statically pre-determined.

Clearly, if protocols ρ, ϕ, and π are PPT then $\rho^{\phi\to\pi}$ is PPT (with a bounding polynomial that is the maximum of the individual bounding polynomials).

Subroutine Respecting protocols. Before stating the theorem, We define the following set of properties of protocols. While natural, these properties are needed for the proof to go through. Roughly speaking, a protocol π is subroutine respecting if: (a) the only input/output interface between each instance of π and other protocol instances in the system is done by the main parties of π, and (b) the identities of all the subsidiaries of each instance of π are known to the adversary.

More precisely, say that protocol π is subroutine respecting if the following properties hold with respect to any instance of π in any execution of any protocol ρ that makes subroutine calls to π:

1. No ITI which is a subsidiary of this instance passes outputs to an ITI which is not a party or subsidiary of this instance. Furthermore, all subsuduaries of this instance of π ignore all inputs received from parties other than the parties and subsidiaries of this instance. (Technically, ignoring an incoming value means immediately erasing it and reverting to the state prior to reading.)

2. At first activation, each ITI that is currently a subsidiary of this instance, or will ever become one, sends a special message to the adversary, notifying it of its own code and identity, as well as the code π and the SID of this instance.[12]

[12]Prior versions of this work did not provide an adequate treatment of the need to expose the subroutine structure of protocols, resulting in a flaw in the proof of the UC theorem (see Footnote 14). The flaw was pointed out in [HS11].

One natural method for guaranteeing the second property (and thus avoiding this flaw) is to mandate a hierarchical tree-like subroutine structure for protocol invocation, and have the hierarchical structure be reflected in the session IDs. This method is indeed mandated in [HS11]. We note however that there are other ways to guarantee property 2. Furthermore, at times this hierarchical structure is over-restrictive. See more discussion in Section 6.3.

Theorem statement. We are now ready to state the composition theorem. First we state a general theorem, to be followed by two corollaries. The general formulation makes the following statement: Let ρ, π, ϕ be protocols, such that protocol π UC-emulates protocol ϕ as in Definition 5 and both ϕ and π are subroutine respecting. Then the protocol $\rho^{\phi\to\pi} = \textsc{uc}(\rho, \pi, \phi)$ UC-emulates protocol ρ. A more quantitative statement of the UC theorem is discussed in Section 5.3.

Theorem 13 (Universal composition: General statement) *Let ρ, π, ϕ be PPT protocols such that π UC-emulates ϕ and both ϕ and π are subroutine respecting. Then protocol $\rho^{\phi\to\pi}$ UC-emulates protocol ρ.*

As a special case, we get:

Corollary 14 *Let ρ, π be PPT protocols such that π UC-realizes a PPT ideal functionality $\mathcal{F}$, and both ϕ and π are subroutine respecting. Then protocol $\rho^{\pi/\mathcal{F}}$ UC-emulates protocol ρ.*

Next we concentrate on protocols ρ that securely realize some ideal functionality $\mathcal{G}$. The following corollary essentially states that if protocol ρ securely realizes $\mathcal{G}$ using calls to an ideal functionality $\mathcal{F}$, $\mathcal{F}$ is PPT, and π securely realizes $\mathcal{F}$, then $\rho^{\pi/\mathcal{F}}$ securely realizes $\mathcal{G}$.

Corollary 15 (Universal composition: Realizing functionalities) *Let $\mathcal{F}, \mathcal{G}$ be ideal functionalities such that $\mathcal{F}$ is PPT. Let ρ be a subroutine respecting protocol that UC-realizes $\mathcal{G}$, and let π be a subroutine respecting protocol that securely realizes $\mathcal{F}$. Then the composed protocol $\rho^{\pi/\mathcal{F}}$ securely realizes $\mathcal{G}$.*

Proof: Let $\mathcal{A}$ be an adversary that interacts with parties running $\rho^{\pi/\mathcal{F}}$. Theorem 13 guarantees that there exists an adversary $\mathcal{A}_{\mathcal{F}}$ such that $\textsc{exec}_{\pi,\mathcal{A}_{\mathcal{F}},\mathcal{E}} \approx \textsc{exec}_{\rho^{\pi/\mathcal{F}},\mathcal{A},\mathcal{E}}$ for any environment $\mathcal{E}$. Since ρ UC-realizes $\mathcal{G}$, there exists a simulator $\mathcal{S}$ such that $\textsc{ideal}_{\mathcal{G},\mathcal{S},\mathcal{E}} \approx \textsc{exec}_{\rho,\mathcal{A}_{\mathcal{F}},\mathcal{E}}$ for any $\mathcal{E}$. Using the transitivity of indistinguishability of ensembles we obtain that $\textsc{ideal}_{\mathcal{G},\mathcal{S},\mathcal{E}} \approx \textsc{exec}_{\rho^{\pi/\mathcal{F}},\mathcal{A},\mathcal{E}}$ for any environment $\mathcal{E}$. □

5.2 Proof of the composition theorem

A high-level sketch of the proof was presented in section 2. Section 5.2.1 contains an outline of the proof. A detailed proof appears in Section 5.2.2.

5.2.1 Proof outline

The proof uses the equivalent formulation of emulation with respect to dummy adversaries (see Claim 10). This formulation considerably simplifies the presentation of the proof.

Let ρ, ϕ and π be PPT protocols such that π UC-emulates ϕ, and let $\rho^{\pi} = \rho^{\phi\to\pi} = \textsc{uc}(\rho, \pi, \phi)$ be the composed protocol. We wish to construct an adversary $\mathcal{S}$ so that no $\mathcal{E}$ will be able to tell whether it is interacting with $\rho^{\phi\to\pi}$ and the dummy adversary or with ρ and $\mathcal{S}$. That is, for any $\mathcal{E}$, $\mathcal{S}$ should satisfy

$$\textsc{exec}_{\rho^{\phi\to\pi},\mathcal{D},\mathcal{E}} \approx \textsc{exec}_{\rho,\mathcal{S},\mathcal{E}}. \tag{2}$$

The general outline of the proof proceeds as follows. The fact that π emulates ϕ guarantees that there exists an adversary (called a simulator) $\mathcal{S}_\pi$, such that for any environment $\mathcal{E}_\pi$ we have:

$$\textsc{exec}_{\pi,\mathcal{D},\mathcal{E}_\pi} \approx \textsc{exec}_{\phi,\mathcal{S}_\pi,\mathcal{E}_\pi}. \tag{3}$$

Simulator $\mathcal{S}$ is constructed out of $\mathcal{S}_\pi$. We then demonstrate that $\mathcal{S}$ satisfies (2). This is done by reduction: Given an environment $\mathcal{E}$ that violates (2), we construct an environment $\mathcal{E}_\pi$ that violates (3).

Simulator $\mathcal{S}$ operates as follows. Recall that $\mathcal{E}$ expects to interact with parties running ρ^π. The idea is to separate the interaction between $\mathcal{E}$ and the parties into several parts. To mimic the sending and receiving of messages from the parties of each instance of π (and their subsidiaries), $\mathcal{S}$ runs an instance of the simulator $\mathcal{S}_\pi$. To mimic the sending and receiving of messages from the rest of the ITIs in the system (including the main parties of ρ and their subsidiaries which are not parties or subsidiaries of an instance of π), $\mathcal{S}$ interacts directly with these parties.

A bit more specifically, recall that $\mathcal{E}$ expects to receive, via the dummy adversary, the messages sent by the parties of ρ, by the parties of all instances of π, and by all their subsidiaries. In addition, $\mathcal{E}$ delivers messages to all these entities. $\mathcal{S}$ runs an instance of the simulator $\mathcal{S}_\pi$ for each instance of π in the system it interacts with. When activated with a message sent by a party which is a party or subsidiary of an instance of ϕ, $\mathcal{S}$ forwards this message to the corresponding instance of $\mathcal{S}_\pi$. If the message is coming from another ITI, $\mathcal{S}$ forwards this message to $\mathcal{E}$, just as the dummy adversary would. (In fact, only "top-level" instances of ϕ, namely only instances of ϕ that are not subsidiaries of other instances of ϕ, will have an instance of $\mathcal{S}_\pi$ associated with them. The other instances of ϕ will be "handled" by the instance of $\mathcal{S}_\pi$ associated with the corrsponding top-level instance of ϕ.)

When activated with message m sent by $\mathcal{E}$ to a party or subsidiary of an instance of π, $\mathcal{S}$ forwards m to the corresponding instance of $\mathcal{S}_\pi$. If the message is to be delivered to another ITI, then $\mathcal{S}$ delivers m to the actual intended recipient. Any output from an instance of $\mathcal{S}_\pi$ is passed as output to $\mathcal{E}$, and any outgoing message delivered by an instance of $\mathcal{S}_\pi$ is delivered to the actual party. (Since π and ϕ are subroutine respecting, $\mathcal{S}$ has enough information to decide, given a message from $\mathcal{E}$ to be delivered, or a message coming from the rest of the system, whether the target ITI specified in the message is a party or subsidiary of some instance of π.) Figure 7 presents a graphical depiction of the operation of $\mathcal{S}$.

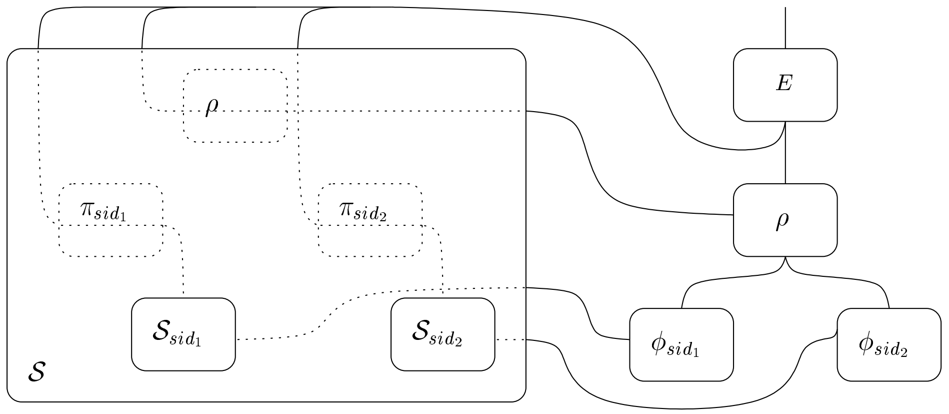

Figure 7: The operation of $\mathcal{S}$. Inputs from $\mathcal{E}$ that represent messages of the instance of ρ are forwarded to the actual instance of ρ. Inputs directed to an instance of π are directed to the corresponding instance of $\mathcal{S}$. Messages from an instance of $\mathcal{S}$ are directed to the corresponding actual instance of ϕ. For graphical clarity we use a single box to represent an instance of a multi-party protocol.

The validity of $\mathcal{S}$ is demonstrated, based on the validity of $\mathcal{S}_\pi$, via a hybrids argument. While the basic logic of the argument is standard, applying the argument to our setting requires some care. We sketch this argument. (The actual argument is slightly more complex; still, this sketch captures the essence of the argument.) Let t be an upper bound on the number of instances of π that are invoked in this interaction. Informally, for $1 \leq l \leq t$ we let ρ_l denote the protocol where the interaction with the first l instances of ϕ remains unchanged, whereas the rest of the instances of ϕ are replaced with instances of π. In particular, protocol ρ_t is essentially identical to protocol ρ. Similarly, protocol ρ_0 is essentially identical to protocol ρ^π.[13]

Now, assume that there exists an environment $\mathcal{E}$ that distinguishes with probability ϵ between an interaction with $\mathcal{S}$ and ρ, and an interaction with $\mathcal{D}$ and $\rho^{\phi\rightarrow\pi}$. Then there is an $0 < l \leq t$ such that $\mathcal{E}$ distinguishes between an interaction with $\mathcal{S}$ and ρ_l, and an interaction with $\mathcal{S}$ and ρ_{l-1}. We then construct an environment $\mathcal{E}_\pi$ that uses $\mathcal{E}$ distinguish with probability ϵ/t between an interaction with $\mathcal{D}$ and parties running a single instance of π, and an interaction with $\mathcal{S}_\pi$ and ϕ.

Essentially, $\mathcal{E}_\pi$ runs a simulated execution of $\mathcal{E}$, adversary $\mathcal{S}$, and parties running ρ_l, but with the following exception. $\mathcal{E}_\pi$ uses its actual interaction (which is either with ϕ or with ρ) to replace the parts of the simulated execution that have to do with the interaction with the lth instance of ϕ, denoted ϕ_l. A bit more specifically, whenever some simulated party running ρ passes an input x to ϕ_l, $\mathcal{E}_\pi$ passes input x to the corresponding actual party. Outputs generated by an actual party running π are treated like outputs from ϕ_l to the corresponding simulated party running ρ. (Since π and ϕ are subroutine respecting, we are guaranteed that the only inputs and outputs between the external protocol instance and $\mathcal{E}_\pi$ are done via the inputs and outputs of the parties themselves.) Furthermore, whenever the simulated adversary $\mathcal{S}$ passes input value v to the instance of $\mathcal{S}_\pi$ that corresponds to ϕ_l, $\mathcal{E}_\pi$ passes input v to the actual adversary it interacts with. Any output obtained from the actual adversary is passed to the simulated $\mathcal{S}$ as an output from the corresponding instance of $\mathcal{S}_\pi$. Once the simulated $\mathcal{E}$ halts, $\mathcal{E}_\pi$ halts and outputs whatever $\mathcal{E}$ outputs. Figure 8 presents a graphical depiction of the operation of $\mathcal{E}_\pi$.

The proof is completed by observing that, if $\mathcal{E}_\pi$ interacts with $\mathcal{S}_\pi$ and ϕ, then the view of the simulated $\mathcal{E}$ within $\mathcal{E}_\pi$ has the same distribution as the view of $\mathcal{E}$ when interacting with $\mathcal{S}$ and ρ_l. Similarly, if $\mathcal{E}_\pi$ interacts with $\mathcal{D}$ and parties running π, then the view of the simulated $\mathcal{E}$ within $\mathcal{E}_\pi$ has the same distribution as the view of $\mathcal{E}$ when interacting with $\mathcal{S}$ and ρ_{l-1}.

5.2.2 A detailed proof

We proceed with a detailed proof of Theorem 13, substantiating the above outline.

Construction of $\mathcal{S}$. Let ρ, ϕ, π be protocols, where π emulates ϕ, and and let $\rho^\pi = \rho^{\phi\rightarrow\pi}$ be the composed protocol. The fact that π UC-emulates ϕ guarantees that there exists an ideal-process adversary $\mathcal{S}_\pi$ such that $\textsc{exec}_{\phi,\mathcal{S}_\pi,\mathcal{E}_\pi} \approx \textsc{exec}_{\pi,\mathcal{E}_\pi}$ holds for any environment $\mathcal{E}_\pi$. Adversary $\mathcal{S}$ uses $\mathcal{S}_\pi$ and is presented in Figure 9. We use the following terminology: An instance of protocol α in an execution is called top-level if it is not a subsidiary of any other instance of α in that execution.

[13]In the actual proof we consider a different model of computation for each hybrid, rather than considering a different protocol. The reason is that the parties running the protocol may not know which is the (globally) lth instance to be invoked. Also, for this argument we only consider the top-level instances of ϕ as discussed above. See details within.

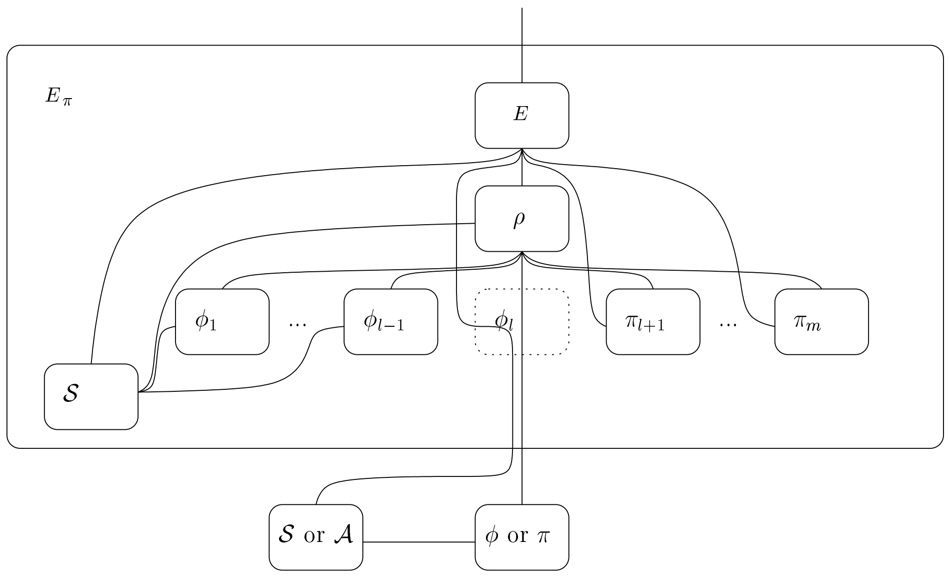

Figure 8: The operation of $\mathcal{E}_\pi$. An interaction of $\mathcal{E}$ with π is simulated, so that the first $l-1$ instances of ϕ remain unchanged, the lth instance is mapped to the external execution, and the remaining instances of ϕ are replaced by instances of π. For graphical clarity we use a single box to represent an instance of a multi-party protocol.

Validity of $\mathcal{S}$. First, note that $\mathcal{S}$ is PPT. In fact, the polynomial $p(\cdot)$ bounding the running time of $\mathcal{S}$ can be set to be the polynomial bounding the running time of $\mathcal{S}$, plus a linear polynomial. (Note that $p(\cdot)$ does not depend on the number of instances of ρ. The linear polynomial is needed to take care of relaying the messages of protocol π.)

Another point to note is that $\mathcal{S}$ can determine, upon receiving an incoming message, whether the sender ITI is a party or subsdiary of an instance of ϕ. Indeed, since ϕ is subroutine respecting, $\mathcal{S}$ gets notified whenever a subsidiary of an instance of ϕ is created, and can thus keep record of which ITIs are subsidiaries of each instance of ϕ, and which is the corresponding top-level instance of ϕ.

Similarly, since π is subroutine respecting, $\mathcal{S}$ gets notified (by the instances of $\mathcal{S}_\pi$ that it runs locally) whenever a subsidiary of an instance of π is created, and can thus keep record of which ITIs are subsidiaries of each instance of ϕ in the simulated execution of ρ^π that $\mathcal{S}$ mimics for $\mathcal{E}$. This allows $\mathcal{S}$ to determine, upon receiving from $\mathcal{E}$ a message to be delivered, whether the recipient is a party or subsidiary of an instance of π, and which is the corresponding top-level instance of π.[14]

Next, assume that there exists an environment machine $\mathcal{E}$ that violates the validity of $\mathcal{S}$ (that is, $\mathcal{E}$ violates Equation (2)). We construct an environment machine $\mathcal{E}_\pi$ that violates the validity of $\mathcal{S}_\pi$ with respect to a single run of π. (That is, $\mathcal{E}_\pi$ violates Equation (3).) More specifically, fix some input value z and a value k of the security parameter, and assume that

$$\text{EXEC}_{\rho^\pi,\mathcal{E}}(k,z) - \text{EXEC}_{\rho,\mathcal{S},\mathcal{E}}(k,z) \geq e. \tag{4}$$

[14] We note that the requirement that π and ϕ expose their subroutine structure is necessary: [HS11] demonstrate that without it the UC theorem may fail.

Adversary $\mathcal{S}$

Adversary $\mathcal{S}$ proceeds as follows, interacting with parties running protocol ρ and environment $\mathcal{E}$.

1. When activated with input (m, id, c) (coming from $\mathcal{E}$), where m is a message, $id = (sid, pid)$ is an identity, and α is a code for an ITM, do:
   (a) If the specified recipient is a party of a top-level instance sid' of π, or a subsidiary thereof, then first locate the internally running instance $\mathcal{S}_{(sid,\top)}$ of $\mathcal{S}_\pi$ that handles the protocol instance with SID= sid'. If no such instance of $\mathcal{S}_\pi$ is found, then internally invoke a new instance of $\mathcal{S}_\pi$ with identity $(sid', \top)$. Next, activate this instance of $\mathcal{S}_\pi$ with input (m, id, α) and follow its instructions.
   (b) Else (i.e., the specified recipient is not a party or subsidiary of an instance of π), deliver the message m to the recipient. Using the terminology of Definition 1, this means that $\mathcal{S}$ executes an external-write request to the incoming message tape of an ITI (c, id).
2. When activated with an incoming message m from an ITI with ID $id = (sid, pid)$ and code α, do:
   (a) If the sending ITI is a party of a top-level instance sid' of protocol ϕ, or a subsidiary thereof, then internally activate the instance $\mathcal{S}_{(sid',\top)}$ of $\mathcal{S}$ with incoming message m from (id, α), and follow its instructions. If no such instance of $\mathcal{S}_\pi$ exists then invoke it, internally, and label it $\mathcal{S}_{(sid',\top)}$.
   (b) Else, pass output (m, id, α) to $\mathcal{E}$.
3. When an instance of $\mathcal{S}_\pi$ internally generates a request to deliver a message m to some party, then deliver m to this party. When an instance of $\mathcal{S}_\pi$ requests to pass an output v to its environment then output v to $\mathcal{E}$, but with the the exception that $\mathcal{S}$ mimics the time bounds of a dummy adversary. That is, $\mathcal{S}$ stops delivering output to $\mathcal{E}$ as soon as the output length exceeds the overall input length of $\mathcal{S}$.

Figure 9: The adversary for protocol ρ.

We show that

$$\text{EXEC}_{\pi,\mathcal{D},\mathcal{E}_\pi}(k, z) - \text{IDEAL}_{\phi,\mathcal{S}_\pi,\mathcal{E}_\pi}(k, z) \geq e/t \tag{5}$$

where $t = t(k, |z|)$ is a polynomial function.

In preparation to constructing $\mathcal{E}_\pi$, we define the following distributions and make some observations on $\mathcal{S}$. Consider an execution of protocol ρ with adversary $\mathcal{S}$ and environment $\mathcal{E}$. Let $t = t(k, |z|)$ be an upper bound on the number of top-level instances of ϕ within ρ in this execution. (The bound t is used in the analysis only. The parties need not be aware of t. Also, t is polynomial in $k, |z|$ since $\mathcal{E}$ is PPT.) For $0 \leq l \leq t$, Let the l-hybrid model for running protocol ρ denote the extended system of ITMs that is identical to the basic model of computation, with the exception that the control function is modified as follows. (Recall that the control function of an extended system of ITMs determines, among other things, the target ITIs of external-write requests.) The external-write requests to tapes of the first l top-level instances of ϕ to be invoked are treated as usual. The external-write requests to the tapes of all other top-level instances of ϕ are directed to the corresponding instances of parties running π. That is, let sid_i denote the SID of the ith top-level instance of ϕ to be invoked in an execution. Then, given an external-write request made by an some ITI to the input tape of $\phi_{(sid_i,pid)}$ (i.e., to the party running ϕ with identity (sid_i, pid)) for some pid, where $i > l$, the control function writes the requested value to the input tape of $\pi_{(sid_i,pid)}$; if no such ITI exists then one is invoked. Note that these modifications apply to external-write requests by *any* ITI, including ITIs that participate in instances of ϕ and π, as well as subsidiaries thereof. Similarly, whenever $\pi_{(sid_i,pid)}$ requests to pass output to some ITI M, the control function changes the code of the sending ITI, as written on the subroutine output tape of M, to be ϕ. We let $\text{EXEC}^l_{\rho,\mathcal{A},\mathcal{E}}(k, z)$ denote the output of this system of ITMs on input z and security parameter k for the environment $\mathcal{E}$.

We observe that, when S interacts with $\mathcal{E}$ and parties running ρ in the l-hybrid model, it internally runs at most l instances of the simulator $\mathcal{S}_\pi$. (These are the instances that correspond to the first top-level l instances of protocol ϕ with which $\mathcal{S}$ interacts.) The remaining instances of $\mathcal{S}_\pi$ are replaced by interacting with the actual parties or sub-parties of the corresponding instances of π. In particular, we have that the output of $\mathcal{E}$ from an interaction with ρ and $\mathcal{S}$ in the t-hybrid model is distributed identically to the output of $\mathcal{E}$ from an interaction with ρ and $\mathcal{S}$ in the basic model, i.e. $\text{EXEC}^t_{\rho,\mathcal{S},\mathcal{E}} = \text{EXEC}_{\rho,\mathcal{S},\mathcal{E}}$. Similarly, the output of $\mathcal{E}$ from an interaction with ρ and $\mathcal{S}$ in the 0-hybrid model is distributed identically to the output of $\mathcal{E}$ from an interaction with ρ^π in the basic model of computation, i.e. $\text{EXEC}^0_{\rho,\mathcal{S},\mathcal{E}} = \text{EXEC}_{\rho,\mathcal{D},\mathcal{E}}$. Consequently, Inequality (4) can be rewritten as:

$$\text{EXEC}^0_{\rho,\mathcal{S},\mathcal{E}}(k, z) - \text{EXEC}^t_{\rho,\mathcal{S},\mathcal{E}}(k, z) \geq \epsilon. \tag{6}$$

We turn to constructing and analyzing environment $\mathcal{E}_\pi$. The construction of $\mathcal{E}_\pi$ is presented in Figure 10. We first note that $\mathcal{E}_\pi$ is PPT. This follows from the fact that the entire execution of the system is completed in polynomial number of steps. (Indeed, the polynomial bounding the runtime of $\mathcal{E}_\pi$ can be bounded by the maximum among the polynomials bounding the running times of $\mathcal{E}$, ρ, and $\rho^{\phi\to\pi}$.)

The rest of the proof analyzes the validity of $\mathcal{E}_\pi$, demonstrating (5). For $1 \leq l \leq t$, let $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}^l_\pi}(k, z)$ denote the distribution of $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}_\pi}(k, z)$ conditioned on the event that $\mathcal{E}_\pi$ chose hybrid l. We first argue that, for every k, z, and $1 \leq l \leq t$, we have:

$$\text{EXEC}_{\phi,\mathcal{S}_\pi,\mathcal{E}^l_\pi}(k, z) = \text{EXEC}^l_{\rho,\mathcal{S},\mathcal{E}}(k, z) \tag{7}$$

and

$$\text{EXEC}_{\pi,\mathcal{D},\mathcal{E}^l_\pi}(k, z) = \text{EXEC}^{l-1}_{\rho,\mathcal{S},\mathcal{E}}(k, z). \tag{8}$$

Equations (7) and (8) follow from inspecting the code of $\mathcal{E}_\pi$ and $\mathcal{S}$. In particular, if $\mathcal{E}_\pi$ interacts with parties running ϕ then the view of the simulated $\mathcal{E}$ within $\mathcal{E}_\pi$ is distributed identically to the view of $\mathcal{E}$ when interacting with ρ and $\mathcal{S}$ in the l-hybrid model. Similarly, if $\mathcal{E}_\pi$ interacts with parties running π then the view of the simulated $\mathcal{E}$ within $\mathcal{E}_\pi$ is distributed identically to the view of $\mathcal{E}$ when interacting with ρ and $\mathcal{S}$ in the $(l-1)$-hybrid model. (Here it is important to note that, since both π and ϕ are subroutine respecting, the only communication between the external instance of π or ϕ, and the ITIs outside this instance, is the inputs and outputs of the main parties of this instance.)

From Equations (6), (7) and (8) it follows that:

$$|\text{EXEC}_{\pi,\mathcal{D},\mathcal{E}_\pi}(k, z) - \text{EXEC}_{\phi,\mathcal{S}_\pi,\mathcal{E}_\pi}(k, z)| = |\frac{1}{t}\sum_{l=1}^{t}(\text{EXEC}^l_{\rho,\mathcal{S},\mathcal{E}}(k, z) - \text{EXEC}^{l-1}_{\rho,\mathcal{S},\mathcal{E}}(k, z))| \geq \epsilon/t \tag{9}$$

in contradiction to the assumption that $\mathcal{S}_\pi$ is a valid simulator for π.

Environment $\mathcal{E}_\pi$

Environment $\mathcal{E}_\pi$ proceeds as follows, given a value k for the security parameter, input z, and expecting to interact with parties running a single instance of π. We first present a procedure called Simulate(). Next we describe the main program of $\mathcal{E}_\pi$.

Procedure Simulate(s, l)

1. Expect the parameter s to contain a global state of a system of ITMs representing an execution of protocol ρ in the l-hybrid model, with adversary $\mathcal{S}$ and environment $\mathcal{E}$. Continue a simulated execution from state s (making the necessary random choices along the way), until one of the following events occurs. Let sid_l denote the SID of the lth top-level instance of ϕ to be invoked in the simulated execution.
   (a) Some simulated ITI with identity id passes input x to an ITI id' which is a party or subsidiary of instance SID_l of ϕ. In this case, save the current state of the simulated system in s, pass input x from claimed source id to the external ITI id', and complete this activation.
   (b) The simulated $\mathcal{E}$ passes input (m, id) to the simulated adversary $\mathcal{S}$, where id is the extended identity of an ITI which is a party or subsidiary of instance SID_l of ϕ. In this case, save the current state of the simulated system in s, pass the input (m, id) to the external dummy adversary, and complete this activation.
   (c) The simulated environment $\mathcal{E}$ halts. In this case, $\mathcal{E}_\pi$ outputs whatever $\mathcal{E}$ outputs and halts.

Main program for $\mathcal{E}_\pi$:

1. When activated for the first time, with input z, choose $l \overset{R}{\leftarrow} \{1..t\}$, and initialize a variable s to hold the initial global state of a system of ITMs representing an execution of protocol π in the l-hybrid model, with adversary $\mathcal{S}$ and environment $\mathcal{E}$ on input z and security parameter k. Next, run Simulate(s, l).
2. In any other activation, let x be the new value written on the subroutine-output tape. Next:
   (a) Update the state s. That is:
      i. If the new value x was written by one of the main parties of the external protocol instance, then write x to the subroutine-output tape of the simulated party that's specified in the output x. (Recall that values written to the subroutine output tape of the environment include an extended identity of a target ITI.)
      ii. If the new value x was written by the external adversary, then update the state of the simulated adversary $\mathcal{S}$ to include an output v generated by the instance of $\mathcal{S}_\phi$ that corresponds to instance SID_l of ϕ.
   (b) Simulate an execution of the system from state s. That is, run Simulate(s, l).

Figure 10: The environment for a single instance of π.

5.3 Discussion and extensions

Some aspects of the universal composition theorem were discussed in Section 2.3. This section highlights additional aspects, and presents some extensions of the theorem.

On composability with respect to closed environments. Recall that the closed-environment variant of the definition of emulation (Definition 5) considers only environments that take external input that contains no information other than its length, e.g. inputs of the form 1^n for some n. We note that the UC theorem still holds even for this variant, with the same proof.

Composing multiple different protocols. The composition theorem (Theorem 13) is stated only for the case of replacing instances of a *single* protocol ϕ with instances of another protocol. The theorem holds also for the case where multiple different protocols $\phi_1, \phi_2, ...$ are replaced by protocols $\rho_1, \rho_2, ...$, respectively. (This can be seen either by directly extending the current proof, or by defining a single "universal" protocol that mimics multiple different ones.)

A quantitative statement of the UC theorem. We observe that the polynomial bounding the running time of the constructed environment $\mathcal{E}_\pi$ is no larger than the polynomial bounding the running time of the given environment $\mathcal{E}$. From this, along with the fact that the distinguishing probability of $\mathcal{E}_\pi$ decreases proportionally to the number of instances of ρ, we have that if ρ (k, ϵ, g)-UC-emulates ϕ for some value k of the security parameter, then protocol $\pi^{\rho/\phi}$ (k, ϵ', g)-UC-emulates protocol π, where $\epsilon'(p_\mathcal{A}, P_\mathcal{E})() = t \cdot \epsilon(c \cdot p_\mathcal{A}, c \cdot p_\mathcal{E})()$, t is a bound on the number of instances of ρ in π^ρ, and c is some small constant. That is, the emulation slack increases by a factor of roughly t, and the simulation overhead remains the same.

An alternative proof of the UC theorem. The above proof of Theorem 13 constructs, in a single step, a simulator $\mathcal{S}_\pi$ that handles all the instances of ρ. An alternative proof proceeds as follows:

1. Prove Theorem 13 for the case where protocol π calls only a single instance of ϕ, or equivalently when only one instance of ϕ, out of potentially many instances called by π, is replaced with an instance of ρ. Call this theorem the "single-instance UC theorem". Proving this theorem is considerably simpler than the current proof; in particular, the hybrids argument is not necessary. Furthermore, in this case the UC theorem preserves both the emulation slack ϵ and the simulation overhead g.

2. To handle replacement of polynomially many concurrent instances of ϕ by instances of ρ, simply apply the "single-instance UC theorem" iteratively, where in each iteration a new instance of ϕ is replaced by an instance of ρ and the remaining instances of ρ, ϕ and π are treated as part of the environment. Then, use the transitivity of UC-emulation to deduce that π^ρ UC-emulates π.

When comparing this proof to the direct proof in Section 5.2, one should take into account the degradation in the quality of the emulation, namely the growth of the emulation slack ϵ and the simulation overhead g. Recall that, by transitivity, the emulation slack increases additively under iterative applications of UC-emulation (see discussion on page 39). This means that the emulation slack grows linearly in the number of instances of the composed protocol; this is the same as in the proof of Section 5.2.

However, the proof of Section 5.2 does better in terms of the simulation overhead: In that proof, there is no increase in the simulation overhead. In contrast, in the alternative proof mentioned here the simulation overhead when composing t instances is the t-wise composition of the overheads incurred by the t applications of the UC theorem. Therefore, the simulation overhead increases proportionally to the number of composed instances. Still, the increase is moderate: By Claim 8, we have that each such overhead is an additive polynomial $p()$ that depends only on protocols ρ and ϕ. This means that the simulation overhead increases by an additive factor of $tp()$.

Nesting of protocol instances. The universal composition operation can be applied repeatedly to perform "nested" replacements of calls to sub-protocols with calls to other sub-protocols. We observe that repeated application of the composition operation preserves security. For instance, if a protocol ρ_1 UC-emulates protocol ϕ_1, and protocol ρ_2 UC-emulates protocol ϕ_2 using calls to ϕ_1, then for any protocol π that uses calls to ϕ_2 it holds that the composed protocol $\pi^{(\rho_2^{\rho_1/\phi_1})/\phi_2} = \mathrm{UC}(\pi, \mathrm{UC}(\rho_2, \rho_1, \phi_1), \phi_2)$ UC-emulates π.

Security for nested applications of universal composition can be argued by repeated application of the UC theorem, similarly to the alternative proof of the UC theorem, sketched above. As there, the emulation slack and the simulation overhead increase linearly in the number of composed instances, regardless of whether these instances are nested or not. In addition, it is possible to extend the direct proof of Section 5.2 to deal with the case of nested applications of the UC operation with arbitrary polynomial depth, and with no increase in the simulation overhead. We omit further details. The fact that the UC theorem extends to arbitrary polynomial nesting of the UC operation was independently observed in [BM04] for their variant of the UC framework.

Beyond PPT. The UC theorem is stated and proven for PPT systems of ITMs, namely for the case where all the involved entities are PPT. It is readily seen that the theorem holds also for other classes of ITMs and systems, as long as the definition of the class guarantees that any execution of any system of ITMs can be "simulated" on a single ITM from the same class.

More precisely, say that a class $\mathcal{C}$ of ITMs is self-simulatable if, for any system (I, C) of ITMs where both I and C (in its ITM representation) are in $\mathcal{C}$, there exists an ITM M in $\mathcal{C}$ such that, on any input and any random input, the output of a single instance of M equals the output of (I, C). (Stated in these terms, Proposition 3 on page 30 asserts that for any super-additive function $T()$, the class of ITMs that run in time $T()$ is self-simulatable.)

Say that protocol π UC-emulates protocol ϕ with respect to class $\mathcal{C}$ if Definition 5 holds when the class of PPT ITMs is replaced with class $\mathcal{C}$, namely when π, $\mathcal{A}$, $\mathcal{S}$, and $\mathcal{E}$ are taken to be ITMs in $\mathcal{C}$. Then we have:

Proposition 16 *Let $\mathcal{C}$ be a self-simulatable class of ITMs, and let π, ρ, ϕ be protocols in $\mathcal{C}$ such that ρ UC-emulates ϕ with respect to class $\mathcal{C}$. Then protocol $\pi^{\rho/\phi}$ UC-emulates protocol π with respect to class $\mathcal{C}$.*

It is stressed, however, that the UC theorem is, in general, *false* in settings where systems of ITMs cannot be simulated on a single ITM from the same class. We exemplify this point for the case where all entities in the system are bound to be PPT, except for the protocol ϕ which is not PPT.[15] More specifically, we present an ideal functionality $\mathcal{F}$ that is not PPT, and a PPT protocol π that UC-realizes $\mathcal{F}$ with respect to PPT environments. Then we present a protocol ρ, that calls two instances of the ideal protocol for $\mathcal{F}$, and such that $\rho^{\pi/\mathcal{F}}$ does not UC-emulate π. In fact, for *any* PPT π' we have that $\rho^{\pi'/\mathcal{F}}$ does not emulate ρ.

[15]We thank Manoj Prabhakaran and Amit Sahai for this example.

In order to define $\mathcal{F}$, we first recall the definition of pseudorandom ensembles of evasive sets, defined in [GK89] for a related purpose. An ensemble $\mathcal{S} = \{S_k\}_{k\in\mathbf{N}}$ where each $S_k = \{s_{k,i}\}_{i\in\{0,1\}^k}$ and each $s_{k,i} \subset \{0,1\}^k$ is a pseudorandom evasive set ensemble if: (a) $\mathcal{S}$ is pseudorandom, that is for all large enough $k \in \mathbf{N}$ and for all $i \in \{0,1\}^k$ we have that a random element $x \xleftarrow{\text{R}} s_{k,i}$ is computationally indistinguishable from $x \xleftarrow{\text{R}} \{0,1\}^k$. (b) $\mathcal{S}$ is evasive, that is for any non-uniform PPT algorithm A and for any $z \in \{0,1\}^*$, we have that $\text{Prob}_{i\xleftarrow{\text{R}}\{0,1\}^k}[A(z,i) \in s_{k,i}]$ is negligible in k, where $k = |z|$. It is shown in [GK89], via a counting argument, that pseudorandom evasive set ensembles exist.

Now, define $\mathcal{F}$ as follows. $\mathcal{F}$ uses the ensemble $\mathcal{S}$ and interacts with one party only. Given security parameter k, it first chooses $i \xleftarrow{\text{R}} \{0,1\}^k$ and outputs i. Then, given an input $(x, i') \in \{0,1\}^k \times [2^k]$, it first checks whether $x \in s_{k,i}$. If so, then it outputs `success`. Otherwise it outputs $r \xleftarrow{\text{R}} s_{k,i'}$.

Protocol π for realizing $\mathcal{F}$ is simple: Given security parameter k it outputs $i \xleftarrow{\text{R}} \{0,1\}^k$. Given an input $x \in \{0,1\}^k$, it outputs $r \xleftarrow{\text{R}} \{0,1\}^k$. It is easy to see that π UC-realizes $\mathcal{F}$: Since $\mathcal{S}$ is evasive, then the probability that the input x is in the set $s_{k,i}$ is negligible, thus $\mathcal{F}$ outputs `success` only with negligible probability. Furthermore, $\mathcal{F}$ outputs a pseudorandom k-bit value, which is indistinguishable from the output of π.

Now, consider the following $\mathcal{F}$-hybrid protocol ρ. ρ runs two instances of $\mathcal{F}$, denoted $\mathcal{F}_1$ and $\mathcal{F}_2$. Upon invocation with security parameter k, it activates $\mathcal{F}_1$ and $\mathcal{F}_2$ with k, and obtains the indices i_1 and i_2. Next, it chooses $x_1 \xleftarrow{\text{R}} \{0,1\}^k$, and feeds (x_1, i_2) to $\mathcal{F}_1$. If $\mathcal{F}_1$ outputs `success` then ρ outputs `success` and halts. Otherwise, π feeds the value x_2 obtained from $\mathcal{F}_1$ to $\mathcal{F}_2$. If $\mathcal{F}_2$ outputs `success` then ρ outputs `success`; otherwise it outputs `fail`. It is easy to see that ρ always outputs success. However, $\rho^{\pi/\mathcal{F}}$ never outputs success. Furthermore, for *any* PPT protocol π' that UC-realizes $\mathcal{F}$, we have that $\rho^{\pi'/\mathcal{F}}$ outputs `success` only with negligible probability.

6 UC formulations of some computational models

Recall that the basic model of computation, captured in Figure 5 on page 35, provides only a very rudimentary way for communication between parties. Specifically, the communication is completely controlled by the adversary, in terms of both the contents and the timing of delivered messages. Also, no specific provisions for party corruption are given. In fact, this model is not intended to be a viable model of computation at all. Rather, it is intended to provide a formal basis on top of which more realistic — and more abstract — models of computation can be formulated.

As sketched and motivated in the Introduction (Section 1.3), communication models can be captured within the present framework by way of formulating appropriate ideal functionalities. That is, we say that an ideal functionality $\mathcal{F}$ represents some communication model if protocols in this communication model can be represented by considering protocols that make subroutine calls to $I_{\mathcal{F}}$. Using the same approach, a corruption model is captured by adding to the protocol description a set of instructions to be executed upon receiving a special "corruption" message from the adversary.

This section exemplifies this approach by presenting ideal functionalities that are aimed at capturing some salient communication models. Specifically, we address authenticated and secure communication, synchronous communication, and non-concurrent protocol execution. We also formulate within the framework some standard models for party corruption.

In addition to presenting the specific ideal functionalities considered, this section intends to exemplify the use of the framework and to provide the reader with basic techniques for writing ideal functionalities. For this purpose, we also set some general writing conventions for ideal functionalities, and demonstrate their use.

We start with presenting some basic corruption models, in Section 6.1. We then turn to writing ideal functionalities: Section 6.2 presents general writing conventions. Sections 6.3 ad 6.4 present the ideal functionalities for capturing authenticated and secure communication, respectively. Section 6.5 present the ideal functionality capturing synchronous communication, and Section 6.6 presents the ideal functionality capturing non-concurrent protocol execution.

6.1 Some corruption models

The operation of *party corruption* is a common basic construct in modeling and analyzing the security of cryptographic protocols. Party corruption is used to capture a large variety of concerns and situations, including preserving secrecy in face of eavesdroppers, resilience to adversarial ("Byzantine") behavior by protocol participants, resilience to viruses and software exploits, resilience to side channel attacks, etc.

The basic model of protocol execution and the definition of protocol emulation from Section 4 do not provide an explicit mechanism for modeling party corruption. Instead, this section demonstrates how party corruption can be modeled via a set of conventions regarding protocol instructions to be performed upon receiving a special message from the adversary. We argue that this choice keeps the basic model simpler and cleaner, and at the same time provides greater flexibility in capturing a variety of concerns via the corruption mechanism.

First we need a way to separate separate between protocol instructions that represent instructions that represent "real code" and ones that represent artificial model operations. We do this as follows. Say that a protocol is compliant if it consists of two separate parts, or "sub-processes," called a body and a shell. The shell; however the body does not have access to the state of the shell. Each incoming message, input or subroutine output is first processed by the shell, which then potentially forwards this message (or some function of it) to the body. Similarly, each external write instruction executed by the body is passed to the shell, who may modify it before sending it out. From now on we assume all protocols are compliant. The conventions below specify the behavior of the shell, which represents artificial model instructions. The body of the protocol remains unrestricted.

To corrupt an ITI M, the adversary delivers a $(\texttt{corrupt}, p)$ message to M, where p is some parameter of the corruption. Say that a protocol π is standard corruption if, as soon as the shell of am ITI M running π receives a $(\texttt{corrupt}, p)$ message, it first passes a $(\texttt{corrupt}, p, M)$ output to all "parents" of M (i.e., to all ITIs of which M is a subroutine). Furthermore, when activated with a $(\texttt{corrupt}, p, M')$ subroutine output, the shell of M passes output $(\texttt{corrupt}, p, M')$ to all parent ITIs. This mechanism guarantees that the fact that some ITI was corrupted immediately "percolates" all the way to the environment. This guarantees that party corruption operations can be accounted for in the analysis. (To hide its subroutine structure, protocol π may choose to modify or hide some of the the information regarding M, the corrupted ITI. See more details below.)

Byzantine corruption. Perhaps the simplest form of corruption to capture is total corruption, often called Byzantine corruptions. A protocol is Byzantine corruptions if, upon receiving the (`corrupt`) message, the shell first complies with the above standard corruption requirement. From this point on, any message received on the incoming communication tape is interpreted as an instruction to write a given value to a tape of another ITI; this instruction is carried out. In an activation due to an incoming input or subroutine output, the protocol instructs to send the entire local state to the adversary. (If the code of M is non-erasing, namely it only writes to each memory location once, then this state includes all past states of M.) Note that here the body of π becomes completely inactive from the time of corruption on.

Non-adaptive (static) corruptions. The above formulation of Byzantine corruption captures adaptive party corruptions, namely corruptions that occur as the computation proceeds, based on the information gathered by the adversary so far. It is sometimes useful to consider also a weaker threat model, where the identities of the adversarially controlled parties are fixed before the computation starts; this is the case of non-adaptive (or, static) adversaries. In the present framework, a protocol is static corruption if it instructs, upon invocation, to send a notification message to the adversary; a corruption message is considered only in it is delivered in the very next activation. Later corruption messages are ignored.

Passive (honest-but-curious) corruptions. Byzantine corruptions capture situations where the adversary obtains total control over the behavior of corrupted parties. Another standard corruption model only allows the adversary to *observe* the internal state of the corrupted party. We call such adversaries passive. Passive corruptions can be captured by changing the reaction of the shell of a party to a (`corrupt`) message from the adversary, as follows. A protocol π is passive corruptions if it proceeds as follows from the point when a (`corrupt`) message has been received. Upon receipt of an incoming input or subroutine output, the shell activates the body, and at the end of the activation sends the internal state to the adversary. If the next activation is due to an incoming (`continue`) message from the adversary, then the shell performs the external write operation instructed by the body in the previous activation. Else the shell halts for the remainder of the protocol execution. when activated due to anther incoming message from the adversary, the shell forwards the message to the body, and follows the instructions of the body in this activation.

We remark that one can consider two variants of passive corruptions, depending on whether the adversary is allowed to *modify* the inputs of the corrupted parties. The two variants can be captured via appropriate sets of instructions for parties upon corruption.

Transient (mobile) corruptions and proactive security. All the corruption methods so far represent "permanent" corruptions, in the sense that once an ITI gets corrupted it remains corrupted throughout the computation. Another variant allows ITIs to "recover" from a corruption and regain their security. Such corruptions are often called mobile (or, transient). Security against such corruptions is often called proactive security. Transient corruptions can be captured by adding a (`recover`) message from the adversary. Upon receiving a `(recover)` message, the ITI stops reporting its state to the adversary, and stops following the adversary's instructions.

Coercion. In a coercion attack an external entity tries to influence the input that the attacked party contributes to a computation, without physically controlling the attacked party at the time where the input is being contributed. The typical coercion method is to ask the coerced party to reveal, at some later point in time, its local state at time of obtaining the secret input, and then verifying consistency with the public transcript of the protocol. Resilience to coercion is an important requirement in setting where the participants are humans that are susceptible to social pressure, such as in voting schemes.

In the present framework, coercion attacks can be modeled in a straightforward way, along the lines of [CG96]. That is, upon receipt of a coercion message, the shell notifies the body of the corruption and follows the instructions of the body. This allows the attacked party to run some pre-determined algorithm for modifying its internal state before handing it over to the adversary.

Physical ("side channel") attacks. A practical and very realistic security concern is protection against "physical attacks" on computing devices, where the attacker is able to gather information on, and sometimes even modify, the internal computation of a device via physical access to it. (Examples include the "timing attack" of [K96], the "microwave attacks" of [BDL97, BS97] and the "power analysis" attacks of [CJRR99].) These attacks are often dubbed "side-channel" attacks in the literature. Some formalizations of security against such attacks appear in [MR04, GLMMR04].

This type of attacks can be directly modeled via different reaction patterns of parties to corruption messages. For instance, the ability of the adversary to observe certain memory locations, or to detect whenever a certain internal operation (such as modular multiplication) takes place, can be directly modeled by having the corrupted ITI send to the adversary an appropriate function of its internal state.

However, this modeling is rather limited since it allows the adversary to only obtain leakage information from a single process (or, ITI). Instead, we would like to be able to capture leakage that is taken from a physical device that runs several processes. This is one of the motivations to the notion of PID-wise corruption, discussed next.

PID-wise corruption. An ITI is an abstract construct that captures a process, or an instance of a running program. It is not directly associated with a physical computing device. Indeed, it is often convenient to consider different processes that run on the same physical computer as separate ITIs, thus "abstracting out" the fact that these processes actually share resources such as memory, CPU and network access. Furthermore, it is possible to capture different components of modern computers and their operating systems as ITIs within the present model, thus enabling rigorous analysis of their security (see e.g. [C+11]). In all of these cases, it is often convenient to provide a mechanism for corrupting a set of ITIs "together". (For instance, we may want all the ITIs that represent processes that "run on the same physical computer," or all the ITIs that use the same public key, to be corrupted together.) We capture this requirement as follows: Assume for simplicity that all the ITIs that are to be corrupted together have the same PID (or the same value of some field in the PID). Furthermore, we assume that the protocols in question are such that all the ITIs that have the same PID are in the same "ancestry tree", namely all ITIs with a given PID are subsidiaries of a single ITI with that PID. Then, upon receiving a (`corrupt`) message, the shell of the corrupted ITI passes along the tree inputs and outputs to all the ITIs with the same PID, notifying them of the corruption. From now on, all the ITIs with that PID behave as corrupted.

In the case of PID-wise leakage a more sophisticated mechanism is required, for aggragating the local states of all ITIs with the same PID, and computing leakage on the aggragate state. See [BCH12] for more details.

So far we only dealt with the modeling of various actual adversarial attacks on protocols by way of specific protocol instructions. Specifying security requirements in face of corruptions is done by specifying the response of the change once more than some number of parties have been corrupted. We give some examples in Section 6.2.

6.2 Writing conventions for ideal functionalities

The model is designed so as to allow expressing a large variety of concerns and expected behavior patterns in face of a wide variety of attacks. Still, in of itself the basic model is rather rudimentary. This section presents some conventions and mechanisms that facilitate expressing common concerns and requirements within the present formalism.

Specifying the identities of the calling parties. It is often important to allow an ideal functionality $\mathcal{F}$ to verify the identites of its parent ITIs, namely the ITIs that provide it with input. Similarly, an ideal functionality should be able to determine the identities of the parties on whose subroutine output tape it is writing. Furthermore, this should hold not only with respect to the dummy parties in the ideal protocol for $\mathcal{F}$. Rather, $\mathcal{F}$ often needs access to the identities of the parents of the dummy parties in the ideal protocol for $\mathcal{F}$. (A quintessential example for this need is $\mathcal{F}_{\text{AUTH}}$, the message authentication functionality, described in the next section.) We note that the framework makes this information available to $\mathcal{F}$; in particular, the code of the dummy parties contains the identites and code of their parent ITIs.

When writing ideal functionalities, we allow ourselves to say "receive input v from party P" and mean "upon activation with an input value v, verify that the writing ITI is a dummy party whose (single) parent is an ITI with identity P". Similarly we say "generate output v for party P", meaning "perform an external write operation of value v to a dummy ITI whose prescribed parent has extended identity P." Recall that the dummy ITI and its parent may actually be *created* as a result of this write instrution. Also, we slightly abuse terminology and say that an ITI P is a parent of $\mathcal{F}$ even when P is a parent of a dummy party in the ideal protocol for $\mathcal{F}$.

SID conventions. It will be convenient to write ideal functionalities so that functionality never expects to get an input from an ITI (other than its initial invoker), before passing an output to this ITI. In other words, the functionality first 'announces its existence" (and its SID) to an ITI, and only then expects an input from this ITI. This convention is in line with protocol design principles in anynchronous networks. It also means that there is no need for prior agreement on the SID of an ideal functionality: The SID is determined by the first invoker of the instance of $\mathcal{F}$. It will also be convenient to include within the SID the identity of the initiator (namely, the identity of the parent that invoked $\mathcal{F}$), and sometimes even the identities of the intended future parents of $\mathcal{F}$.

Note that the SID is typically treated as a public value, so if it is desired that the identities of the participants be kept secret then some aliasing mechanism should be used (say, the SID includes a list of aliases and the functionality is privately given the identity that corresponds to each alias).

Behavior upon party corruption. In the ideal protocol $I_{\mathcal{F}}$, corruption of a parties is modeled as messages delivered by the adversary to the dummy parties and the ideal functionality $\mathcal{F}$. By convention, corruption messages delivered to the dummy parties are ignored. This makes sure that decisions about behavior upon party corruption are made only within $\mathcal{F}$. Indeed, the behavior of $\mathcal{F}$

upon receipt of a corruption message is an important part of the security specification represented by $\mathcal{F}$.

We set some conventions for this behavior. While we do not mandate that an ideal functionality is writen as a compliant protocol, we set a notion of standard corruption which is similar to the corresponding notion for general protocols. We say that an ideal functionality $\mathcal{F}$ is standard corruption if it proceds as follows upon receiving a `(corrupt` P`)` message from the adversary $\mathcal{S}$, where P is an identity of a dummy party for the present instance of $I_{\mathcal{F}}$: First, $\mathcal{F}$ marks P as corrupted and passes output to P notifying it of the corruption. In the next activation, $\mathcal{F}$ sends to the adversary all the inputs and outputs of P so far. In addition, from this point on, whenever $\mathcal{F}$ gets an input value v from P, it forwards v to the adversary, and receives a "modified input value" v' from the adversary. Also, all output values intended for P are sent to the adversary instead. This captures the standard behavior of the ideal process upon corruption of a party in existing definitional frameworks. The functionalities presented in this section are standard-corruption unless explicitly said otherwise.[16]

Delayed output. Recall that an output from an ideal functionality to a party is read by the recipient immediately, in the next activation. In contrast, we often want to be able to represent the fact that outputs generated by distributed protocols are inevitably delayed due to delays in message delivery. One natural way to relax an ideal functionality so as to allow this slack is to have the functionality "ask for the permission of the adversary" before generating an output. More precisely, we say that an ideal functionality $\mathcal{F}$ sends a delayed output v to party P if it engages in the following interaction: Instead of simply outputting v to P, $\mathcal{F}$ first sends to the adversary a message that it is ready to generate an output to P. If the output is public, then the value v is included in the message to the adversary. If the output is private then v is not not mentioned in this message. Furthermore, the message contains a unique identifier that distinguishes it from all other messages sent by $\mathcal{F}$ to the adversary in this execution. When the adversary replies to the message (say, by echoing the unique identifier), $\mathcal{F}$ outputs the value v to P.

Sending public delayed output v to a set $\mathcal{P}$ of parties is carried out as follows: First, $\mathcal{F}$ sends $(v, \mathcal{P})$ to the adversary. Then, whenever the adversary responds with an identity $P \in \mathcal{P}$, $\mathcal{F}$ outputs v to P. If the output is private then the procedure is the same except that $\mathcal{F}$ does not send v to the adversary. Sending delayed output which is partially private and partially public is defined analogously.

Running arbitrary code. It is often convenient to let an ideal functionality $\mathcal{F}$ receive a description of an arbitrary code c from parties, or even directly from the adversary, and then run this code while inspecting some properties of it. One use of such technique is for writing ideal functionalities with only minimal, well-specified requirements from the implementation. For instance, $\mathcal{F}$ may receive from the adversary a code for an algorithm; it will then run this algorithm as long as some set of security or correctness properties are satisfied. If a required property is violated, $\mathcal{F}$

[16]We remark that other behavior patterns upon party corruption are sometimes useful in capturing realistic concerns. For instance, *forward secrecy* can be captured by making sure that the adversary does not obtain past inputs or outputs of the party even when the party is corrupted. Partial security reduction due to leakage can be captured by sending to the adversary only some function of the inputs and outputs. Furthermore, more global changes in behavior can be formulated, say letting the adversary learn some secret input as soon as more than some fraction of the participants are corrupted.

will output an error message to the relevant parties. Examples of this use include the signature and encryption functionalities as formalized in [CH11, C05]. Another use for this "programming technique" is to enable expressing the requirement that some adversarial processes be carried out in isolation from the external environment the the protocol runs in. An example for this use is the formulation of non-concurrent security in Section 6.6.

At first glance, this technique seems problematic in that $\mathcal{F}$ is expected to run algorithms of arbitrary polynomial running time, whereas its runtime is bounded by some fixed polynomial. We get around this problem by assuming that the entity that provides the code c explicitly invokes an ITI γ that runs the program c; γ then takes inputs from $\mathcal{F}$ and providing outputs to $\mathcal{F}$. This way, the "burden of providing sufficient resources to run c" is shifted to the entity that provides c. Note however that the length of the outputs of c should be bound by the polynomial bounding $\mathcal{F}$, otherwise $\mathcal{F}$ will not be able to read these outputs. Also, recall that the basic model allows $\mathcal{F}$ to know the real code run by this ITI. In particular, $\mathcal{F}$ can make sure that c does not provide any illegitimate output to parties other than $\mathcal{F}$.

6.3 Authenticated Communication

Ideally authenticated message transmission means that an entity R will receive a message m from an entity S only if S has sent the message m to R. Furthermore, if S sent m to R only t times then R will receive m from S at most t times. These requirements are of course meaningful only as long as both S and R follow their protocols, namely are not corrupted. In the case of adaptive corruptions, the authenticity requirement is meaningful only if both S and R are uncorrupted *at the time when R receives the message.*

In the present framework, protocols that assume ideally authenticated message transmission can be cast as protocols with access to an "ideal authenticated message transmission functionality". This functionality, denoted $\mathcal{F}_{\text{AUTH}}$, is presented in Figure 11. $\mathcal{F}_{\text{AUTH}}$ first waits to receive an input $(\texttt{Send}, S, R, sid, m)$ from a dummy party D, where S is the identity of the ITI that called D (hereforth called the sender), R is an identity for the intended reciver, sid is the local SID, and m is the message to be delivered. (It will be convenient to think of sid as containing S and R as subfields, but this is certainly not necessary for the modeling to work.) $\mathcal{F}_{\text{AUTH}}$ then generates a public delayed output $(\texttt{Sent}, S, R, sid, m)$ to R. That is, $\mathcal{F}_{\text{AUTH}}$ first sends this value to the adversary. When the adversary responds, $\mathcal{F}_{\text{AUTH}}$ writes this value to the subroutine output tape of a dummy party with SID sid and PID R. This dummy party then passes output $(\texttt{Sent}, S, sid, m)$ to an ITI with identity R.

$\mathcal{F}_{\text{AUTH}}$ is a standard corruption functionality. That is, if $\mathcal{F}_{\text{AUTH}}$ receives a message from the adversary to corrupt a party, then $\mathcal{F}_{\text{AUTH}}$ notifies the relevant dummy party of the corruption. In addition, if the sender gets corrupted before the output value was actually delivered to R, then $\mathcal{F}_{\text{AUTH}}$ allows the adversary to provide a new message m' of its choice. In this case, $\mathcal{F}_{\text{AUTH}}$ outputs $(\texttt{Corrupted})$ to S, outputs $(\texttt{Send}, S, sid, m')$ to R and halts.

We highlight several points regarding the security guarantees provided by $\mathcal{F}_{\text{AUTH}}$. First, $\mathcal{F}_{\text{AUTH}}$ reveals the contents of the message to the adversary. This captures the fact that secrecy of the message is not guaranteed. Second, the output is delayed, namely $\mathcal{F}_{\text{AUTH}}$ delivers a message only when the adversary responds, even if both the sender and the receiver are uncorrupted. This means that $\mathcal{F}_{\text{AUTH}}$ allows the adversary to delay a message indefinitely, and even to block delivery altogether. Third, $\mathcal{F}_{\text{AUTH}}$ allows the adversary to change the contents of the message, as long as

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

1. Upon receiving an input $(\texttt{Send}, S, R, sid, m)$ from ITI S, generate a public delayed output $(\texttt{Sent}, S, sid, m)$ to R and halt.
2. Upon receiving $(\texttt{Corrupt-sender}, sid, m')$ from the adversary, and if the $(\texttt{Sent}, S, sid, m)$ output is not yet delivered to R, then output $(\texttt{Sent}, S, sid, m')$ to R and halt.

Figure 11: The Message Authentication functionality, $\mathcal{F}_{\text{AUTH}}$. For simplicity, the dummy parties are omitted from this description. Also, the function is standard corruption, meaning that upon receiving a $(\texttt{Corrupt}, P)$ instructionm it outputs $(\texttt{Corrupted})$ to the dummy party with PID P.

the sender is corrupted *at the time of delivery,* even if the sender was uncorrupted at the point when it sent the message. This provision captures the fact that in general the received value is not determined until the point where the recipient actually generates its output.[17] Fourth, $\mathcal{F}_{\text{AUTH}}$ guarantees "non-transferable authentication": By interacting with $\mathcal{F}_{\text{AUTH}}$, the receiver R does not gain ability to run protocols with a third party V, whereby V reliably learns that the message was indeed sent by the sender. In situations where this strong guarantee is not needed, it might suffice to use an appropriately relaxed variant of $\mathcal{F}_{\text{AUTH}}$.

Next, let us highlight two modeling aspects of $\mathcal{F}_{\text{AUTH}}$. First, $\mathcal{F}_{\text{AUTH}}$ deals with authenticated transmission of a *single message.* Authenticated transmission of multiple messages is obtained by using multiple instances of $\mathcal{F}_{\text{AUTH}}$, and relying on the universal composition theorem for security. This is an important property: It allows different instances of protocols that use authenticated communication to use different instances of $\mathcal{F}_{\text{AUTH}}$, thereby making sure that these protocols can be analyzed per instance, independently of other instances. This modeling also significantly simplifies the analysis of protocols that obtain authenticated communication. Another modeling aspect is that $\mathcal{F}_{\text{AUTH}}$ generates an output for the receiver without requiring the receiver to provide any input. This means that the SID is determined exclusively by the sender, and there is no need for the sender and receiver to agree on the SID in advance.[18]

On realizing $\mathcal{F}_{\text{AUTH}}$. $\mathcal{F}_{\text{AUTH}}$ is used not only as a formalization of the authenticated communication model. It also serves as a way for specifying the security requirements from authentication protocols. (As discussed earlier, the validity of this dual use comes from the universal composition theorem.) We very briefly summarize some basic results regarding the realizability of $\mathcal{F}_{\text{AUTH}}$.

As a first step, we note that it is *impossible* to realize $\mathcal{F}_{\text{AUTH}}$ in the bare model, except by protocols that never generate any output. That is, say that a protocol is useless if no party ever generates output with non-negligible probability for any PPT environment and adversary. Then, we have:

[17]Previous formulations of of $\mathcal{F}_{\text{AUTH}}$ failed to let the adversary change the delivered message and recipient identity if the sender gets corrupted between sending and delivery. This results in an unrealistically strong security guarantee, that is not intuitively essential and is not provided by reasonable authentication protocols. This oversight was pointed out in several places, including [HMS03, AF04].

[18] We point out that this non-interactive formulation of $\mathcal{F}_{\text{AUTH}}$ makes crucial use of the fact that the underlying computational model from Section 3.1 allows for dynamic addressing and generation of ITIs. Indeed, allowing such simple and powerful formulation of $\mathcal{F}_{\text{AUTH}}$ and similar functionalities has been one of the main motivations for the present formulation of the underlying computational model.

Claim 17 ([c04]) *Any protocol that UC-realizes $\mathcal{F}_{\text{AUTH}}$ in the bare model is useless.*

Still, there are a number of ways to realize $\mathcal{F}_{\text{AUTH}}$ in algorithmic ways, given some other abstractions on the system. Following the same definitional approach, these abstractions are again formulated by way of ideal functionalities. One such abstraction is allowing the parties to communicate in an ideally authenticated way at some preliminary stage. Another abstraction postulates existence of a trusted "bulletin board", where parties can register public values (e.g., keys) that can be authentically obtained by other parties upon request. These abstractions come in a number of flavors, that significantly influence the feasibility and efficiency of realizing $\mathcal{F}_{\text{AUTH}}$. See more discussion in [c04].

6.4 Secure Communication

The abstraction of secure communication, often called secure message transmission, usually means that the communication is authenticated, and in addition the adversary has no access to the contents of the transmitted message. It is typically assumed that the adversary learns that a message was sent, plus some partial information on the message (such as, say, its length, or more generally some information on the domain from which the message is taken). In the present framework, having access to an ideal secure message transmission mechanism can be cast as having access to the "secure message transmission functionality", $\mathcal{F}_{\text{SMT}}$, presented in Figure 12. The behavior of $\mathcal{F}_{\text{SMT}}$ is similar to that of $\mathcal{F}_{\text{AUTH}}$ with the following exception. $\mathcal{F}_{\text{SMT}}$ is parameterized by a leakage function $l : \{0,1\}^* \to \{0,1\}^*$ that captures the allowed information leakage on the transmitted plaintext m. That is, the adversary only learns the leakable information $l(m)$ rather than the entire m. (In fact, $\mathcal{F}_{\text{AUTH}}$ can be regarded as the special case of $\mathcal{F}^{l}_{\text{SMT}}$ where l is the identity function.) In particular, $\mathcal{F}_{\text{SMT}}$ is standard corruption.

Functionality $\mathcal{F}^{l}_{\text{SMT}}$

$\mathcal{F}^{l}_{\text{SMT}}$ proceeds as follows, when parameterized by leakage function $l : \{0,1\}^* \to \{0,1\}^*$.

1. Upon receiving an input $(\texttt{Send}, S, R, sid, m)$ from ITI S, send $(\texttt{Sent}, S, R, sid, l(m))$ to the adversary, generate a private delayed output $(\texttt{Sent}, S, sid, m)$ to R and halt.
2. Upon receiving $(\texttt{Corrupt}, sid, P)$ from the adversary, where $P \in \{S, R\}$, disclose m to the adversary. Next, if the adversary provides a value m', and $P = S$, and no output has been yet written to R, then output $(\texttt{Sent}, S, sid, m')$ to R and halt.

Figure 12: The Secure Message Transmission functionality parameterized by leakage function l. The functionality is standard corruption.

Like $\mathcal{F}_{\text{AUTH}}$, $\mathcal{F}_{\text{SMT}}$ only deals with transmission of a single message. Secure transmission of multiple messages is obtained by using multiple instances of $\mathcal{F}_{\text{SMT}}$. In addition, like $\mathcal{F}_{\text{AUTH}}$, $\mathcal{F}_{\text{SMT}}$ allows the adversary to change the contents of the message and the identity of the recipient as long as the sender is corrupted *at the time of delivery,* even if the sender was uncorrupted at the point when it sent the message. In addition, following our convention regarding party corruption, when either the sender or the receiver are corrupted, $\mathcal{F}_{\text{SMT}}$ discloses the sender's input to the adversary.

Stronger variants. One common requirement from secure message transmission protocols is forward secrecy: the transmitted message should remain secret, even if the the sender and receiver of the message are corrupted *after* the protocol execution is completed. A natural way to capture forward secrecy in the present formulation is to modify the behavior upon corruption of either the sender or the receiver, so as to not disclose the plaintext message m to the adversary. Note that the rest of the code of $\mathcal{F}_{\text{SMT}}$ need not be changed at all.

Another common requirement is protection from traffic analysis. Recall that, whenever a party S sends a message to some R, $\mathcal{F}_{\text{SMT}}$ notifies the adversary that S sent a message to R. This reflects the common view that encryption does not hide the fact that a message was sent, namely there is no protection against traffic analysis. To capture security against traffic analysis, modify $\mathcal{F}_{\text{SMT}}$ so that the adversary does not learn that a message was sent, or alternatively employ a mechanism for hiding the identities of the sender or the receiver from the adversary.

On realizing $\mathcal{F}_{\text{SMT}}$. Protocols that UC-realize $\mathcal{F}_{\text{SMT}}$ can be constructed, based on public-key encryption schemes that are semantically secure against chosen plaintext attacks, by using each encryption key for encrypting only a single message, and authenticating the communication via $\mathcal{F}_{\text{AUTH}}$. That is, let $E = (gen, enc, dec)$ be an encryption scheme for domain D of plaintexts. (Here gen is the key generation algorithm, enc is the encryption algorithm, dec is the decryption algorithm, and correct decryption is guaranteed for any plaintext in D.) Then, consider the following protocol, denoted π_E. When invoked with input $(\texttt{Send}, sid, m)$ where $m \in D$ and $sid = (S, R, sid')$, π_E first sends an initialization message to R, namely it invokes an instance of $I_{\mathcal{F}_{\text{AUTH}}}$ with input $(\texttt{Send}, sid'', \texttt{init-smt})$, where $sid'' = (S, R, sid'1)$, and with PID S. Upon invocation with subroutine output $(\texttt{Sent}, sid'', \texttt{init-smt})$ and with identity (R, sid), π_E runs algorithm gen, gets the secret key sk and the public key pk, and sends (sid, pk) back to (sid, S), using $\mathcal{F}_{\text{AUTH}}$ in the same way. Next, (sid, S) computes $c = enc(pk, m)$, uses $\mathcal{F}_{\text{AUTH}}$ again to send c to (sid, R), and returns. Finally, upon receipt of (sid, c), π_E within R computes $m = dec(sk, c)$, and outputs $(\texttt{Sent}, sid, m)$.

It can be verified that the above protocol UC-realizes $\mathcal{F}_{\text{SMT}}$ as long as the underlying encryption scheme is semantically secure against chosen plaintext attacks. That is, given a domain D of plaintexts, let l_D be the "leakage function" that, given input x, returns $\perp$ if $x \in D$ and returns x otherwise. Then:

Claim 18 *If E is semantically secure for domain D as in* [GM84, G01] *then π_E UC realizes $\mathcal{F}_{\text{SMT}}^{l_D}$ in the presence of non-adaptive adversaries.*

Furthermore, if E is non-committing (as in [CFGN96]*) then π_E UC-realizes $\mathcal{F}_{\text{SMT}}^{l_D}$ with adaptive adversaries. This holds even if data erasures are not trusted and the adversary sees all the past internal states of the corrupted parties.*

Choosing new keys for each message to be transmitted is of course highly inefficient and does not capture common practice for achieving secure communication. As in the case of $\mathcal{F}_{\text{AUTH}}$, it is possible to realize multiple instances of $\mathcal{F}_{\text{SMT}}$ using a single instance of a more complex protocol, in a way that is considerably more efficient than running multiple independent instances of a protocol that realizes $\mathcal{F}_{\text{SMT}}$. One way of doing this is to use the same encryption scheme to encrypt all the messages sent to some party. Here the encryption scheme should have additional properties on top of being semantically secure. In [CKN03] it is shown that replayable chosen ciphertext security (RCCA) suffices for this purpose for the case of non-adaptive party corruptions. In the case of adaptive corruptions stronger properties and constructions are needed, see further discussion in [N02, CHK05].

6.5 Synchronous communication

A common and convenient abstraction of communication networks is that of *synchronous* communication. Roughly speaking, here the computation proceeds in *rounds,* where in each round each party receives all the messages that were sent to it in the previous round, and generates outgoing messages for the next round. There are of course many variants of the synchronous model. We concentrate here on modeling one basic variant, which provides the following guarantee:

Timely delivery. Each message sent by an uncorrupted party is guaranteed to arrive in the next communication round. In other words, no party will receive messages sent at round r before all uncorrupted parties had a chance to receive the messages sent at round $r-1$.

This guarantee implies in turn two other ones:

Guaranteed delivery. Each party is guaranteed to receive all messages that were sent to it by uncorrupted parties.

Authentic delivery. Each message sent by an uncorrupted party is guaranteed to arrive unmodified. Furthermore, the recipient knows the real sender identity of each message.

Typically, the order of activation of parties within a round is assumed to be under adversarial control, thus the messages sent by the corrupted parties may depend on the messages sent by the uncorrupted parties in the *same round.* This is often called the "rushing" model for synchronous networks.

Synchronous variants of the UC framework are presented in [N03, HM04a, KMTZ13]. Here we provide an alternative way of capturing synchronous communication within the UC framework: We show how synchronous communication can be captured by having access to an ideal functionality $\mathcal{F}_{\text{SYN}}$ that provides the above guarantees. We first describe $\mathcal{F}_{\text{SYN}}$, and then discuss and motivate some aspects of its design.[19] $\mathcal{F}_{\text{SYN}}$ is presented in Figure 13. It expects its SID to include a list $\mathcal{P}$ of parties among which synchronization is to be provided. At the first activation, $\mathcal{F}_{\text{SYN}}$ initializes a round number r to 1, and sends a delayed public output `init` to all parties in $\mathcal{P}$. (The purpose of this output is to notify the parties in $\mathcal{P}$ that a synchronous session with the given SID has started; in particular, it frees the calling protocol from the need to agree on the SID in advance.) Next, $\mathcal{F}_{\text{SYN}}$ responds to two types of inputs: Given input of the form $(\texttt{Send}, sid, M)$ from party $P \in \mathcal{P}$, $\mathcal{F}_{\text{SYN}}$ interprets M as a list of messages to be sent to other parties in $\mathcal{P}$. The list M is recorded together with the sender identity and the current round number, and is also forwarded to the adversary. Upon receiving a $(\texttt{Corrupt}, P)$ from the adversary, for some $P \in \mathcal{P}$, $\mathcal{F}_{\text{SYN}}$ marks P as corrupted.

Given an input $(\texttt{Receive}, sid, r')$ from a party $P \in \mathcal{P}$, where r' is a round number, $\mathcal{F}_{\text{SYN}}$ proceeds as follows. If r' is the current round number and the round is complete (in the sense that all uncorrupted parties have sent their messages for this round), then $\mathcal{F}_{\text{SYN}}$ completes this round, prepares the list of messages sent to each party at this round, and advances the round number.

[19]The formulation of $\mathcal{F}_{\text{SYN}}$ in earlier versions of this work was slightly different: It allowed the adversary to indefinitely delay the round advancement operation, consequently that formulation did not guarantee eventual delivery of messages. This is pointed out in [KMTZ13], where an alternative fix to the one used here is proposed.

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

$\mathcal{F}_{\text{SYN}}$ expects its SID to be of the form $sid = (\mathcal{P}, sid')$, where $\mathcal{P}$ is a list of party identities among which synchronization is to be provided. It proceeds as follows.

1. At the first activation, initialize a round counter $r \leftarrow 1$, and send a public delayed output `(Init,`sid`)` to all parties in $\mathcal{P}$.
2. Upon receiving input `(Send,`sid, M`)` from a party $P \in \mathcal{P}$, where $M = \{(m_i, R_i)\}$ is a set of pairs of messages m_i and recipient identities $R_i \in \mathcal{P}$, record (P, M, r) and output (sid, P, M, r) to the adversary.
3. Upon receiving input `(Receive,`sid, r'`)` from a party $P \in \mathcal{P}$, do:
   (a) If $r' = r$ (i.e., r' is the current round), and all uncorrupted parties in $\mathcal{P}$ have already sent their messages for round r, then:
      i. Interpret N as the list of messages sent by all parties in this round. That is, $N = \{(S_i, R_i, m_i)\}$ where each $S_i, R_i \in \mathcal{P}$, m is a message, and $S_i, R_i \in \mathcal{P}$. (S_i is taken as the sender of message m_i and R_i is the receiver. Note that either sender or receiver may be corrupted.)
      ii. Prepare for each party $P \in \mathcal{P}$ the list L_P^r of messages that were sent to it in round r.
      iii. Increment the round number: $r \leftarrow r + 1$.
      iv. Output `(Received,`sid, L_P^{r-1}`)` to P. (Let $L_P^0 = \perp$.)

   (b) If $r' < r$ then output `(Received,`$sid, L_P^{r'}$`)` to P.

   (c) Else (i.e., $r' > r$ or not all parties in $\mathcal{P}$ have sent their messages for round r), output `(Round Incomplete)` to P.

Figure 13: The synchronous communication functionality, $\mathcal{F}_{\text{SYN}}$.

Then it returns to the requesting party the messages sent to it in this round. If r' refers to an already completed round then the corresponding list of messages for that round is returned. If r' refers to an incomplete or future round then an error message is returned.

It is stressed that $\mathcal{F}_{\text{SYN}}$ does not deliver messages to a party until being explicitly requested by the party to obtain the messages. This mechanism facilitates capturing guaranteed delivery of messages within the present framework. In particular, a protocol that uses $\mathcal{F}_{\text{SYN}}$ can be guaranteed that as soon as all uncorrupted have sent messages for a round, the round will complete and all sent messages will be available to their recipients. Similarly, any protocol that realizes $\mathcal{F}_{\text{SYN}}$ must guarantee delivery of all messages sent by uncorrupted parties. See more discussion below.

To highlight the properties of $\mathcal{F}_{\text{SYN}}$, let us sketch a typical use of $\mathcal{F}_{\text{SYN}}$ by some protocol π. All parties of π use the same instance of $\mathcal{F}_{\text{SYN}}$. This instance can be invoked by any of the parties, or even the adversary. If the parties know in advance the SID of this instance then they can send messages to it right away. Otherwise, they can wait for the `init` message where the SID is specified. In any case, each party of π first initializes a round counter to 1, and inputs to $\mathcal{F}_{\text{SYN}}$ a list M of first-round messages to be sent to the other parties of π. In each subsequent activation, the party calls $\mathcal{F}_{\text{SYN}}$ with input `(Receive,`sid, r`)`, where sid is typically derived from the current SID of π and r is the current round number. In response, the party obtains the list of messages received in this round, performs its local processing, increments the local round number, and calls $\mathcal{F}_{\text{SYN}}$ again with input $(\texttt{Send}, sid, M)$ where M contains the outgoing messages for this round. If $\mathcal{F}_{\text{SYN}}$ returns $(\texttt{round Incomplete})$, this means that some parties have not completed this round yet. In this case, π does nothing (thus returning the control to the environment).

It can be seen that the message delivery pattern for such a protocol π is essentially the same as in a traditional synchronous network. Indeed, $\mathcal{F}_{\text{SYN}}$ requires that all parties actively participate in the computation in each round. That is, the round counter does not advance until all uncorrupted parties are activated at least once and send a (possibly empty) list of messages for that round. Furthermore, as soon as one uncorrupted party is able to obtain its incoming messages for some round, all uncorrupted parties are able to obtain their messages for that round.

The present formulation of $\mathcal{F}_{\text{SYN}}$ does not guarantee "fairness", in the sense that the adversary may obtain the messages sent by the uncorrupted parties for the last round while the uncorrupted parties may have not received these messages. (This might happen if the adversary fails to advance the round.) To guarantee fairness, modify $\mathcal{F}_{\text{SYN}}$ so that the adversary learns the messages sent by the uncorrupted parties in a round only after the round is complete.

Another point worth elaboration is that each instance of $\mathcal{F}_{\text{SYN}}$ guarantees synchronous message delivery only within the context of the messages sent using that instance. Delivery of messages sent in other ways (e.g., directly or via other instances of $\mathcal{F}_{\text{SYN}}$) may be arbitrarily faster or arbitrarily slower. This allows capturing, in addition to the traditional model of a completely synchronous network where everyone is synchronized, also more general settings such as synchronous execution of a protocol within a larger asynchronous environment, or several protocol executions where each execution is internally synchronous but the executions are mutually asynchronous.

Also note that, even when using $\mathcal{F}_{\text{SYN}}$, the inputs to the parties are received in an "asynchronous" way, i.e. it is not guaranteed that all inputs are received within the same round. Still, a protocol that uses $\mathcal{F}_{\text{SYN}}$ can deploy standard mechanisms for guaranteeing that the actual computation does not start until enough (or all) parties have inputs.

Finally, including the set $\mathcal{P}$ of participants in the SID captures cases where the identities of all participants are known to the initiator in advance. Alternative situations, where the set of participants is not known a priori can be captured by letting parties join in as the computation proceeds, and have $\mathcal{F}_{\text{SYN}}$ update the set $\mathcal{P}$ accordingly.

Potential relaxations. The reliability and authenticity guarantees provided within a single instance of $\mathcal{F}_{\text{SYN}}$ are quite strong: Once a round number advances, all the messages to be delivered to the parties at this round are fixed, and are guaranteed to be delivered upon request. It is possible to relax $\mathcal{F}_{\text{SYN}}$ by, say, allowing the adversary to stop delivering messages or to modify messages sent by corrupted parties even in "mid-round".

Another potential relaxation of $\mathcal{F}_{\text{SYN}}$ is to relax the "timeliness" guarantee. Specifically, it may only be guaranteed that messages are delivered within a given number, δ, of rounds from the time they are generated. The bound δ may be either known in advance or alternatively unknown and determined dynamically (e.g., specified by the adversary when the message is sent). The case of known delay δ corresponds to the "timing model" of [DNS98, G02, LPT04]. The case of unknown delay corresponds to the model of *non-blocking asynchronous communication model* where message are guaranteed to be delivered, but with unknown delay (see, e.g., [BCG93, CR93]).

On composing $\mathcal{F}_{\text{SYN}}$-hybrid protocols. Recall that each instance of $\mathcal{F}_{\text{SYN}}$ represents a single synchronous system, and different instances of $\mathcal{F}_{\text{SYN}}$ are mutually asynchronous, even when they run in the same system. This means that different protocol instances that wish to be mutually synchronized would need to use the same instance of $\mathcal{F}_{\text{SYN}}$. In particular, if we have a complex, multi-component protocol that assumes a globally synchronous network, then all the components of this protocol would need to use the same instance of $\mathcal{F}_{\text{SYN}}$.

A priori, this observation may seem to prevent the use of the universal composition operation for constructing such protocols, since this operation does not allow the composed protocols to have any joint subroutines. We note, however, that protocol instances that use the same instance of $\mathcal{F}_{\text{SYN}}$ can be composed using a tool called universal composition with joint state [CR03]. This tool allows analyzing each such protocol instance as if it is the only instance using $\mathcal{F}_{\text{SYN}}$, while still asserting security of the system where multiple protocol instances use the same instance of F_{SYN}. A key observation here is that one can straightforwardly realize multiple independent instances of $\mathcal{F}_{\text{SYN}}$ using a single instance of $\mathcal{F}_{\text{SYN}}$, whose set of participants is the union of the sets of participants of the realized instances.

6.6 Non-concurrent Security

One of the main features of the UC framework is that it guarantees security even when protocol instances are running concurrently in an adversarially controlled manner. Still, sometimes it may be useful to capture within the UC framework also security properties that are not necessarily preserved under concurrent composition, and are thus realizable by simpler protocols or with milder setup assumptions.

This section provides a way to express such “non-concurrent” security properties within the present framework. Specifically, we present an ideal functionality, $\mathcal{F}_{\text{NC}}$, that guarantees to the calling protocol instance that no other parties are activated as long as the current instance is running. other protocol instances are running concurrently in the system. Thus, modeling a protocol as an $\mathcal{F}_{\text{NC}}$-hybrid protocol is paramount to analyzing this protocol in a system where there are no protocol instances running concurrently.

Functionality $\mathcal{F}_{\text{NC}}$ is presented in Figure 14. It first expects to receive a code of an adversary, $\hat{\mathcal{A}}$. It then behaves as adversary $\hat{\mathcal{A}}$ would in the non-concurrent security model. That is, $\mathcal{F}_{\text{NC}}$ runs $\hat{\mathcal{A}}$ and follows its instructions with respect to receipt and delivery of messages between parties. As soon as $\hat{\mathcal{A}}$ terminates the execution, $\mathcal{F}_{\text{NC}}$ reports the current state of $\hat{\mathcal{A}}$ back to the external adversary, and halts. (Recall the convention from Section 6.2 regarding running arbitrary code.)

We remark that, in addition to analyzing “pure” non-concurrent security of protocols, $\mathcal{F}_{\text{NC}}$ can also be used to analyze systems where some of the components may be running concurrently with each other, where other components cannot. Here the “non-composable” components can be written as $\mathcal{F}_{\text{NC}}$-hybrid protocols.

Non-concurrent security of protocols that assume some idealized communication model (such as, say, authenticated or synchronous communication) can be captured by using $\mathcal{F}_{\text{NC}}$ in conjunction with the ideal functionality that captures the desired model. For instance, to capture non-concurrent security of synchronous protocols, let the protocol use $\mathcal{F}_{\text{SYN}}$ as described in Section 6.5, with the exception that $\mathcal{F}_{\text{SYN}}$ interacts with $\mathcal{F}_{\text{NC}}$ instead of interacting directly with the adversary. ($\mathcal{F}_{\text{NC}}$, in turn, interacts with the adversary as described above.)

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

1. When receiving message $(\texttt{Start}, sid, \hat{\mathcal{A}})$ from the adversary, where $\hat{\mathcal{A}}$ is the code of an ITM (representing an adversary), invoke $\hat{\mathcal{A}}$ and change state to `running`.
2. When receiving input $(\texttt{Send}, sid, m, Q)$ from party P, and if the state is `running`, activate $\hat{\mathcal{A}}$ with incoming message (m, Q) from party P. Then:
   (a) If $\hat{\mathcal{A}}$ instructs to deliver a message (m', P') to party Q' then output (sid, m', P') to Q'.
   (b) If $\hat{\mathcal{A}}$ halts without delivering a message to any party then send the current state of $\hat{\mathcal{A}}$ to the adversary and halt.

Figure 14: The non-concurrent communication functionality, $\mathcal{F}_{\text{NC}}$.

Equivalence with the definition of [C00]. Recall the security definition of [C00], that guarantees that security is preserved under non-concurrent composition of protocols. (See discussion in Section 1.1.) More specifically, recall that the notion of [C00] is essentially the same as UC security with two main exceptions: first, there the model of execution is synchronous, which is analogous to the present use of $\mathcal{F}_{\text{SYN}}$. Second, there the environment $\mathcal{E}$ and the adversary $\mathcal{A}$ are prohibited from sending inputs and outputs to each other from the moment where the first activation of a party of the protocol until the last activation of a party of the protocol.

Here we wish to concentrate on the second difference. We thus provide an alternative formulation within the current framework. Say that an environment is non-concurrent if it does not provide any input to the adversary other than the input provided to the adversary at its first activation, and furthermore does not provide any inputs to the protocol parties after receiving the first output of the adversary. Then:

Definition 19 *Let π and ϕ be PPT protocols. We say that π NC-emulates ϕ if for any PPT adversary $\mathcal{A}$ there exists a PPT adversary $\mathcal{S}$ such that for any non-concurrent PPT environment $\mathcal{E}$, we have $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}} \approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$.*

We argue that NC-emulation captures the essence of the notion of [C00]. An important feature of this notion of security is that it is easier to realize. In particular, known protocols (e.g., the protocol of [GMW87], see also [G04]) for realizing a general class of ideal functionality with any number of faults, assuming authenticated communication as the only set-up assumption, can be shown secure in this model. This stands in contrast to the impossibility results regarding the realizability of the same functionalities in the UC framework, even with authenticated communication.

We show that having access to $\mathcal{F}_{\text{NC}}$ is equivalent to running in the non-concurrent security model described above. More specifically, say that a protocol π is NC if all protocol messages are sent via $\mathcal{F}_{\text{NC}}$ (that is, there is no use of the communication links). Then, an NC protocol π UC-realizes some ideal functionality $\mathcal{F}$ if and only if π realizes $\mathcal{F}$ with non-concurrent security. More generally:

Proposition 20 *Let π and ϕ be protocols, where π is NC. Then π UC-emulates ϕ if and only if π NC-emulates ϕ.*

Notice that Proposition 20, together with the UC theorem, provide an alternative (albeit somewhat roundabout) formulation of the non-concurrent composition theorem of [C00].

Proof: Clearly if π UC-emulates ϕ then π NC-emulates ϕ. For the other direction, assume that π NC-emulates ϕ. This means that for any adversary $\mathcal{A}$ there exists an adversary (simulator) $\mathcal{S}$ such that $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}} \approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$ for all non-concurrent environments $\mathcal{E}$. To show that π UC-emulates ϕ, consider an interaction of π with the dummy adversary $\mathcal{D}$. Since π is an $\mathcal{F}_{\text{NC}}$-hybrid protocol, the environment $\mathcal{E}$ must now provide some code $\hat{\mathcal{A}}$ to $\mathcal{F}_{\text{NC}}$. From this point on, $\mathcal{E}$ sees no outgoing messages from any of the participants of π. Furthermore, it only receives one message from $\mathcal{F}_{\text{NC}}$, describing the "final" state of $\hat{\mathcal{A}}$. Once this message is given, no party generates any further output to $\mathcal{E}$.

Now, consider the simulator $\hat{\mathcal{S}}$ such that $\text{EXEC}_{\phi,\hat{\mathcal{S}},\mathcal{E}} \approx \text{EXEC}_{\pi,\hat{\mathcal{A}},\mathcal{E}}$ for any non-concurrent $\mathcal{E}$. (Existence of $\hat{\mathcal{S}}$ follows from the assumption that π NC-emulates ϕ.) We claim that $\text{EXEC}_{\phi,\hat{\mathcal{S}},\mathcal{E}} \approx \text{EXEC}_{\pi,\mathcal{D},\mathcal{E}}$ even when $\mathcal{E}$ is not necessarily non-concurrent. This is so since the interaction of $\mathcal{D}$ with π is limited in the same way as that of a non-concurrent environment. In other words, when interacting with $\mathcal{D}$ and an $\mathcal{F}_{\text{NC}}$-hybrid protocol π, any environment is effectively bound to be non-concurrent. $\square$

Acknowledgments

Much of the motivation for undertaking this project, and many of the ideas that appear here, come from studying secure key-exchange protocols together with Hugo Krawczyk. I thank him for this long, fruitful, and enjoyable collaboration. I am also grateful to Oded Goldreich who, as usual, gave me both essential moral support and invaluable technical and presentational advice.

Many thanks also to the many people with whom I have interacted over the years on definitions of security and secure composition. A very partial list includes Martin Abadi, Michael Backes, Mihir Bellare, Ivan Damgaard, Marc Fischlin, Shafi Goldwasser, Rosario Gennaro, Shai Halevi, Dennis Hofheinz, Yuval Ishai, Ralf Küsters, Eyal Kushilevitz, Yehuda Lindell, Phil MacKenzie, Tal Malkin, Cathy Meadows, Silvio Micali, Daniele Micciancio, Moni Naor, Rafi Ostrovsky, Rafael Pass, Birgit Pfitzmann, Tal Rabin, Charlie Rackoff, Phil Rogaway, Victor Shoup, Paul Syverson and Michael Waidner. In particular, it was Daniele who proposed to keep the basic framework minimal and model subsequent abstractions as ideal functionalities within the basic model.

References

[AG97] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *4th ACM Conference on Computer and Communications Security,* 1997, pp.36-47. Fuller version available at http://www.research.digital.com/SRC/ abadi.

[AR00] M. Abadi and P. Rogaway. Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). *J. Cryptology* 15(2): 103-127 (2002). Preliminary version at *International Conference on Theoretical Computer Science IFIP TCS 2000,* LNCS, 2000. On-line version at http://pa.bell-labs.com/ abadi/.

[AF04] M. Abe and S. Fehr. Adaptively Secure Feldman VSS and Applications to Universally-Composable Threshold Cryptography. *Crypto '04,* 2004.

[A04] J. Almansa. The Full Abstraction of the UC Framework. BRICS, Technical Report RS-04-15 University of Aarhus, Denmark, August 2004. Also available at eprint.iacr.org/2005/019.

[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.

[BPW07] M. Backes, B. Pfitzmann, and M. Waidner. The reactive simulatability (RSIM) framework for asynchronous systems. Inf. Comput. 205(12): 1685-1720 (2007)

[B01] B. Barak. How to go Beyond the Black-Box Simulation Barrier. In *42nd FOCS*, pp. 106–115, 2001.

[B$^+$11] B. Barak, R. Canetti, Y. Lindell, R. Pass and T. Rabin. Secure Computation Without Authentication. J. Cryptology 24(4): 720-760 (2011)

[BCNP04] B. Barak, R. Canetti, J. B. Nielsen, R. Pass. Universally Composable Protocols with Relaxed Set-Up Assumptions. *36th FOCS,* pp. 186–195. 2004.

[BGGL04] B. Barak, O. Goldreich, S. Goldwasser and Y. Lindell. Resettably-Sound Zero-Knowledge and its Applications. *42nd FOCS,* pp. 116-125, 2001.

[BLR04] B. Barak, Y. Lindell and T. Rabin. Protocol Initialization for the Framework of Universal Composability. Eprint archive. eprint.iacr.org/2004/006.

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

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

[B96] D. Beaver. Adaptive Zero-Knowledge and Computational Equivocation. *28th Symposium on Theory of Computing (STOC),* ACM, 1996.

[B97] D. Beaver. Plug and play encryption. *CRYPTO 97*, 1997.

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

[BCK98] M. Bellare, R. Canetti and H. Krawczyk. A modular approach to the design and analysis of authentication and key-exchange protocols. *30th Symposium on Theory of Computing (STOC),* ACM, 1998.

[BDPR98] M. Bellare, A. Desai, D. Pointcheval and P. Rogaway. Relations among notions of security for public-key encryption schemes. *CRYPTO '98*, 1998, pp. 26-40.

[BR93] M. Bellare and P. Rogaway. Entity authentication and key distribution. *CRYPTO'93*, LNCS. 773, pp. 232-249, 1994. (Full version from the authors or from `http://www-cse.ucsd.edu/users/mihir`.)

[BCG93] M. Ben-Or, R. Canetti and O. Goldreich. Asynchronous Secure Computations. *25th Symposium on Theory of Computing (STOC),* ACM, 1993, pp. 52-61.

[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.

[BM04] M. Ben-Or, D. Mayers. General Security Definition and Composability for Quantum & Classical Protocols. arXiv archive, http://arxiv.org/abs/quant-ph/0409062.

[BS97] E. Biham, A. Shamir. Differential Fault Analysis of Secret Key Cryptosystems. *CRYPTO '97,* pp. 513–525. 1997.

[BCH12] N. Bitansky, R. Canetti, S. Halevi. Leakage-Tolerant Interactive Protocols. TCC 2012: 266-284

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

[BDL97] D. Boneh, R. A. DeMillo, R. J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). *Eurocrypt '97,* pp. 37–51. 1997.

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

[BAN90] M. Burrows, M. Abadi and R. Needham. A logic for authentication. DEC Systems Research Center Technical Report 39, February 1990. Earlier versions in *the Second Conference on Theoretical Aspects of Reasoning about Knowledge*, 1988, and *the Twelfth ACM Symposium on Operating Systems Principles,* 1989.

[C95] R. Canetti. Studies in Secure Multi-party Computation and Applications.*Ph.D. Thesis,* Weizmann Institute, Israel, 1995.

[C98] R. Canetti. Security and composition of multi-party cryptographic protocols. ftp://theory.lcs.mit.edu/pub/tcryptol/98-18.ps, 1998.

[C00] R. Canetti. Security and composition of multi-party cryptographic protocols. *Journal of Cryptology,* Vol. 13, No. 1, winter 2000.

[C01] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. Version of October 2001. Available at http://eccc.uni-trier.de/eccc-reports/2001/TR01-016/revision01.ps.

[C04] R. Canetti. Universally Composable Signature, Certification, and Authentication. *17th Computer Security Foundations Workshop (CSFW),* 2004. Long version at eprint.iacr.org/2003/239.

[C05] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. Version of December 2005. Available at http://eccc.uni-trier.de/eccc-reports/2001/TR01-016.

[c06] Ran Canetti. Security and composition of cryptographic protocols: A tutorial. SIGACT News, Vol. 37, Nos. 3 & 4, 2006. Available also at the Cryptology ePrint Archive, Report 2006/465.

[c07] R. Canetti. Obtaining Universally Composable Security: Towards the Bare Bones of Trust. ASIACRYPT 2007, pp. 88-112.

[c08] R. Canetti. Composable Formal Security Analysis: Juggling Soundness, Simplicity and Efficiency. ICALP (2) 2008: 1-13

[c13] Ran Canetti. Security and composition of cryptographic protocols. Chapter in *Secure Multiparty Computation*, ed. Prabhakaran and Sahai. IOS Press, 2013.

[c+11] R. Canetti, S. Chari, S. Halevi, B. Pfitzmann, A. Roy, M. Steiner, W. Venema. Composable Security Analysis of OS Services. ACNS 2011: 431-448

[c+05] R. Canetti, L. Cheung, D. Kaynar, M. Liskov, N. Lynch, O. Pereira, and R. Segala. Using Probabilistic I/O Automata to Analyze an Oblivious Transfer Protocol. MIT Technical Report MIT-LCS-TR-1001, August 2005.

[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.

[cg96] R. Canetti and R. Gennaro. Incoercible multi-party computation. *37th Symp. on Foundations of Computer Science (FOCS)*, IEEE, 1996.

[chk05] R. Canetti, S. Halevi and J, Katz. Adaptively Secure Non-Interactive Public-Key Encryption. *2nd theory of Cryptology Conference (TCC)*, 2005.

[ch11] R. Canetti and J. Herzog. Universally Composable Symbolic Analysis of Cryptographic Protocols (The case of encryption-based mutual authentication and key exchange). J. Cryptology 24(1): 83-147 (2011)

[ck01] R. Canetti and H. Krawczyk. Analysis of key exchange protocols and their use for building secure channels. Eurocrypt '01, 2001. Extended version at http://eprint.iacr.org/2001/040.

[ckn03] R. Canetti, H. Krawczyk, and J. Nielsen. Relaxing Chosen Ciphertext Security of Encryption Schemes. *Crypto '03*, 2003. Extended version at the eprint archive, eprint.iacr.org/2003/174.

[cr93] R. Canetti and T. Rabin. Optimal Asynchronous Byzantine Agreement. *25th STOC*, 1993, pp. 42-51.

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

[cv12] R. Canetti, M. Vald. Universally Composable Security with Local Adversaries. SCN 2012: 281-301. See also IACR Cryptology ePrint Archive 2012: 117.

[CJRR99] S. Chari, C. S. Jutla, J. R. Rao, P. Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. *CRYPTO '99,* pp. 398–412. 1999.

[D05] A. Datta, Security Analysis of Network Protocols: Compositional Reasoning and Complexity-theoretic Foundations. PhD Thesis, Computer Science Department, Stanford University, September 2005.

[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.

[DIO98] G. Di Crescenzo, Y. Ishai and R. Ostrovsky. Non-interactive and non-malleable commitment. *30th STOC,* 1998, pp. 141-150.

[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),* ACM, 1991.

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

[DLMS04] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004.

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

[EGL85] S. Even, O. Goldreich and A. Lempel, A randomized protocol for signing contracts, *CACM,* vol. 28, No. 6, 1985, pp. 637-647.

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

[FF00] M. Fischlin and R. Fischlin, Efficient non-malleable commitment schemes, *CRYPTO '00, LNCS 1880*, 2000, pp. 413-428.

[GM00] J. Garay and P. MacKenzie, Concurrent Oblivious Transfer, *41st FOCS*, 2000.

[GLMMR04] R. Gennaro, A. Lysyanskaya, T. Malkin, S. Micali and T. Rabin. Tamper Proof Security: Theoretical Foundations for Security Against Hardware Tampering. *Theory of Cryptography Conference (TCC),* LNCS 2951. 2004.

[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.* Cambridge Press, Vol 1 (2001).

[G02] O. Goldreich. Concurrent Zero-Knowledge With Timing, Revisited. *34th STOC,* 2002.

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

[GK88] O. Goldreich and A. Kahan. How to Construct Constant-Round Zero-Knowledge Proof Systems for NP. *Jour. of Cryptology*, Vol. 9, No. 2, pp. 167–189, 1996.

[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),* ACM, 1987, pp. 218-229.

[GO94] O. Goldreich and Y. Oren. Definitions and properties of Zero-Knowledge proof systems. *Journal of Cryptology*, Vol. 7, No. 1, 1994, pp. 1–32. Preliminary version by Y. Oren in *28th Symp. on Foundations of Computer Science (FOCS),* IEEE, 1987.

[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.

[G11] V. Goyal. Positive Results for Concurrently Secure Computation in the Plain Model. FOCS 2012: 41-50

[HM00] M. Hirt and U. Maurer. Complete characterization of adversaries tolerable in secure multiparty computation. *Journal of 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.

[HMS03] D. Hofheinz and J. Müler-Quade and R. Steinwandt. Initiator-Resilient Universally Composable Key Exchange. *ESORICS,* 2003. Extended version at the eprint archive, eprint.iacr.org/2003/063.

[HM04a] D. Hofheinz and J. Müller-Quade. A Synchronous Model for Multi-Party Computation and the Incompleteness of Oblivious Transfer. Eprint archive, http:/eprint.iacr.org/2004/016, 2004.

[HMU09] D. Hofheinz, J. Müller-Quade and D. Unruh. Polynomial Runtime and Composability. IACR Cryptology ePrint Archive (IACR) 2009:23 (2009)

[HS11] D. Hofheinz, V. Shoup. GNUC: A New Universal Composability Framework. IACR Cryptology ePrint Archive 2011: 303 (2011)

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

[KMTZ13] J. Katz, U. Maurer, B. Tackmann, V. Zikas. Universally Composable Synchronous Computation. *Theory of Cryptology Conference (TCC)* 2013: 477-498

[KMM94] R. Kemmerer, C. Meadows and J. Millen. Three systems for cryptographic protocol analysis. *J. Cryptology,* 7(2):79-130, 1994.

[K96] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. *CRYPTO '96,* pp. 104–113. 1996.

[K06] R. Küsters. Simulation-Based Security with Inexhaustible Interactive Turing Machines. CSFW 2006, pp. 309-320.

[KT13] R. Ksters, M. Tuengerthal. The IITM Model: a Simple and Expressive Model for Universal Composability. IACR Cryptology ePrint Archive 2013: 25 (2013)

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

[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.

[LMMS99] P. Lincoln, J. Mitchell, M. Mitchell, A. Scedrov. Probabilistic Polynomial-time equivalence and security analysis. *Formal Methods Workshop,* 1999. Available at ftp://theory.stanford.edu/pub/jcm/papers/fm-99.ps.

[Ly96] N. Lynch. *Distributed Algorithms.* Morgan Kaufman, San Francisco, 1996.

[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.

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

[MR11] U. Maurer, R. Renner. Abstract Cryptography. *Innovations in Computer Science* 2011: 1-21

[MPR06] S. Micali, R. Pass, A. Rosen. Input-Indistinguishable Computation. FOCS 2006, pp. 367-378.

[MR04] S. Micali and L. Reyzin. Physically Observable Cryptography. *1st Theory of Cryptography Conference (TCC)*, LNCS 2951, 2004.

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

[MW04] D. Micciancio and B. Warinschi. Soundness of formal encryption in the presence of active adversaries. In the *1st Theory of Cryptography Conference (TCC)*, LNCS 2951, pp. 133–151. 2004.

[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. Schedrov. A Linguistic Characterization of Bounded Oracle Computation and Probabilistic Polynomial Time. *39th FOCS*, 1998, pp. 725-734.

[NY90] M. Naor and M. Yung. Public key cryptosystems provably secure against chosen ciphertext attacks". *22nd STOC,* 427-437, 1990.

[N02] J. B. Nielsen. Separating Random Oracle Proofs from Complexity Theoretic Proofs: The Non-committing Encryption Case. *CRYPTO,* pp. 111–126. 2002.

[N03] J. B. Nielsen. On Protocol Security in the Cryptographic Model. PhD thesis, Aarhus University, 2003.

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

[P06] R. Pass. A Precise Computational Approach to Knowledge. PhD Thesis, MIT. 2006.

[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.

[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*, 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.

[R81] M. Rabin. How to exchange secrets by oblivious transfer. Tech. Memo TR-81, Aiken Computation Laboratory, Harvard U., 1981.

[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.

[R06] Alon Rosen. *Concurrent Zero-Knowledge.* Series on Information Security and Cryptography. Springer-Verlag, 2006.

[R$^+$00] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe and B. Roscoe. *The Modeling and Analysis of Security Protocols: the CSP Approach.* Addison-Wesley, 2000.

[SL95] R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing,* Vol. 2. No. 2, pp 250-273, 1995.

[Sh99] V. Shoup. On Formal Models for Secure Key Exchange. manuscript, 1999. Available at: http://www.shoup.org.

[Si05] M. Sipser. *Introduction to the Theory of Computation.* Second edition, Course Technology, 2005.

[S+06] C. Sprenger, M. Backes, D. A. Basin, B. Pfitzmann, and M. Waidner. Cryptographically sound theorem proving. In *CSFW*, pages 153–166. IEEE Computer Society, July 2006.

[Y82] A. Yao. Theory and applications of trapdoor functions. In *Proc. 23rd Annual Symp. on Foundations of Computer Science (FOCS),* pages 80–91. IEEE, 1982.

[Y82] A. Yao. Protocols for Secure Computation. In *Proc. 23rd Annual Symp. on Foundations of Computer Science (FOCS),* pages 160–164. IEEE, 1982.

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

A Related work

This section surveys some related work. For brevity, we concentrate on works lead to the present framework or directly affect it, thus omitting many works that use this framework, study it and extend it. The surveys in [C01, C06, C07, C08, C13] cover some of these works.

Two works that essentially "laid out the field" of general security definitions for cryptographic protocols are the work of Yao [Y82], which stated 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 "trusted-party paradigm", namely the approach of defining security via comparison with an ideal process involving a trusted party (albeit in a very informal way).

Another work that greatly influenced the formation of notions of security is the work of Dolev, Dwork and Naor [DDN00]. This work points out some important security concerns that arise when cryptographic protocols run concurrently within a larger system. In particular, making sure that the concerns pointed out in [DDN00] are addressed is central to the present framework.

The first rigorous definitional framework is due to Goldwasser and Levin [GL90], and 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 present one. Beaver's framework was the first to directly formalize the idea of comparing a run of a protocol to an ideal process. (However, the [MR91, B91] formalisms only address security in restricted settings; in particular, they do not deal with computational issues.) [GL90, MR91, B91] are surveyed in [C00] in more detail.

The frameworks of [GL90, MR91, B91] concentrate on *synchronous* communication. Also, although in [GMW87] the trusted-party paradigm was put forth for reactive functionalities, the three frameworks concentrate on the task of *secure function evaluation.* An extension to *asynchronous* communication networks with eventual message delivery 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, [C00] defines a general composition operation, called *modular composition,* which is a non-concurrent version of universal composition. That is, only a single protocol instance can be active at each point in time. (See more details in Section 6.6.) In addition, security of protocols in that framework is shown to be preserved under modular composition. That work also contains sketches on how to strengthen the definition to support concurrent composition. The UC framework implements these sketches in a direct way. A closely related formulation appears in [G04].

The framework of Hirt and Maurer [HM00] provides a rigorous treatment of reactive functionalities. Dodis and Micali [DM00] build on the definition of Micali and Rogaway [MR91] for unconditionally secure function evaluation, where ideally private communication channels are assumed. In that setting, they prove that their notion of security is preserved under a general concurrent composition operation similar to universal composition. They also formulate an additional composition operation (called *synchronous composition*) that provides stronger security guarantees, and show that their definition is closed under that composition operation in cases where the scheduling of the various instances of the protocols can be controlled. However, their definition applies only to settings where the communication is ideally private. It is not clear how to extend this definitional approach to settings 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. (This work is based on the sketches in [PW94]). 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 to asynchronous networks appears in [PW01].

At high level, the notion of security in [PSW00, PW00, PW01], called reactive simulatability, is similar to the one here. In particular, the role of their "honest user" can be roughly mapped to the role of the environment as defined here. However, there are several differences. They use a finite-state machine model of computation that builds on the I/O automata model of [Ly96], as opposed to the ITM-based model used in this work. On the one hand, their model provides a richer set of methods for scheduling events in an execution. On the other hand, they postulate a closed system where the number of participants is constant and fixed in advance. Furthermore, the number of *protocol instances* run by the parties is constant and fixed in advance, thus it is impossible to argue about the security of systems where the number of protocol instances may be a non-constant function of the security parameter (even if this number is known in advance). That model also automatically provides each pair of parties with a dedicated and labeled communication channel; thus it does not facilitate arguing about situations where dedicated channels are not available a-priori. Other technical differences include the notion of polynomial time computation and the generation of outputs.

Backes, Pfitzmann and Waidner [BPW04] extend the framework of [PW01] to deal with the case where the number of parties and protocol instances depends on the security parameter. In that framework, they prove that reactive simulatability is preserved under universal composition. The [BPW07] formulation returns to the original approach where the number of entities and protocol instances is fixed irrespective of the security parameter. Nielsen [N03], Hofheinz and Müller-Quade [HM04a], and Katz et. al. [KMTZ13] formulate synchronous variants of the UC framework.

Lincoln, Mitchell, Mitchell and Scedrov [LMMS98, LMMS99] develop a process calculus, based on the π-calculus of Milner [M89, M99], that incorporates random choices and computational limitations on adversaries. (In [MMS98] it is demonstrated how to express probabilistic polynomial time within such a process calculus.) In that setting, their definitional approach has a number of similarities to the simulation-based approach taken here: They define a computational variant of *observational equivalence*, and say that a real-life process is secure if it is observationally equivalent to an "ideal process" where the desired functionality is guaranteed. This is indeed similar to requiring that no environment can tell whether it is interacting with the ideal process or with the protocol execution. However, their ideal process must vary with the protocol to be analyzed, and they do not seem to have an equivalent of the notion of an "ideal functionality" which is associated only with the task and is independent of the analyzed protocol. This makes it harder to formalize the security requirements of a given task.

Mateus, Mitchell and Scedrov [MMS03] and Datta, Küsters, Mitchell, and Ranamanathan [DKMR05] (see also [D05]) extend the [LMMS98, LMMS99] framework to express simulatability as defined here, cast in a polytime probabilistic process calculus, and demonstrate that the universal composition theorem holds in their framework. They also rigorously compare certain aspects of the present framework (as defined in [C01]) and reactive simulatability (as defined in [BPW07]). Tight correspondence between the [MMS03] notion of security and the one defined here is demonstrated in Almansa [A04].

Canetti et al. [C+05] extend the probabilistic I/O automata of Lynch, Segala and Vaandrager [SL95, LSV03] to a framework that allows formulating security of cryptographic protocols along the lines of the present UC framework. This involves developing a special mechanism, called the *task schedule,* for curbing the power of non-deterministic scheduling; it also requires modeling resource-bounded computations. The result is a framework that represents the concurrent nature of distributed systems in a direct way, that allows for analyzing partially-specified protocols (such as, say, standards), that allows some scheduling choices to be determined non-deterministically during run-time, and at the same time still allows for meaningful UC-style security specifications.

Küsters [K06, KT13] formulates an ITM-based model of computation that allows for defining UC-style notions of security. The model contains new constructs that facilitate both flexible addressing of messages and a flexible notion of resource-bounded computation in a distributed environment. This work also adapts abstract notations from the process calculus literature to his ITM-based model, allowing for succinct and clear presentation of composition theorems and proofs. The present version of this work uses ideas from [K06].

Hofheinz, Müller-Quade and Unruh [HMU09] provides an alternative definition of polynomial time ITMs that works well within the present framework. Hofheinz and Shoup [HS11] point to a number of flaws in previous versions of this work and formulate a variant of the UC framework that avoids these flaws. Their framework (called GNUC) differs from the present one in two main ways: First, their notion of polynomial time is close to that of [HMU09]. Second, they mandate a more rigid subroutine structure for protocols, as well as a specific format for SIDs that represents the said subroutine structure. While indeed simplifying the argumentation on a natural class of protocols, the GNUC framework does not allow representing and arguing about other natural classes (see e.g. Footnote 18).

Finally, we note that the present framework has evolved considerably over the years. Previous versions of this work, and some comparisons between them, appear in [C01].

B The main changes from the previous versions

We list the changes made to this document since its first public posting in 2001. The changes are listed in chronological order.

Changes in the 1/6/2005 version from the 10/2001 version. The changes from this version (see [C01]) are detailed throughout the text. Here we provide a brief, high-level outline of the main changes:

Non-technical changes:

1. A more complete survey of related work (prior, concurrent, and subsequent) is added in Section A.
2. The section on realizing general ideal functionalities (Section 7 in the October '01 version) is not included. It will be completed to contain a full proof and published separately.
3. Motivational discussion is added throughout.

Technical changes:

1. Extended and more detailed definitions for a "system of interacting ITMs" are added, handling dynamic generation and addressing of ITM instances (i.e., ITIs) in a multi-party, multi-protocol, multi-instance environment.
2. New notions of probabilistic polynomial time ITMs and systems are used. The new notions provide greater expressibility and generality. They also allow proving equivalence of several natural variants of the basic notion of security.
3. The model for protocol execution is restated in terms of a system of interacting ITMs. In particular, the order of activations is simplified.
4. The ideal process and the hybrid model are simplified and made more expressive. In particular, they are presented as special types of protocols within the general model of protocol execution.
5. The composition theorem is stated and proven in more general terms, considering the general case of replacing one subroutine protocol with another.

6. Various models of computation, including authenticated channels, secure channels, and synchronous communication, are captured as hybrid protocols that use appropriate ideal functionalities within the basic model of computation. This avoids defining extensions to the model to handle these cases, and in particular avoids the need to re-prove the UC theorem for these extended models. Various corruption models are also captured as special protocol instructions within the same model of execution.

Additional changes in the 1/28/2005 version. The main change in this version is in the definition of PPT ITMs and systems. Instead of globally bounding the running time of the system by a fixed polynomial, we provide a more "locally enforceable" characterization of PPT ITMs that guarantees that an execution of the system completes in polynomial time overall. See discussion in Section 3.2.1. These changes required updating the notion of black-box simulation and the proof of Claim 10, both in Section 4.4. Thanks to Dennis Hofheinz for pointing out the shortcomings of the previous formulation of PPT ITMs.

Additional changes in the 12/2005 version. Discussions are updated and added. In addition, the main technical changes are:

1. The notions of security with respect to the dummy adversary and with black-box simulation were modified. Indeed, the proof of equivalence of security with dummy adversary and standard UC security was flawed, and a change in the definition was needed. See more details in Section 4.4. (Thanks to Ralf Küsters for pointing out the flaw.)

   A related change is that in prior versions the definition of security allowed the adversaries to be A-PPT rather than PPT. Here all ITMs, including the adversaries, should be PPT (namely, their running time is bounded by a function only of the input length and the security parameter.) This simplifies the definition somewhat.

2. A more concrete treatment of UC security with respect to actual numeric parameters is added, parameterizing two main quantities: the emulation slack, namely the probability of distinguishing between the emulated and the emulating executions, and the simulation overhead, namely the difference between running times of the adversaries in the compared executions.

3. Formulations of ideal functionalities for most of the primitives considered in the 10/2001 version, and some additional primitives, are added. The formulations of practically all these primitives are updated in a number of ways, see more details in the relevant sections. Still, the spirit of the formulations remains unchanged. Some of the ideal functionalities included in the 1/2005 versions were also updated.

Main changes in the 7/2013 version from the 12/2005 version Section 7 (UC formulations of some primitives) was deleted. The hope is to upload an up-to-date version of section 7 shortly. Also much of the survey of subsequent work in the Introduction was deleted. The survey of prior and concurrent work was moved to the Appendix. In addition, discussions were updated throughout.

We list the main technical changes. The list is rather laconic. Motivation and more details appear within.

1. The basic model of computation (Section 3.1) was simplified in a number of ways, most prominently in the semantics of the "external write" operation and in the notion of polynomial time machines.

2. The model of protocol execution (Section 4) was simplified and updated in two ways:

   (a) Party corruption (with its many variants) was taken out of the basic model. It it modeled as protocol operations.

   (b) The model now allows the main parties of the protocol to explicitly specify the target of their outputs. This is accompanied by a mechanism for the environment to "impersonate" identities of ITIs that give inputs to and obtain outputs from the protocol instance.

3. The notion of UC emulation (Section 4) was relaxed to quantify only over environments that are "balanced" in the lengths of inputs it gives to the adversary vs the protocol parties.

4. The notion of *subroutine respecting* protocols (Section 5) was updated to require also that the subroutine structure of the protocol (namely the IDs and programs of all subroutines) be known to the adversary.

5. The behavior of parties upon corruption was modified, to allow for a more expressive treatment of the party corruption operation.

6. The formulation of the synchronous communication functionality, $\mathcal{F}_{\text{SYN}}$, was fixed. The previous formulation was buggy: contrary to what was claimed, $\mathcal{F}_{\text{SYN}}$ did not guarantee "liveness", or delivery of messages. The present formulation does.