Universally Composable Security: A New Paradigm for Cryptographic Protocols*

Ran Canetti†

December 30, 2018

Abstract

We present a general framework for describing 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 simpler 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, which 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.

*An extended abstract of this work appears in the proceedings of the 42nd Foundations of Computer Science conference, 2001. This is an updated version. While the overall spirit and the structure of the definitions and results in this paper has remained the same, many important details have changed. We point out and motivate the main differences from the 2013 version in Appendix B. Earlier versions of this work appeared in 2001, 2005 and 2013, under the same title, and in December 2000 under the title "A unified framework for analyzing security of protocols". These earlier versions can be found at the ECCC archive, TR 01-16 (http://eccc.uni-trier.de/eccc-reports/2001/TR01-016); however they are not needed for understanding this work and have only historic significance.

†Boston University and Tel Aviv University. Member of CPIIS. Supported by the NSF MACS project, NSF Grant 1801564, and ISF Grant 1523/14.

Contents

1 Introduction **1**
- 1.1 The definitional approach . . . 3
- 1.2 Universal Composition . . . 5
- 1.3 Making the framework useful: Simplicity and Expressibility . . . 7
- 1.4 Overview of the rest of this paper . . . 9

2 Warmup: A simplified model **10**
- 2.1 Machines and Protocols . . . 10
- 2.2 Defining security of protocols . . . 13
  - 2.2.1 Protocol execution . . . 13
  - 2.2.2 Ideal functionalities and ideal protocols . . . 16
  - 2.2.3 Protocol emulation and realizing an ideal functionality . . . 17
- 2.3 Universal Composition . . . 18
- 2.4 Modeling tasks and protocols of interest . . . 20

3 The model of computation **20**
- 3.1 The basic model . . . 21
  - 3.1.1 Interactive Turing Machines (ITMs) . . . 21
  - 3.1.2 Executing ITMs . . . 23
- 3.2 Polynomial time ITMs & parameterized systems . . . 27
- 3.3 Discussion . . . 30
  - 3.3.1 Motivating the use of ITMs . . . 30
  - 3.3.2 On the identity mechanism . . . 31
  - 3.3.3 On the external-write mechanism and the control function . . . 34
  - 3.3.4 On capturing resource-bounded computations . . . 37

4 Defining security of protocols **39**
- 4.1 The model of protocol execution . . . 39
- 4.2 Protocol emulation . . . 42
- 4.3 Realizing ideal functionalities . . . 44
- 4.4 Alternative formulations of UC emulation . . . 46
  - 4.4.1 Emulation with respect to the dummy adversary . . . 46
  - 4.4.2 Emulation with respect to black-box simulation . . . 50
  - 4.4.3 Letting the simulator depend on the environment . . . 51
- 4.5 Some Variants of UC emulation . . . 52

5 Universal composition **54**
- 5.1 The universal composition operation and theorem . . . 54
- 5.2 Proof of Theorem 18 . . . 57
  - 5.2.1 Proof outline . . . 57
  - 5.2.2 A detailed proof . . . 60

5.3 Discussion and extensions 65

6 UC formulations of some computational models **67**
6.1 Writing conventions for protocols and ideal functionalities 68
6.1.1 A basic convention: Bodies and Shells 68
6.1.2 Some corruption models 69
6.1.3 Writing conventions for ideal functionalities 73
6.2 Some communication models 75
6.2.1 Authenticated Communication 75
6.2.2 Secure Communication 78
6.2.3 Synchronous communication 80
6.3 Non-concurrent Security 83

A Related work **96**

B The main differences from the 2013 version **100**

1 Introduction

Rigorously demonstrating that a protocol "does its job securely" is an essential component of cryptographic protocol design. Doing so 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. First, the model should be rich enough to represent all realistic adversarial behaviors, as well as the plethora of prevalent design techniques for distributed algorithms and networks. Next, 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.

Initial definitions of security for specific cryptographic tasks (e.g. [GM84, GMRa89]) considered simplified models which capture 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 these 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] 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] were shown not to be closed under composition, and new notions and constructions were needed [GO94, GK89, F91, DNS98, RK99, BGGL04]; Key Exchange, where the original notions allow protocols that fail to provide secure communication, even when combined with secure symmetric encryption and authentication protocols [BR93, BCK98, Sh99, CK01, GL01]; Oblivious Transfer [R81, EGL85, GM00] and secure multiparty funtion evaluation [GL90, B91, MR91, PW00, C00, DM00, G04] 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, GL01], non-malleable commitments [DDN00], concurrent zero-knowledge [DNS98], and general concurrently secure protocols [P04, BS05] 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 whose scope is inherently limited to 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 refer only to 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 obtained by formulating the definitions of security in a way that guarantees the preservation of security 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 their security properties 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 multiple 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 formal operation 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 shall 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.

A fair number of frameworks for defining security of protocols in a way that guarantees security-preserving composition have been proposed since the first publication of this work [C01]. Many of these works are influenced by this work, and many of them influenced later versions of this work (including this one). Here let us mention only [N03, PS04, W05, K06, BPW07, CDPW07, MR11, KT13, MT13, HS11, CCL15]. Specific influences are mentioned when relevant.

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 instantiating the general approach within a framework that is both expressive and usable. Related work, including both

[1] We use similar names for two very different objects: A notion of security and a composition operation. We do so since we believe that the two are intimately tied together. We note that elsewhere (e.g. [G04, Section 7.7.2]) the notions are decoupled, with security being called *environmental security* and composition being articulated as concurrent.

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. 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 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.

Prior formalisms. Several formalizations of this general definitional approach exist, including the definitional works mentioned above. These formalisms provide 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.

A protocol is a computer program (or several programs), intended to be executed by a number of communicating computational entities, or parties. In accordance, the model of protocol execution consists of a set of interacting computing elements, each running the protocol on its own local input and making its own random choices. (The same model is considered also in [G04, Section 7.5.1].) Formally, these elements are modeled as interactive Turing machines (ITMs).[2] An additional computing element, called the adversary, represents an entity that controls some subset of the parties and in addition has some control over the communication network. The ITMs running the protocol and adversary interact (i.e., exchange messages) in some specified manner, until each entity 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. That is, the adversary determines the inputs of the corrupted parties and learns their outputs.

Protocol π securely evaluates a function f if for any adversary $\mathcal{A}$ (that interacts with the protocol and controls some of the parties) there exists an ideal-process adversary $\mathcal{S}$, that controls the same parties as $\mathcal{A}$, such that the following holds: For any input values given to the parties, the global

[2]While following tradition, the specific choice of Turing machines as the underlying computational model is somewhat arbitrary. Any other imperative model that provides a concrete way to measure the complexity of realistic computations would be adequate, the RAM and PRAM models being quintessential examples.

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, the only pertinent information learned by any set of corrupted parties is their own inputs and outputs from the computation, in the sense that the output of any adversary that controls the corrupted parties is indistinguishable from an output generated (by a simulator) given only the relevant inputs and outputs. However, this definition provides only limited guarantees regarding the security of systems that involve the execution of two or more protocols. Specifically, general secure composition is guaranteed only as long as no two protocol instances run concurrently. Indeed, there are natural protocols that meet the [C00] definition but are insecure when as few as *two* instances are active at the same time and subject to a coordinated attack against them. 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, C13].

The UC framework. The UC framework preserves the overall structure of the above 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 formulations, 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 represents *whatever is external to the current protocol execution.* This includes other protocol executions and their adversaries, human users, etc.) The environment interacts with the protocol execution twice: First, before the execution starts, the environment hands arbitrary inputs of its choosing to the parties and to the adversary. Next, once the execution terminates, the environment 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, π, or with the ideal process for f. Now, say that 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. (The above description corresponds to the static-corruptions variant of the [C00] definition, where the set of corrupted parties is fixed in advance. In the case of adaptive corruption, the [C00] definition allows some additional interaction between the environment and the protocol at party-corruption events.)

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 at any point throughout the course of the protocol execution. 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 models of computation 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 system 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 between the protocol and its environment 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 is constructed in the proof of security: In order to make sure that the environment $\mathcal{E}$ cannot tell the difference 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" $\mathcal{E}$, and thus it cannot "take back" information that it previously sent $\mathcal{E}$. Indeed, it is this pattern of intensive interaction between $\mathcal{E}$ and $\mathcal{A}$ that allows proving that security is preserved under universal composition. (Indeed, this restriction on $\mathcal{S}$ is incompatible with the "black-box simulation with rewinding" technique which underlies much of traditional cryptographic protocol analysis; alternative techniques are thus called for.)

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 extending the "trusted party" in the ideal process for secure function evaluation to constitute 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 their inputs, even when new inputs are chosen adaptively based on previous outputs and the protocol communication. 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 (see e.g. [G04, Section 7.7.1.3]). Other differences from [C00], such as capturing different communication models and the ability to dynamically generate programs, are discussed in later sections.

1.2 Universal Composition

Consider the following method for composing two protocols into a single composite protocol. (It might be helpful to think about this composition operation as a generalization of the "subroutine substitution" operation for sequential algorithms to the case of distributed protocols.) Let ρ be a protocol where the parties make "subroutine calls" to some ideal functionality $\mathcal{F}$. That is, in addition to the standard set of instructions, ρ may include instructions to provide $\mathcal{F}$ with some input value; furthermore, ρ contains instructions for the case of obtaining outputs from $\mathcal{F}$. (Recall that an instance of ρ typically involves multiple ITMs; this in particular means that a single instance of $\mathcal{F}$ operates as a subroutine of multiple ITMs.)

Furthermore, we allow ρ to call *multiple instances* of $\mathcal{F}$, and even have multiple instances of $\mathcal{F}$ run concurrently. We provide ρ with a general mechanism for "naming" the instances of $\mathcal{F}$ in order to distinguish them from one another, but leave it up to ρ to decide on the actual names. The instances of $\mathcal{F}$ run independently of each other, without any additional coordination.

Now, let π be a protocol that UC-realizes $\mathcal{F}$, according to the above definition. Construct the

composed protocol $\rho^{\mathcal{F}\to\pi}$ by starting with protocol ρ, and replacing each call to an instance of $\mathcal{F}$ with a call to an instance of π. Specifically, an input given to an instance of $\mathcal{F}$ is now given to an ITM in the corresponding instance of π, and outputs of an ITM in an instance of π are treated by ρ as outputs obtained from the corresponding instance of $\mathcal{F}$.[3]

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

It is stressed that the universal composition theorem considers an adversary and an environment that potentially run a "coordinated attack" that uses information collected from the execution of the "high-level part" of $\rho^{\mathcal{F}\to\pi}$, as well as from the the various instances of π. The theorem guarantees that any such coordinated attack can be translated to an "attack against ρ (with calls to $\mathcal{F}$)", plus a set of attacks, where each such attack operates separately against a single instance of $\mathcal{F}$.

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 (as in e.g. [DNS98, CK02, CF01, CLOS02]). A more detailed taxonomy and discussion appears in [C06, C13].

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 $\mathcal{G}$ continues to UC-realize $\mathcal{G}$ even when composed with other protocols using any of the composition operations considered in the literature.

Universal composition also allows formulating new ways to put together protocols (or, equivalently, new ways to decompose complex systems into individual protocols). A salient example here is the case where two or more protocol instances have some "joint state", or more generally "joint subroutines". Said otherwise, this is the case where a single instance of some protocol γ serves as a subroutine of two or more different instances of protocol π. Furthermore, γ may also serve as a subroutine of the "calling protocol" ρ or of other protocols in the system. Still, we would like to be able to analyze each instance separately in vitro, and deduce the security of the overall system — in very much the same way as the traditional case where the instances of π do not share any state. Situations where this type of (de-)composition becomes useful include the commonplace settings where multiple secure communication sessions use the same long-term authentication modules, or where multiple protocol instances use the same shared reference string or same randomly chosen hash function

[3] In prior versions of this work, as well as elsewhere in the literature, the original protocol is denoted $\rho^{\mathcal{F}}$ and the composed protocol is denoted ρ^{π}. However that notation suggests a model operation that "attaches" some fixed protocol (either $\mathcal{F}$ or π) to *any* subroutine call made by ρ. In contrast, we wish to consider situations where ρ makes multiple subroutine calls, to different protocols, and where only calls to $\mathcal{F}$ are replaced by calls to π, whereas other calls remain unaffected.

Universal composition in such situations was initially investigated in [CR03], for the case of protocols that share subroutines only with other instances of themselves, and in [CDPW07] for the case of protocols that share subroutines with arbitrary, untrusted protocols. (The latter case is often dubbed as the case of global subroutines.) In particular, the above-mentioned situations are addressed in [CDPW07, CJS14, CSV16]. While the formalism developed in this work in fact suffices for rigorous formalization of joint-state universal composition and universal composition with global subroutines, we leave the actual treatment of these cases out of scope for this paper.

Implications of the composition theorem: Modularity and stronger security. 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 secure realization 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 implication is indeed very useful. In addition, this work articulates another implication of the composition theorem, which is arguably stronger: We use it to address the concern described at the beginning of the Introduction, namely to argue that UC-realization is a robust notion of security that holds in arbitrary protocol environments. 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, unpredictable and adversarial environments such as modern communication networks.

1.3 Making the framework useful: Simplicity and Expressibility

In order to turn the general definitional approach described in Sections 1.1 and 1.2 into an actual definition of security that can be used to analyze protocols of interest, one has to first pinpoint a formal model that allows representing protocols written for distributed systems. The model should also allow formulating ideal functionalities, and make sure that there is a clear and natural interpretation of these ideal functionalities as specifications (security and otherwise) for tasks of interest. In particular, the model should allow rigorous representation of executions of a protocol alongside an adversary and an environment, as well as the ideal processes for an ideal functionality alongside a simulator and an environment, as outlined in Sections 1.1 and 1.2. In addition, the model should allow representing the universal composition operation and asserting the associated theorem.

Devising such a model involves multiple "design choices" on various levels. These choices affect the level of detail and formality of the resulting definition of security, its expressive power (in terms of ability to capture different situations, tasks, and real-life protocols), its faithfulness (in terms of ability to capture all realistic attacks), as well as the complexity of describing the definition and working with it. The goal, of course, is to devise a model which is simple and intuitive, while

being as expressive and faithful to reality as possible; however, simplicity, expressive power, and faithfulness are often at odds.

This work presents two such models, providing two points of tradeoff among these desiderata. The first model is a somewhat simplistic one whose goal is to highlight the salient points in the definitional approach with minimal formalism, at the price of restricting the class of protocols and tasks that can be naturally modeled. The second model is significantly more expressive and general, at the price of some additional formalism. The rest of this section highlights several of the definitional choices taken. More elaborate discussions of definitional choices appear throughout this work.

One aspect that is common to both models is the need to rigorously capture the notion of "subroutine machines" and "subroutine protocol instances". (Here we use the term "machine" to denote a computational entity without getting into specific details.) This in particular involves making rigorous the concept of providing an input to a subroutine ITM, obtaining output from a subroutine ITM, and an ITM that is a subroutine of multiple ITMs.

Next we mention three observations that are used in both models and significantly simplify the treatment. The first observation is that there is no need to devise separate formalisms for representing protocols and representing ideal functionalities. Similarly, there is no need to formalize the ideal process separately from the process of protocol execution. Instead we allow representing ideal functionalities as special cases of protocols. We then let the ideal process be the same as the process of executing a protocol alongside an adversary and an environment, where the protocol is one that represents an ideal functionality. One caveat here is that the model will need to formalize the ability machines to interact directly with the adversary, to enable representing the capabilities of ideal functionalities.

The second simplifying observation is that there is no need to directly formalize within the basic model of computation an array of different "corruption models", namely different ways by which parties turn adversarial and deviate from the original protocol. (Traditional models here are "honest-but-curious" and "Byzantine" corruption models, where the set of corrupted parties is chosen either statically or adaptively, as well as a variety of other attacks such as side-channel leakage, coercion, transient break-ins, and others.) In fact, the basic model need not formally model corruption at all. Instead, the different corruption models can be captured by having the adversary deliver special corruption messages to parties, and considering protocols whose behavior changes appropriately upon receipt of such messages.

The third observation is that there is no need to directly formalize an array of communication and synchronization models as separate models of computation. (Traditionally, such models would include authenticated communication, private communication, synchronous communication, broadcast, etc.) Instead, communication can be captured by way of special "channel ITMs" that are subroutines of two or more other ITMs, where the latter ITMs represent the communicating entities. Different communication models are then captured via different programs for the channel ITMs, where these programs may include communication with the adversary. The meaningfulness of this approach is guaranteed by the UC theorem: Indeed, if we compose a protocol ρ that was designed in a model where some communication abstraction is captured via an ideal functionality $\mathcal{F}$, with a protocol π that UC-realizes $\mathcal{F}$, where π operates in a communication model that is captured via some ideal functionality $\mathcal{G}$, then the composed protocol $\rho^{\mathcal{F}\to\pi}$ is a protocol that uses only calls to $\mathcal{G}$, and at the same time UC-emulates ρ. This approach also provides flexibility in expressing multiple variants of common communication models.

The tradeoff between simplicity and expressibility comes to play in the modeling of identities and programs for computational entities, the addressing of messages, and the way protocol instances evolve over time. In the first, simplistic variant of the model, we postulate a static system with a fixed set of computational entities, with fixed identities and programs, and where each such entity can communicate only with an a-priori fixed set of entities with known identities. Similarly the sets of computational entities that constitute "protocol instances" are fixed ahead of time.

While this modeling is indeed simpler to present and argue about, it does not facilitate capturing many realistic settings. (Natural examples include servers that need to interact with clients whose identities are not known in advance and may be dynamically chosen, peer-to-peer protocols whose membership is open and dynamic, or where participants are instructed to execute code that was dynamically generated by others, or even just systems which allow for an adversarially-controlled number of independent instances of a simple protocol where the number of instances is not known to the protocol.) In contrast, the more general model provides buit-in mechanisms for representing dynamically generated computatioal processes, with dynamically generated programs, identities, and communication patterns.

Furthermore, the model provides a natural way to delineate an "instance of a protocol" (namely, a set of processes that we wish to analyze as a single instance of some protocol) even when an a-priori unbounded number of processes join the instance dynamically throughout the computation, without global coordination.

We note that the second modeling approach stands apart from existing models of distributed computing. Indeed, existing models typically impose more static restrictions on the system; this results in reduced ability to express protocols, scenarios and threats that are prevalent in modern networks.

A final choice relates to the level of formalism: While we strive to pin down the details of the model as much as possible, we do not provide much in terms of syntax and a "programming language" for expressing protocols and their execution. Instead, we rely on the basic minimal syntax of Turing machines. We leave it to future work to develop more formal and abstract domain-specific programming languages that will facilitate mechanized representation and analysis of protocols and ideal functionalities within this framework.

Finally, we note that the models of computation presented here are different than the one in the first public version of this work [C00A, version of 2000]. Indeed the model has evolved considerably over time, with the main steps being archived at [C00A, later versions]. In addition, a number of works in the literature provide different tradeoffs between simplicity and expressibility, e.g. [N03, W05, W16, Sh99, CCL15]. See the Appendix for more details on these works as well as on previous versions of the present work.

1.4 Overview of the rest of this paper

The simplified model of computation is presented in Section 2. To further increase readability, the presentation in that section is somewhat informal.

Section 3 presents a general model for representing distributed systems. While this model is, of course, designed to be a basis for formulating definitions of security and asserting composability, we view it as a contribution of independent interest. Indeed, this model is quite different from other models in the literature for representing distributed computations: First, as mentioned above, it captures fully dynamic and evolving distributed systems. Second, it accounts for computational

costs and providing the necessary mechanisms and "hooks" for more abstract concepts such as protocol instances, subroutines, emulation of one protocol by another, and composition of protocols.

Section 4 presents the basic model of protocol execution in the presence of an adversary and environment, as well as the general notions of protocol emulation and realizing ideal functionalities. This section also presents some variants of the basic definition of protocol emulation (called UC-emulation) and asserts relationships among them.

Section 5 presents the universal composition operation, and then states and proves the universal composition theorem.

Section 6 exemplifies the use of the framework. It first proposes some writing conventions for protocols and ideal functionalities that capture a number of common settings and "benchmarks" in cryptographic protocols. This includes several types of models for corruption of parties and adversarial control, as well as delay and timeliness guarantees from protocols. Next it presents a handful of ideal functionalities that capture some salient communication and corruption models.

Finally, the Appendix reviews related work and its relationship with the present one. It also briefly reviews previous versions of this work.

2 Warmup: A simplified model

This section presents a simplified version of the definition of security and the composition theorem. We obtain simplicity by somewhat restricting the model of computation, thereby somewhat restricting the expressive power of the resulting definition and the applicability of the composition theorem. On the positive side, the restriction allows us to highlight the main ideas of the definition and composition theorem with minimal formalism. To further improve readability, we also allow ourselves to be somewhat informal in this section.

Section 2.1 defines the main object of interest, namely protocols. Section 2.2 presents the definition of security. Section 2.3 presents the universal composition theorem for this model, and sketches its proof.

2.1 Machines and Protocols

As discussed earlier, 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.) In contrast with an algorithm written for standard one-shot sequential execution, a protocol consist of several separate programs, where each program is intended to run independently of (and potentially concurrently with) all others. In particular each program obtains its own inputs and generates its own outputs. During their execution, the programs interact by transmitting information to each other. In the rest of this subsection we provide some basic formalism that will allow defining and arguing about protocols. We start with formalism regarding the individual programs (which we call machines), and then move on to formalizing protocols as collections of machines with certain properties.

The reader should keep in mind that the formalism presented below will differ in a number of ways from the traditional cryptographic view of a protocol as a "flat" collection of programs (machines) where each program represents the overall actions of a "party", namely an actual real-world entity. First, to enable the use of composition we allow a "party" to have multiple

machines, where each machine may have several subroutine machines. Second, we use the same construct (machines) to represent both computational processes that physically execute on an actual processor, as well as abstract processes that do not actually execute on any physical machine but rather represent ideal functionalities as per the modeling sketched in the introduction. Third, we will allow the machines that represent abstract entities to be the subroutines of multiple "caller machines", where some of these caller machines represent different principals.

As for the formalism itself: We almost completely refrain from defining the syntax of machines. (In Section 3 we do propose some rudimentary syntax based on interactive Turing machine [GMRa89, G01], but any other reasonable model or syntax will do.) Still, we define the following constructs.

First, we let each machine have a special value called the identity of the machine. The identity remains unchanged throughout the computation. Second, we require that incoming information to a machine be labeled either as input (representing inputs from a "calling machine"), or as subroutine-output (representing output from subroutines of the machine). We also allow a third form of incoming information, called backdoor, which will be used to model information coming from the adversary (to be defined in Section 2.2), or disclosed to it.

Third, we provide each machine μ with a communication set, which lists the set of identities of machines that μ can send information to, including the type of information: input, subroutine-output, or backdoor. That is, the communication set C of μ consists of a sequence of pairs (ID, L), where ID is an identity string and $L \in \{\text{input, subroutine-output, backdoor}\}$. (As we'll see shortly, we'll restrict attention to collections of machines where a machine μ can provide input to machine μ' if and only if μ' can provide subroutine output to μ, and μ can provide backdoor information to μ' if and only if μ' can provide backdoor information to μ. Thus the set of machines that can send inputs, subroutine outputs or backdoor information to a given machine μ can be inferred from its own communication set.)

In all, a machine is a triple $\mu = (ID, C, \tilde{\mu})$ where ID is the identity, C is the communication set, and $\tilde{\mu}$ is the program of the machine. See Figure 2.1 for a graphical depiction of a machine.

Protocols. The next step is to define multi-party protocols, namely collections of programs (machines) designed to be executed separately from each other, while exchanging information. As mentioned above, our formalism incorporates in a single protocol machines that are subroutines of other machines in the protocol, and allows both the case where a machine has multiple subroutine, and the case where a machine is a subroutine of multiple others machines. In particular, a "party" may be associated with multiple machines, some of which are subroutines of others; furthermore, some machines in the protocol may not be associated with any particular "party".

deviates in a number of ways from the traditional view where there is a one-to-one cor

First, we define subroutine machines: Machine μ is a subroutine of machine μ' if the communication set of μ' allows it to provide input to μ (and the communication set of μ allows it to provide subroutine output to μ'). In this case μ' is a caller of μ.

Next we distinguish two types of machines in a protocol: The machines whose communication set allows them to provide subroutine output to machines outside the protocol are called the main machines of the protocol. and those who cannot. The remaining machines are called the internal machines of the protocol.

We note that, in the present formalism, the intuitive notion of an m-party protocol naturally translates to a protocol that has m main machines, and potentially other subsidiary machines.

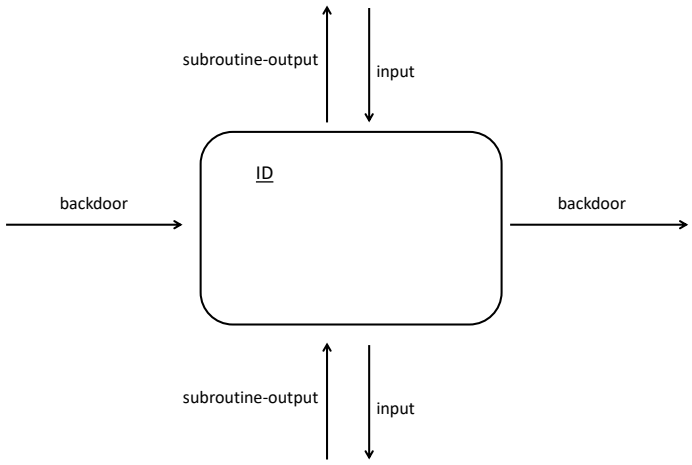

Figure 1: The basic computing unit (machine). Each machine has an identity that remains unchanged throughout the computation. Information from the outside (i.e., from other machines) is labeled as either *input* or or *subroutine-output*. Information sent to other machines is labeled similarly. In addition, a machine can receive information from and send information to the adversary; this information is labeled as *backdoor*. For graphical clarity, in future drawings we draw inputs as lines coming from above, subroutine outputs as lines coming from below, and backdoor communication as lines coming from either side. The communication set of the machine is not depicted.

When mapping the above traditional notion of an m-party protocol to the present formalism, a "party" now corresponds to a number of machines, one of which is a main machine, and the rest are internal machines. Furthermore, the formalism will guarantee that the internal machines associated with a party will be subsidiaries of the corresponding main machine. (A machine is a subsidiary of machine μ if it is a subroutine of μ or of other subsidiaries of μ.)

In addition, we will include in a protocol machines that represent abstract constructs (such as ideal functionalities). These machines will be internal machines; they will also not be necessarily associated with any specific party.

More formally, we let a protocol π be a set of machines $\pi = (\mu_1, ..., \mu_n)$ such that the identities of the machines are unique, and such that the communication sets of the machines satisfy the following requirements:

- If π contains a machine $\mu = (ID, C, \tilde{\mu})$ whose communication set C contains an entry (ID',input), then π also contains a machine $\mu' = (ID', C', \tilde{\mu}')$ such that C' contains an entry (ID, subroutine-output). That is, if μ can send input to μ' then μ' too is in π, and in addition μ' can send subroutine-output to μ.

- If π contains a machine $\mu = (ID, C, \tilde{\mu})$ whose communication set includes an entry (ID',subroutine-output), and in addition π contains a machine μ' with identity ID', then the communication set of μ' contains an entry (ID,input). (That is, if μ can send subroutine-output to μ' and μ' is a machine in π, then μ' can send input to μ.) If π does not contain a machine with identity ID' then we say that μ is a main party of π.

- π contains no "subroutine cycles". That is, say that a machine μ is a descendant of machine μ' if μ is a subroutine of μ' or of another descendant of μ'. Then π contains no machine that is a descendant of itself.

Modeling network communication. So far, the model does not have a mechanism for the natural operation of communication between machines in a protocol, nor does it make any provision for the "backdoor channels". Both of these mechanisms will be discussed in Section 2.2.

Polynomial-time machines and protocols. We require that the overall number of steps taken by a machine is polynomial in a global security parameter. Jumping ahead, this will mean that an execution of any protocol can be simulated on a standard Turing machine in time polynomial in the security parameter and the number of machines.

2.2 Defining security of protocols

As discussed in the Introduction, the 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 single trusted party that obtains all the inputs. We substantiate this approach as follows.

First, we will formulate the process of executing a protocol. Next, we will define the ideal process for carrying out a distributed computational task; this will be done using the same process of protocol execution where the protocol is of a special type, called an "ideal protocol" for the task. Finally we will define what it means for a protocol π to *securely realize* the ideal protocol. This will formalize the requirement that, from the point of view of the rest of the system, running π has essentially the same effect as running the ideal protocol.

As discussed in Section 1.3, the model of execution does not explicitly represent network communication between machines. nor does it directly represent corrupted machines Still, we informally discuss how these crucial mechanism can be modeled on top of the present formalism.

2.2.1 Protocol execution

We first present the formal model of protocol execution, and then discuss and motivate a number of aspects of the formalism.

In order to execute a protocol π we add two machines to the model: The environment $\mathcal{E}$ and the adversary $\mathcal{A}$. (The definition of security will later quantify over all polynomial time $\mathcal{E}$ and $\mathcal{A}$.) The environment has identity 0, and its communication set allows it to provide inputs to $\mathcal{A}$ and all the main parties of π. The adversary has identity 1, and its communication set allows it to provide backdoor information to all machines. Next, the communication sets of the machines of π are augmented to include the ability to provide backdoor information to $\mathcal{A}$.

An execution of π with adversary $\mathcal{A}$ and environment $\mathcal{E}$, on initial input z, starts by running $\mathcal{E}$ on input z. From this point on, the machines take turns in executing according to the following order: Whenever machine μ provides information to machine μ' (namely μ either provides input to μ', or provides subroutine-output to μ', or sends a backdoor message to μ'), the execution of μ is suspended. If the transmission of the message is consistent with the communication set of μ, then the execution of μ' begins (or resumes), with the incoming information from μ (including μ's identity), until the point where μ' provides information to another machine. If μ pauses without

sending information, or if the sending of information is inconsistent with the communication set of μ, then the execution of $\mathcal{E}$ resumes.

The communication between $\mathcal{E}$ and the main parties of π is subject to the following two exceptions from the general rules of execution: First, If a main party μ of π instructs to provide subroutine-output to a machine μ' which is not in π, then the value is given to $\mathcal{E}$, along with the identity of μ' and μ's own identity. Second, when $\mathcal{E}$ provides input to a main party μ of π, then $\mathcal{E}$ can choose the sender-identity that is seen by μ, as long as the chosen identity is one that μ can provide subroutine output to. (Figuratively, $\mathcal{E}$ assumes all the identities of the machines that the main parties of π can provide subroutine outputs to.)

The execution ends when $\mathcal{E}$ halts. We assume that $\mathcal{E}$ has a special binary *output* variable. The output of the execution is the contents of that variable when $\mathcal{E}$ halts.

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 an execution of π with environment $\mathcal{E}$ and adversary $\mathcal{A}$, on input z, as described above. Let $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$ denote the ensemble $\{\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(z)\}_{z\in\{0,1\}^*}$.

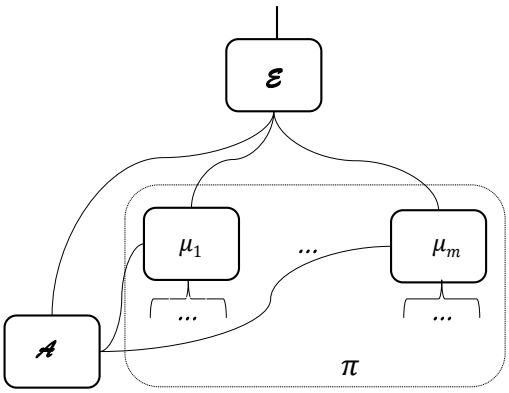

Figure 2: The model of execution of protocol π. The environment $\mathcal{E}$ writes the inputs and reads the subroutine-outputs of the main parties of π. In addition, $\mathcal{E}$ and $\mathcal{A}$ interact freely. The main parties of π may have subroutines, to which $\mathcal{E}$ has no direct access.

Discussion. We discuss some aspects of the model. See additional discussion in Section 3.3.

On the order of activations and "true concurrency". One might wonder whether the above simple and sequential order of activations adequately represents concurrent systems. Indeed, the model appears at first to stand in contrast with the physical nature of distributed systems, where computations take place in multiple physically separate places at the same time.

Traditionally, true concurrency is captured via a "mental experiment" where events happen with no particular order except for certain causality constraints, and are then ordered non-deterministically to form an interleaved execution. We go one step further: We consider a simpler mental experiment where events happen one by one, and the order of events is completely determined by the actions of the participants with no additional non-determinism. Still, we argue that this simple order does provide a sound basis for the study of concurrent systems in general, and their security properties

in particular. Indeed, by setting individual activations of ITMs to represent the appropriate granularity of "atomic sequential operations", it is possible to represent distributed computations at any desired level of abstraction or detail - from the granularity of physical interrupts in actual CPUs, to virtualized processes in a multi-process virtual system, to abstract processes that do not necessarily correspond to specific physical computations. Furthermore, the modeling of the scheduling mechanism as an *algorithmic* and potentially adversarial entity is crucial for capturing security guarantees that hold only probabilistically and only against computationally bounded adversaries.

Modeling network and inter-process communication. As discussed above, the present model only allows machines to communicate via inputs and outputs, and does not provide direct representation of a "communication link". Instead, communication links can be captured via dedicated machines that exhibit the properties of the channels modeled.

For sake of illustration, let us describe the machine, $\mu_{[1,2]}$, that represents authenticated asynchronous communication, without guaranteed delivery and with possible replay, from machine μ_1 to machine μ_2. $\mu_{[1,2]}$ has two types of triggers: (a) Upon receiving input m from μ_1, $\mu_{[1,2]}$ records m and send m to $\mathcal{A}$ as backdoor information. (b) Upon receiving m as backdoor, $\mu_{[1,2]}$ checks that m is recorded and if so outputs m to μ_2. Other types of communication are modeled analogously. (Note that this simplified modeling of authenticated communication is slightly different than that of Section 6.2.1.)

We argue that leaving the modeling of the communication links outside the basic model of computation helps keep the model simple, and at the same time general and expressive. In particular, different types of communication channels can be captured via different programs for the channel machine; in particular the backdoor tape of the channel machine will be useful to encode various forms of adversarial control and leakage.

Modeling party corruption. The above model of protocol execution does not contain explicit constructs for representing adversarial behavior of parties. As mentioned in Section 1.3, this too is done for sake of simplifying the basic model and definition of security, while preserving generality and expressibility. We sketch how adversarial behavior is represented within this model.

Adversarial behavior of parties is captured by way of having the adversary "assume control" over a set of machines, by handing a special corruption instruction to the target machines as backdoor information. Protocols should then contain a set of formal instructions for following the directives in these messages. The specific set of instructions can be considered to be part of a more specialized corruption model, and should represent the relevant expected behavior upon corruption.

This mechanism allows representing many types of corruption, such as outright malicious behavior, honest-but-curious behavior, side-channel leakage, transient failures, coercion. See further discussion in Section 6.1.2. For instance, Byzantine corruptions can be modeled by having the corruption instruction include a new program, and having the machine execute the new program in all future activations, instead of the original program. To model static Byzantine corruptions, the switch to the new program will happen only if the corruption instruction arrives at the very first activation. To model adaptive Byzantine corruptions, the instructions are activated even if the corruption message arrives later on in the computation; in addition the machine will now send its entire current state to the adversary upon corruption.

Jumping ahead, we note that in order to make the definition of protocol emulation meaningful

even in the presence of machine corruption, we will need a mechanism that allows the environment to obtain information regarding which machines are currently under control of the adversary. One way to implement such a mechanism is to have each protocol include a special "record-keeping" machine that is notified whenever a machine that is part of the protocol is corrupted; the environment will then query this record-keeping machine to learn the current list of corrupted machines within the relevant protocol.

2.2.2 Ideal functionalities and ideal protocols

Security of protocols is defined by way of comparing the protocol execution to an *ideal process* for carrying out the task at hand. As sketched in Section 1.3, we avoid formulating the ideal process from scratch by casting the ideal process for a task as a special protocol, and then use the above model of protocol execution. We call this special protocol the ideal protocol for the task. A key ingredient in the ideal protocol is the ideal functionality that captures the desired functionality, or the specification, of the task by way of a set of instructions for a "trusted party".

More specifically, an ideal protocol for capturing a task for m participants consists of a machine $\mathcal{F}$, representing the ideal functionality, plus m special machines called dummy parties. Upon receiving input, each dummy party forwards this input to $\mathcal{F}$, along with the identity of the caller machine. Upon receiving an output value from $\mathcal{F}$, a dummy party forwards the value to its specified destination machine, along with its own identity. The dummy parties ignore backdoor information (coming from $\mathcal{A}$). This means that $\mathcal{A}$ can meaningfully communicate only directly with $\mathcal{F}$. A graphical depiction of the ideal protocol for $\mathcal{F}$, denoted $\text{IDEAL}_{\mathcal{F}}$, appears in Figure 3. (Using the terminology of Section 2.1, the main machines of $\text{IDEAL}_{\mathcal{F}}$ are the dummy parties. $\mathcal{F}$ is an internal machine of $\text{IDEAL}_{\mathcal{F}}$.)

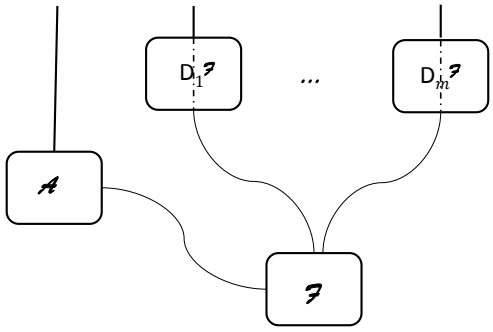

Figure 3: The ideal protocol $\text{IDEAL}_{\mathcal{F}}$ for an ideal functionality $\mathcal{F}$. The main parties of $\text{IDEAL}_{\mathcal{F}}$, denoted $D_1^{\mathcal{F}}, ..., D_m^{\mathcal{F}}$, are "dummy parties": they only relay inputs to $\mathcal{F}$, and relay outputs of $\mathcal{F}$ to the calling machines. The adversary $\mathcal{A}$ communicates only with $\mathcal{F}$ (by providing and receiving backdoor information).

The backdoor communication between $\mathcal{F}$ and $\mathcal{A}$ provides a flexible and expressive way to relax and "fine-tune" the security guarantees provided by $\mathcal{F}$. Indeed, as discussed in the Introduction, a natural way to represent tasks that allow some "leakage of information" is by having $\mathcal{F}$ explicitly

provide this information to $\mathcal{A}$. Similarly, tasks that allow some amount of "adversarial influence" on the outputs of the participants can be represented by letting $\mathcal{F}$ take into account information received from $\mathcal{A}$. Also, note that $\mathcal{F}$ can naturally capture the security requirements from reactive tasks. Indeed, it can maintain local state and each of its outputs may depend on all the inputs received and all random choices so far.

2.2.3 Protocol emulation and realizing an ideal functionality

We define what it means for protocol π to realize an ideal functionality $\mathcal{F}$. For this purpose we first formulate a more general definition for when protocol π "emulates" another protocol ϕ. The idea is to directly capture the requirement that no environment should be able to tell whether it is interacting with π and some adversary of choice, or with ϕ and some other adversary:

Definition 1 (protocol emulation, simplified model) *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, for any input, the probability that $\mathcal{E}$ outputs 1 after interacting with $\mathcal{A}$ and π differs by at most a negligible amount from the probability that $\mathcal{E}$ outputs 1 after interacting with $\mathcal{S}$ and ϕ.*

The asymptotics in Definition 1 are taken over a security parameter, which is naturally taken to be the length of he initial input to the environment. Defining when protocol π realizes an ideal functionality $\mathcal{F}$ is now straightforward:

Definition 2 (realizing an ideal functionality, simplified model) *Protocol π* UC-realizes *ideal functionality $\mathcal{F}$ if π UC-emulates IDEAL_F.*

Discussion. As discussed in the Introduction, UC-emulation provides a combined guarantee that includes the correctness of the outputs of the "principals" that run π, as well as the privacy of the local inputs of these principals. More precisely, correctness and privacy are guaranteed to the extent provided by the ideal functionality $\mathcal{F}$ (or, more generally, by ϕ).

That is, it is first guaranteed that the number of main parties in π, as well as their identities, are exactly the same as those of $\text{IDEAL}_{\mathcal{F}}$ (or, more generally, ϕ). Next, it is guaranteed that outputs provided by π are distributed indistinguishably from those provided by $\mathcal{F}$ (or, ϕ) on the same inputs. At the same time, it is guaranteed that any information learnt by $\mathcal{A}$ is "simulatable", in the sense that it is indistinguishable from information that is generated by $\mathcal{S}$ given only the information provided to it by $\mathcal{F}$ (or ϕ). When the output of $\mathcal{F}$ (or ϕ) is randomized, it is guaranteed that the joint distribution of the outputs of the parties running π is indistinguishable from the joint distribution of their outputs from running $\mathcal{F}$ (or ϕ); furthermore, this holds even when the outputs are viewed jointly with the output of $\mathcal{A}$ or $\mathcal{S}$, respectively.

Another point worth highlighting is that the notion of indistinguishability considered in this definition is strong, in the following sense: It considers an "interactive distinguisher" (namely, the environment) that takes an active role in trying to distinguish between the "real system" and the "specification system." That is, the definition requires that $\mathcal{E}$ is unable to distinguish between the process of running protocol π with adversary $\mathcal{A}$ and the process of running protocol $\text{IDEAL}_{\mathcal{F}}$ (or,

more generally, ϕ) with simulator $\mathcal{S}$ *even given the ability to interact with these processes as they evolve.*

At the same time, UC emulation is a relaxation of the notion of *observational equivalence* of processes (see, e.g., [M89]). Indeed, observational equivalence essentially fixes the entire system that interacts with either π or ϕ, whereas UC emulation allows the analyst to modify part of the external system (namely, substituting the simulator $\mathcal{S}$ for the adversary $\mathcal{A}$) so as to make sure that the rest of the external system cannot distinguish between π and ϕ.

2.3 Universal Composition

We present the universal composition operation and theorem. We concentrate on the case of composing general protocols; The case of ideal functionalities and ideal protocols follows as a special case. See Section 1.2 for interpretation and discussion of universal composition.

Subroutine protocols. In order to present the composition operation, we first define subroutine protocols. Let $\rho = \rho_1, ..., \rho_n$ be a protocol, and let ϕ be a subset of the machines in ρ. We say that ϕ is a subroutine protocol of ρ if ϕ is a valid protocol in of itself, and in addition none of the main parties of ρ are in ϕ (i.e., all the machines in ϕ are internal machines of ρ). For notational convenience, we sometimes refer to the machines in ϕ as $\phi_1, ..., \phi_{n'}$.

The universal composition operation. The universal composition operation is a natural generalization of the "subroutine substitution" operation from the case of sequential algorithms to the case of distributed protocols. Specifically, let ρ, ϕ and π be protocols such that ϕ is a subroutine protocol of ρ, π UC-emulates ϕ, and no machine in π has the same identity as a machine in $\rho \setminus \phi$. The composed protocol, denoted $\rho^{\phi \to \pi}$, is identical to ρ, except that the subroutine protocol ϕ is replaced by protocol π. That is, given our interpretation of protocols as sets of machines, we have $\rho^{\phi \to \pi} = (\rho \setminus \phi) \cup \pi$.

Notice that $\rho^{\phi \to \pi}$ is a valid protocol; in particular, the communication sets of the machines in $\rho^{\phi \to \pi}$ satisfy the necessary consistency requirements. (Indeed, since π UC-emulates ϕ, there must exist an identity-preserving injective correspondence between the main parties of π and those of ϕ. Furthermore, the external identities in the communication set C of each main party of π, namely the identities in C that are not part of π, appear also in the communication set of the corresponding main party of ϕ.)

Figure 4 presents a graphical depiction of the composition operation for the special case where the substituted protocol ϕ is IDEAL_F for some ideal functionality $\mathcal{F}$:

The composition theorem. Let protocol ϕ be a subroutine of protocol ρ. Say that protocol π is identity-compatible with ρ and ϕ if no machine in π has the same identity as a machine in $\rho \setminus \phi$. In its general form, the composition theorem says that if π UC-emulates ϕ, then the composed protocol $\rho^{\phi \to \pi}$ UC-emulates the original protocol ρ:

Theorem (universal composition for the simplified model): *Let ρ, ϕ, π be protocols such that ϕ is a subroutine of ρ, π UC-emulates ϕ, and π is identity-compatible with ρ and ϕ. Then protocol $\rho^{\phi \to \pi}$ UC-emulates ρ.*

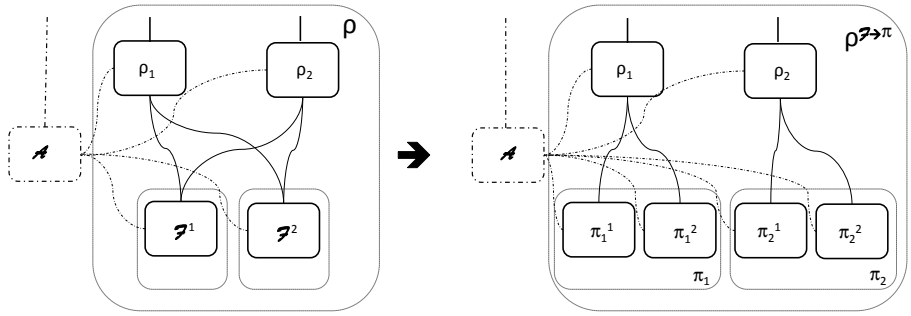

Figure 4: The universal composition operation for the case where ρ uses two copies of $\text{IDEAL}_{\mathcal{F}}$ (left figure), and each copy of $\text{IDEAL}_{\mathcal{F}}$ is replaced by a copy of π (right figure). The solid lines represent inputs and outputs. The dashed lines represent backdoor communication. The dotted lines delineate the various protocols. The "dummy parties" for the copies of $\mathcal{F}$ are omitted from the left figure for graphical clarity.

Discussion. The composition theorem 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.

A special case 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 $\rho^{\text{IDEAL}_{\mathcal{F}\to\pi}}$ UC-emulates ρ. An immediate corollary is that if ρ UC-realizes an ideal functionality $\mathcal{G}$, then so does $\rho^{\text{IDEAL}_{\mathcal{F}\to\pi}}$

The UC theorem can be applied repeatedly to substitute multiple subroutine protocols of ρ with protocols that UC-realize them. In particular, if protocols $\phi_1, ..., \phi_k$ are subroutine protocols of ρ with disjoint machines, protocol π_i UC-realizes ϕ_i for all $i = 1..k$, and $\pi_1, ..., \pi_k$ are identity-compatible and have disjoint identities, then $\rho^{\phi_1,...,\phi_k\to\pi_1,...,\pi_k}$ UC-realizes ρ, where $\rho^{\phi_1,...,\phi_k\to\pi_1,...,\pi_k} = \rho \setminus \phi_1 \setminus ... \setminus \phi_k \cup \pi_1 \cup ... \cup \pi_k$. Furthermore, repeated applications of the theorem may use nested subroutine protocols. (If k, the number of composed protocols, grows with the security parameter then there should exist a uniform bound, across all protocols, on the complexity of the simulator as a function of the complexity of the adversary.)

Proof: We sketch a natural 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 $\mathcal{S}$. We 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 $\mathcal{S}_\pi$.

Adversary $\mathcal{D}$ is a "dummy adversary" that merely serves as a "channel" between $\mathcal{E}$ and π. That is, $\mathcal{D}$ expects to receive in its input, that comes from $\mathcal{E}$, requests to deliver backdoor messages to prescribed machines of π. $\mathcal{D}$ then carries out these requests. In addition, any incoming backdoor message (from some machine of π) is forwarded by $\mathcal{D}$ to its environment. Since π UC-emulates ϕ, 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 $\mathcal{S}_\pi$. (We observe that $\mathcal{D}$ is in fact the "hardest adversary to simulate", in the sense that if there exists a simulator for the dummy adversary, then there exists

also a simulator for *any* polytime adversary. This observation, which is proven with respect to the more general model in Claim 11, is used to simplify the full proof of the UC theorem.)

The overall simulator $\mathcal{S}$ operates as follows. $\mathcal{S}$ runs $\mathcal{A}$ and follows the instructions of $\mathcal{A}$, with the exception that the interaction of $\mathcal{A}$ with the machines of π is simulated using $\mathcal{S}_\pi$. This is done by having $\mathcal{S}$ play the role of the environment for $\mathcal{S}_\pi$: When $\mathcal{S}$ receives instruction from the environment to deliver a backdoor message to a machine in π, $\mathcal{S}$ hands this request to (an internally run instance of) $\mathcal{S}_\pi$. When $\mathcal{S}_\pi$ outputs to $\mathcal{S}$ a backdoor value from some machine, $\mathcal{S}$ forwards this output value to $\mathcal{E}$.

The validity of the simulation is demonstrated via a reduction to the validity of $\mathcal{S}_\pi$: Given an environment $\mathcal{E}$ that distinguishes between an execution of ρ with $\mathcal{A}$, and an execution of $\rho^{\phi\to\pi}$ with $\mathcal{S}$, we construct an environment $\mathcal{E}_\pi$ that distinguishes between an execution of π with $\mathcal{D}$ and an execution of ϕ with $\mathcal{S}_\pi$. Essentially, $\mathcal{E}_\pi$ orchestrates for $\mathcal{E}$ an entire interaction with ρ, where the interaction with the subroutine protocol (either ϕ or π) is relayed to the external system that $\mathcal{E}_\pi$ interacts with. We then argue that if $\mathcal{E}_\pi$ interacts with π, then $\mathcal{E}$, run by $\mathcal{E}_\pi$, "sees" an interaction with $\rho^{\phi\to\pi}$. Similarly, if $\mathcal{E}_\pi$ interacts with an instance of ϕ, then $\mathcal{E}$ "sees" an interaction with ρ. Here we crucially use the fact that an execution of an entire system can be efficiently simulated on a single machine. (A more detailed proof can be derived from the one in Section 5.) □

2.4 Modeling tasks and protocols of interest

Sections 2.1 through 2.3 presented the basic analytical framework, namely the model of computation, the definitions of security and the composition theorem. While this is a good start, it is however just the beginning: To demonstrate relevance and usefulness of the framework, one would need to show how it can be used to capture computational tasks and security concerns of interest, as well as natural and realistic protocols for potentially realizing the tasks.

The question of relevance and expressive power is particularly salient here, given that the present framework pushes a considerable part of the formalism to the description of protocols and ideal functionalities. In particular, recall that the framework expects the program of a machine to include, in addition to instructions that are to be executed "in real life implementations," also "model instructions" (such as, say, "sending information to the adversary") that only take part in the security analysis. Thus, a clear way to separate "actual instructions" from "model instructions" would be an essential ingredient for adequate modeling of actual protocols and tasks.

While these are important questions, we defer addressing them to Section 6 — where they are addressed in the context of the general framework.

3 The model of computation

The treatment of Section 2 considers only systems where the number, identities, programs and connectivity of computing elements are fixed and known in advance. In fact, many definitional frameworks for distributed systems share similar limitations. While helpful in keeping the model simple, these restrictions do not allow representing many realistic situations, protocols, and threats.

This section extends the model to account for fully dynamic and evolving distributed systems of computational entities. In contrast to the treatment of Section 2, here we present this basic model separately from the model of executing a protocol (with environment and adversary). The model

of protocol execution is then formulated (in Section 4) within that basic model. Indeed, this model may be of interest in of itself, regardless of these definitions. In particular, it may potentially be used as a basis for different notions of security and correctness of distributed computation.

We strive to completely pinpoint the model of computation. When some details do not seem to matter, we 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 abstract properties that hold irrespective of any specific implementation or computational considerations.

Section 3.1 presents the basic model. Section 3.2 presents the definition of resource-bounded computation. To facilitate reading, we postpone longer discussions and comparison with other models to Section 3.3.

3.1 The basic model

As in Section 2, we start by defining the basic object of interest, namely protocols. Here however the treatment of protocols is very different than there. Specifically, recall that in Section 2 protocols come with a fixed number of computing elements, including the identities and connectivity of the elements. In contrast, here we let the identities, connectivity, and even programs of computing elements be chosen adaptively as part of the execution process. In particular the model captures systems where new computational entities get added dynamically, with dynamically generated identities and programs. It also captures the inevitable ambiguities in addressing of messages that result from local and partial knowledge of the system, and allows representing various behaviors of the communication media in terms of reliability. Further, the model facilitates taking into account the computational costs of addressing and delivery of information. We note that many of the choices here are new, and require re-thinking of basic concepts such as addressing of messages, identities, "protocol instances", and resource-bounded computation.

We proceed in two main steps. First (Section 3.1.1), we define a *syntax,* or a rudimentary "programming language" for protocols. This language, which extends the notion of *interactive Turing machine* [GMRa89], contains data structures and instructions 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 3.3. We point to the relevant parts of the discussion as we go along.

3.1.1 Interactive Turing Machines (ITMs)

Interactive Turing machines (ITM) extend 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 for ITMs. The semantics of the added syntax are described as part of the model of execution of systems of ITMs, in Section 3.1.2.

We note that the formalism given below aims to use only minimal syntax programming ab-

stractions. This is intentional, and leaves the door open to building more useful programming languages on top of this one. Also, as mentioned in the introduction, the use of Turing machines as the underlying computational 'device' is mainly due to tradition. Other computational models that allow accounting for computational complexity of programs can serve as a replacement. RAM or PRAM machines, Boolean or arithmetic circuits are quintessential candidates. See additional discussion in Section 3.3.

Definition 3 *An* interactive Turing machine (ITM) μ *is a Turing machine (as in, say, [*Si05*]) with the following augmentations:*

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

- *An* identity tape. *This tape is "read only". That is, μ cannot write to this 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 μ (namely, its state transition function and initial tape contents). We call this description the* code *of μ. The second string is called the* identity *of μ. The identity of μ together with its code is called the* extended identity *of μ.*

  (Informally, the contents of this tape is used to identify an "instance" of an ITM within a system of ITMs.)
- *An* outgoing message tape. *Informally, this tape holds the current outgoing message generated by μ, together with sufficient addressing information for delivery of the message.*
- *Three* externally writable *tapes for holding information coming from other computing devices:*
  - *An* input tape. *Informally, this tape represents information that is to be treated as inputs from "calling programs" or an external user.*
  - *A* subroutine-output tape. *Informally, this tape represents information that is to be treated as outputs of computations performed by programs or modules that serve as "subroutines" of the present program.*
  - *A* backdoor tape. *Informally, this tape represents information "coming from the adversary." This information is used to capture adversarial influence on the program. (It is stressed that the messages coming in on this tape are only a modeling artifact; they do not represent messages actually sent by protocol principals.)*

  These three tapes are read-only *and* read-once. *That is, the ITM cannot write into these tapes, and the reading head moves only in one direction.*
- *A one-bit* activation tape. *Informally, this tape represents whether the ITM is currently "in execution".*

New instructions:

- *An* external-write *instruction. Informally, 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. More concrete specification is postponed to Section 3.1.2.*

- *A* read next message *instruction. This instruction specifies a tape out of* {input, subroutine-output, backdoor}. *The effect is that the reading head jumps to the beginning of the next message on that tape. (To implement this instruction, we assume that each message ends with a special* end-of-message (eom) *character.)*[4]

Definition 4 *A* configuration *of an ITM μ consists of the contents of all tapes, as well as the current state and the location of the head. A configuration is* active *if the activation tape is set to 1, else it is* inactive.

An instance *M of an ITM μ consists of the contents of the identity tape alone. (Recall that the identity tape of μ contains the code μ, plus a string id called the identity. That is, $M = (\mu, id)$. Also, the contents of the identity tape remains unchanged throughout an execution.) We say that a configuration is a configuration of instance M if the contents of the identity tape in the configuration agrees with M, namely if the program encoded in the identity tape is μ and the rest of the identity tape holds the string id.*

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

Throughout this work we treat an ITM instance (ITI) as a rune-time object (a "process") associated with program μ. Indeed, the fact that the identity tape is read-only makes sure that it remains immutable during an execution (i.e. all configurations of the same computation contain the same value of M). In the language of Section 2, an ITI corresponds to a machine, with the exception that the communication set of the machine is not specified.

3.1.2 Executing ITMs

We specify the semantics of executing ITMs. As we'll see, an execution, even of a single ITM, might involve multiple ITIs. We thus add to our formalism another construct, called the control function, which regulates the transfer of information between ITIs. That is, the control function determines which "external-write" instructions are "allowed" within the present execution. (We note that this added mechanism provides both clarity and flexibility to the execution model: All the model restrictions are expressed explicitly and in "one place." It also facilitates defining different execution models. See more discussion in Section 3.3.) We proceed to the formal description. In the language of Section 2, the control function corresponds to (and extends) the collection of all the communication sets of all the machines in the system.

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.

[4] If a RAM or PRAM machine is used as the underlying computing unit then this instruction is redundant. However it is needed in the Turing machine setting to handle incoming messages with unbounded length. See more discussion in Section 3.3.

Executions of systems of ITMs. An execution of a system $S = (I, C)$ on input z consists of a sequence of activations of ITIs. The first activation is an activation of I, starting from the configuration where the identity tape contains the identity 0 followed by the code of I, the input tape contains the value x, and a sufficiently long random string is written on the random tape.[5] In accordance, the ITI $(0, I)$ 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). The output of an execution is the contents of first cell in the outgoing message tape of the initial ITI when it halts. 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. Specifically, the effect of an external-write instruction is the following.

Let $M = (\mu, id)$ denote the ITI which executes the instruction. Then the current contents of the outgoing message tape of M is interpreted (using some standard encoding) as a tuple:

$$(f, M', t, r, M, m)$$

where M' is an extended identity of a "target ITI", t is a tape name out of {input, subroutine-output, backdoor}, $r \in \{0,1\}$ is a reveal-sender-id flag, $f \in \{0,1\}$ is a forced-write flag, and $m \in \{0,1\}^*$ is the message. Consider the result of applying the control function C to the current execution prefix, including (f, M', t, r, M, m). If this result is *disallow*, then the instruction is not carried out and the initial ITI is activated next (i.e., its activation tape is set to 1). If C outputs *allow,* then:

1. If $f = 1$ then M' is interpreted as an extended identity $M' = (\mu', id')$. In this case:
   (a) If the ITI $M' = (\mu', id')$ currently exists in the system (namely, one of the past configurations in the current execution prefix has extended identity M'), then the message m is written to tape t of M', starting at the next blank space. If the reveal-sender-id flag is set (i.e., $r = 1$), then the extended identity $M = (\mu, id)$ of the writing ITI is also written on the same tape. The target ITI M' is activated next. (That is, a new configuration of M' is generated; this configuration is the previous configuration of M' in this execution, with the new information written on the incoming messages tape. The activation tape in this configuration is set to 1.)
   (b) If the ITI $M' = (\mu', id')$ does not currently exist in the system, then a new ITI M' with code μ' and identity id' is invoked. That is, a new configuration is generated, with code μ', the value M' written on the identity tape, and the random tape is populated as in the case of the initial ITI. Once the new ITI is invoked, the external-write instruction is carried out as in Step 1a. In this case, we say that M invoked M'.
2. If $f = 0$ then M' is interpreted as a predicate P on extended identities. Let M'' be *earliest-invoked* ITI such that $P(M'')$ holds. Then, the message m is delivered to M'' as in Step 1a.

[5] Without loss of generality, the random tape is read-once (i.e. the head can only move in one direction). This means that the tape can be thought of as infinitely long and each new location read can be thought of as chosen at random at the time of reading.

If no ITI M'' exists such that $P(M'')$ holds, then the message is not delivered and the initial ITI is activated.

When an ITI M writes a value x to the backdoor tape of ITI M', we say that M sends backdoor message x to M'. When M writes a value x onto the input tape of M', we say that M passes input x to M'. When M' writes x to the subroutine-output tape of M, we say that M' passes output x (or simply outputs x) to M.

Notation. We use the following notation. Let $\text{OUT}_{I,C}(z)$ denote the random variable describing the output of the execution of the system (I, C) of ITMs when I's input is z. Here the probability is taken over the random choices of all the ITMs in the system. Let $\text{OUT}_{I,C}$ denote the ensemble $\{\text{OUT}_{I,C}(z)\}_{z\in\{0,1\}^*}$.

Discussion: On the uniqueness of identities. Section 3.3 discusses several aspects of the external-write instruction, and in particular motivates the differences from the 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 extended identity. That is, no execution of a system of ITIs has two ITIs with the same identity and code. This property makes sure that the present addressing mechanism is unambiguous. Furthermore, the non-forced-write writing mode (where $f = 0$) allows ITIs to communicate unambiguously even without knowing the full code, or even the full ID of each other. (This is done by setting the predicate P to represent the necessary partial knowledge of the intended identity M').

Extended systems. The above definition of a system of ITMs provides several mechanisms for an ITI to specify the code and identity of the ITIs it transmits information to. It also provides mechanisms for an ITI to know the extended identity of the ITIs that transmitted information to it. These provisions are indeed crucial for the meaningfulness of the model, and may suffice for the sole purpose of describing a distributed computational system.

However, our definitions of security, formulated in later sections, make some additional requirements from the model. Recall that these definitions involve a "mental experiment" where one replaces some protocol instance with an instance of another protocol. Within the present framework, such replacement requires the ability to create a situation where some ITI M sends a message to another ITI M', but the message is actually delivered to another ITI, M'' — where M'' has, say, different code than M'. Similarly, M'' should be able to send messages back to M, whereas it appears to M that the sender is M'. Other modifications of a similar nature need to be supported as well.

The mechanism we use to enable such situations is the control function. Recall that in a system $S = (I, C)$ the control function C outputs either *allowed* or *disallowed.* We extend the definition of a control function so that it can also *modify* the external-write requests made by parties. That is, an extended system is a system (I, C) where the output of C given external-write instruction (f, M', t, r, M, m) consists of completely new set of values, i.e. a tuple $(\tilde{f}, \tilde{M}', \tilde{t}, \tilde{r}, \tilde{M}, \tilde{m})$ to be executed as above.

We note 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 modify the the external-write operations only in very specific cases and in very limited ways.

Subroutines, etc. We say that M' is a subroutine of M if M has passed input to M' or M' has passed output to M in this execution. (Note that M' may be a subroutine of M even when M' was invoked by an ITI other than M.) If M' is a subroutine of M then we say that M is a caller of M'. M' is a subsidiary of M if M' is a subroutine of M or of another subsidiary of M.

Note that the basic model does not impose a "hierarchical" subroutine structure for ITIs. For instance, two ITIs can be subroutines of each other, an ITI can also be a subroutine of itself, and an ITI can be a subroutine of several ITIs. Some restrictions are imposed later in specific contexts.

Protocols. A protocol is defined as a (single) ITM as in Definition 3. 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.)

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. (Indeed, this is the case in the framework of Section 2.) Such a definitional approach, however, does not account for protocol instances where the identities of the participants, and perhaps even the number of participants, are determined dynamically as the execution unfolds. It also does not 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 desirable.

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 of ITMs 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 3.3.

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'.

Comparison with the modeling of Section 2.1. Recall that the notion of a protocol in Section 2.1 is different than here: There, different machines in a protocol may have different programs, and there is no session ID mechanism. Indeed, a protocol in Section 2.1 most closely corresponds to an *extended instance of a protocol* in the present formalism.

The restriction for same program and a common field in the ID are useful in that they allow delineating protocols and protocol instances in an intuitive and unambiguous way, even in the present more dynamic model of computation. See more discussion in Section 3.3.2.

3.2 Polynomial time ITMs & parameterized systems

We adapt the standard notion 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. However, 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 "asymptotically 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 only interested in the second interpretation. However, recall that to argue security we often provide 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. We would like this reduction to 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 capture both interpretations.

Let $T : \mathbf{N} \to \mathbf{N}$. Traditionally, a Turing machine μ is said to be T-bounded if, given any input of length n, μ 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, say, 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 repeatedly 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 expect each message to include a special field, called the import field of the message. The import field contains a natural number called the import of the message. We then define the runtime budget, n, of an ITI at a certain configuration to be the sum of the imports of the messages received by the ITI, *minus the imports of the messages sent by the ITI.* As we'll see, this provision allows guaranteeing that, for all "reasonable" functions T (specifically, whenever T is increasing and super-additive), the overall number of steps taken in a system of ITMs which are all T-bounded is finite. In fact, this number is bounded by $T(n)$, where n is the import of the initial input to the system (namely, the import of the message written on the input tape of the initial ITI in the initial configuration). Intuitively, this provision treats the imports of messages as "tokens" that provide runtime. An ITI receives tokens when it gets incoming messages with import, and gives out tokens to other ITIs by writing messages with import to other ITIs. 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 tapes of each other.

Definition 5 (T-bounded, PPT) *Recall that the* import *of a message is the value written in the import field of the message. Let $T : \mathbf{N} \to \mathbf{N}$. An ITM μ is* locally T-bounded *if, at any prefix of an execution of a system of ITMs, any ITI M with program μ satisfies the following condition. 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 imports of the messages written by other machines on the externally writable tapes of M, and n_O is the overall imports of the messages written by M on externally writable tapes of other machines.*

If μ is locally T-bounded, and in addition either μ does not make external-writes with forced-write, or each external-write with forced-write specifies a recipient ITM which is T-bounded, then we say that μ is T-bounded. *μ is* PPT *if there exists a polynomial p such that μ is p-bounded. A protocol is PPT if it is PPT as an ITM.*

Let us briefly motivate two ways in which Definition 5 departs from the traditional formulation of resource-bounded computation. First, in contrast to the traditional notion where only *inputs*

count for the resource bound, here we allow even messages written to other tapes of the recipient to "provide runtime". Furthermore, we separate the import of a message One use of this extra generality is in modeling cases where ITIs are invoked (i.e., activated for the first time) via a message that's not an input message.

Third, here the import is represented in binary, instead of the traditional "length of message" convention. The present convention allows for significant additional expressive power in some situations. (For instance, consider the following two situations: In one situation we have an algorithm that uses a large number ℓ of subroutines, where all subroutines expect to receive a single input message and take the same number, n, of computational steps. In the other situation, it is known that one of the subroutines will eventually need n computational steps, whereas the remaining subroutines will require only m steps, where $m << n$. Still, it is not known in advance which of the subroutines will need n steps. The traditional length-of-input formalism does not distinguish between the two situations, since in both the overall algorithm must take time ℓn. (Indeed, it takes ℓn time only to invoke the ℓ subroutines.) In contrast, the present formalism allows distinguishing between the two situations: In the first one the overall algorithm runs in time ℓn, whereas in the second it runs in time only $\ell(m + \log n) + n$.

For clarity and generality we refrain from specifying any specific mechanism for making sure that ITMs are T-bounded by some specific function T. In Section 3.3.4 we informally discuss some specific methods, as well as other potential formulations of resource-bounded distributed computation.

Consistency with standard notions of resource-bounded computation. We show that an execution of a resource-bounded system of ITMs can be simulated on a standard TM with comparable resources. That is, 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 6 *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 μ, which takes for input the initial input x and runs in time $O(T(n)T'(T(n)))$ where n is the import of x. The same holds also for extended systems of ITMs, as long as all the ITMs invoked are T-bounded.*

In particular, if both I and C are PPT then so is μ. Furthermore, if the import of x is taken to be its length, then μ is PPT in the standard length-of-input sense.

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 $T(n)$, where n is the import of the initial input of I. As mentioned above, this can be seen by treating the bits of resource messages as "tokens" that give runtime. Initially, there are n 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. Let m_i be the set of ITIs that were active up till the ith configuration in the execution. For each $M \in m_i$ let $n_{M,i}$ be the overall import of the messages received by ITI M at the last configuration where it was active before the ith configuration in the execution, minus the overall import of the messages

written by M to other ITIs in all previous configurations. Since I is T-bounded we have that M is also T-bounded, namely for any i, the number of steps taken by each $M \in m_i$ is at most $T(n_{M,i})$. It follows that $i = \sum_{M \in m_i}(\#\text{ steps taken by } M) \leq \sum_{M \in m_i} T(n_{M,i})$. By super-additivity of T we have that $i \leq \sum_{M \in m_i} T(n_{M,i}) \leq T(\sum_{M \in m_i} n_{M,i})$. However, $\sum_{M \in m_i} n_{M,i} \leq n$. Thus $i \leq T(n)$.

The machine μ 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 μ, it thus remains to bound the time spent on evaluating the control function C. However, C is evaluated at most $T(n)$ times, on inputs of import at most $T(n)$ 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 there is a common value, called the security parameter, that serves as a minimum value for the initial runtime budget of each ITI. More specifically, we say that an ITM is parameterized with security parameter k if it does not start running unless its overall import is at least k. A system of ITMs is parameterized with security parameter k if all the ITIs ever generated in the system are parameterized with security parameter k.

Subsequent sections will concentrate on the behavior of systems where the import of the initial input to the system, i.e. the import of the input to the initial ITI, is at most some function of (specifically, polynomial in) the security parameter.

3.3 Discussion

This section contains more lengthy discussion that highlights and motivates the main aspects of the model of computation, as well as the definition of polynomial time distributed computation.

Other 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 the 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], the Abstract Cryptography model of Maurer and Renner [MR11], and the equational approach of Micciancio and Tessaro [MT13]. (Other approaces are mentioned in the Appendix.) Throughout, we briefly compare the present model with some of these approaches.

We very roughly partition the discussion to four parts. It is stressed however that the partitioning is somewhat arbitrary and all topics are of course inter-related.

3.3.1 Motivating the use of ITMs

A first definitional choice is to use an explicit, imperative formalism as the underlying computational model. That is, a computation is represented as a sequence of mechanical steps (as in Turing

machines) rather than in a functional or descriptive way (as in the λ-calculus) or in a denotational way (as in Domain Theory). Indeed, while this imperative model is less "elegant" and not as easily amenable to abstraction and formal reasoning, it most directly captures the *complexity* of computations, as well as physical side-effects of certain types of basic computational steps. Indeed, this modeling provides a direct way of capturing the interplay between the complexity of local computation, communication, randomness, "physical side channels," and resource-bounded adversarial activity. This interplay is often at the heart of the security of cryptographic protocols.

Moreover, the imperative formalism strives to faithfully represent 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, 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 imperative 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.

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

Several imperative 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 only a relatively low level abstraction of computer programs and protocols. In contrast, current literature describes protocols in a much higher-level (and often informal) language. One way to bridge this gap is to develop a library of subroutines, or even a programming language that will allow for more convenient representation of protocols while not losing the correspondence to ITMs (or, say, interactive 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]. We leave this line of research for future work.

3.3.2 On the identity mechanism

The extended identity, i.e. the contents of the identity tape, is the mechanism used by the model to distinguish between ITIs (representing computational processes) in a distributed computation. That is, the model guarantees that no two ITIs have the same extended identity. Furthermore, the identity of an ITI M is determined by the ITI that invokes (creates) M. While it is fully accessible to the ITI itself, the extended identity cannot be modified throughout the execution. Finally, the extended identity is partitioned into three parts: The code (program), the session ID, and the party ID. We motivate these choices.

Identities are algorithmically and externally chosen. The fact that the identity is determined by the process that creates the new ITI is aimed at representing natural software engineering practice. Indeed, when one creates a new computational process, one usually provides the program — either directly or via some proxy mechanism — plus sufficient information for identifying this process from other ones. Furthermore, it has been demonstrated that externally chosen identities are essential for performing basic tasks such as broadcast and Byzantine agreement within a general distributed computing framework such as the present one [LLR02].

Allowing the creating ITI to determine identities algorithmically allows expressing this natural real-world ability within the model. The protocol of [B+11] is an example for how this ability can be meaningfully used, and then analyzed, within the present model.

Preventing an ITI from modifying its own identity is done mainly to simplify the delineation of individual computational processes in a system. We allow ourselves to do it since no expressive power appears to be lost (indeed, ITIs can always invoke other ITIs with related identities and programs).

Including the code in the identity. Including the ITI's code in the (extended) identity is a useful tool in the identification of properties of ITIs by other ITIs. Indeed, the external-write mechanism (discussed at more length later on) allows the recipient ITI to sometimes see the code of the sending ITI. It also allows the delivery of messages to depend on the code of the recipient ITI. This convention might appear a bit odd at first, since it is of course possible to write the code of an ITI in an extremely generic way (e.g. as a universal Turing machine) and then include the actual program in the first message that the ITI receives. Furthermore, verifying practically any interesting property of arbitrary code is bound to be impossible.

Still, general undecidability notwithstanding, it is indeed possible to write code that will make it easy to verify that the code has certain desirable properties, e.g. that it implements some algorithm, that it does not disclose some data, that it only communicates with certain types of other ITIs, that its runtime is bounded, etc. This allows the protocol designer to include in the protocol π instructions to verify that the ITIs that π interacts with satisfy a basic set of properties. As we'll see, this is an extremely powerful tool that makes the framework more expressive.

On globally unique identities. The guarantee that extended identities are globally unique throughout the system simplifies the model and facilitates protocol analysis. However, it might appear at first that this "model guarantee" is an over-simplification that does not represent reality. Indeed, in reality there may exist multiple processes that have identical programs and identities, but are physically separate and are not even aware of each other. To answer this concern we note that such situations are expressible within the present model - simply consider protocols that ignore a certain portion of the identity. (We note that other formalisms, such as the IITM model [DKMR05, K06, KT13], *mandate* having part of the identity inaccessible to the program.)

Furthermore, the model allows multiple ITIs in an execution have the same (non-extended) identity - as long as they have different programs. This again underlines the fact that identity does not guarantee uniqueness in of itself.

On the SID mechanism. The SID mechanism provides a relatively simple and flexible way to delineate individual protocol instances in a dynamically changing distributed system. In particular

it allows capturing, within the formal model, the intuitive notion of "creating an instance of a distributed protocol" as a collection of local actions at different parts of the system.

Indeed, *some* sort of agreement or coordination between the entities that create participants in a protocol instance is needed. The SID mechanism embodies this agreement in the form of a joint identifier. We briefly consider a number of common methods for creating logically-separate protocol instances in distributed systems and describe how the SID mechanism fits in. Finally we point out some possible relaxations.

One simple method for designing a system with individual protocol instances is to set all the protocol instances statically, in advance, as part of the system design. The SID mechanism fits such systems naturally - indeed, here it is trivial (and convenient) to ensure that all ITIs in a protocol instance have the same SID.

Another, more dynamic method for designing systems with multiple individual protocol instances is to have each protocol instance start off with a single ITI (representing a computational process within a single physical entity) and then have all other ITIs that belong to that protocol instance be created indigenously from within the protocol instance itself. This can be done even when these other ITIs model physical processes in other parts of the system, and without prior coordination - say by sending of messages, either directly or via some subroutine. Indeed, most prevalent distributed protocols (in particular, client-server protocols) fall into this natural category.

The SID mechanism allows capturing such protocols in a straightforward way: The first ITI in a protocol instance (π, s) is created, by way of an incoming input that specifies code π and SID s. All the other ITIs of this instance are created, with the same SID and code, by way of receiving communication from other ITIs in this instance. (Jumping ahead, we note that in the model of protocol execution described in the next sections, receiving network communication is modeled by way of receiving subroutine-output from an ITI that models the actual communication. This ITI is a subroutine of both the sending ITI and of the receiving ITI.) All the functionalities in Section 6 are written in this manner.

Alternatively, one may wish to design a system where protocol instances are created dynamically, but computational processes that make up a new protocol instance are created "hierarchically" via inputs from existing processes rather than autonomously from within the protocol instance itself. Here again the SID mechanism is a natural formalism. Indeed, if the existing processes (ITIs) have sufficient information to create new ITIs with that have the same SID then the creation of a new protocol instance can be done without additional coordination. When this is not the case, additional coordination might be needed to agree on a common SID. See [BLR04, B+11] for a protocol and more discussion of this situation.

Either way, we stress that the session ID should not be confused with values that are determined (and potentially agreed upon) as part of the execution of the protocol instance. Indeed, the SID is determined before the instance is invoked and remains immutable throughout.

One can also formulate alternative conventions regarding the delineation of protocol instances. For example one may 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 remark that other frameworks, such as [HS11], put additional restrictions on the format of the SIDs. Specifically, in [HS11] the SID of a protocol instance is required to include the SID of the calling protocol instance, enforcing a "hierarchical" SID structure. 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 or not.

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. Modeling ephemeral and reusable identities can be done via protocol-specific structures that is separate from the identity mechanism provided by the model.

3.3.3 On the external-write mechanism and the control function

As discussed earlier, traditional models of distributed computation model inter-component communication 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 static "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 a single computational step regardless of the number of components in the system or the length of the message.

That modeling of the communication is clean and elegant. It also facilitates reasoning about protocols framed within the 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, as discussed earlier, this modeling does not allow representing realistic situations where the number and makeup of components changes as the system evolves. It also does not capture commonplace situations where the sender has only partial information on the identity or code of the recipient. 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 instruction, together with the control function, are aimed at providing a sufficiently expressive and flexible mechanism that better captures the act of transmitting information from one process (ITI) to another. We highlight and motivate salient aspects of this mechanism.

Invoking new ITIs. The model allows for dynamic invocation of new ITIs as an algorithmic step. This feature is important for modeling situations where parties join a computation as it unfolds, and moreover where parties "invite" other parties to join. It is also crucial for modeling situations where the numbers of ITIs and protocol instances that run in the systems are not known in advance.

Indeed, such situations are commonplace: Open peer-to-peer protocols (such as, e.g., public block-chain systems), client-initiated interaction with a server here the server learns that the client exists only via a message of the protocol itself, programs that are generated algorithmically, and then "downloaded" and incorporated in a computation "on the fly".

Identifying the recipient ITI. A basic tenet of the external-write mechanism is that the writing ITI is responsible for identifying the recipient ITI in a sufficiently unambiguous way. The external-write operation provides two different modes for identifying the recipient. These modes are captured by the value of the forced-write flag. When the flag is set, an external-write to an ITI that does not exist in the sysem results in the creation of a new ITI. When the flag is not set, an external-write to an ITI that does not exist in the system is either directed to an existing ITI that best matches the specification provided in the operation, or fails if no existing ITI matches the specification.

The two modes represent two different real-life operations: The first mode represents the creation of a new computational process. Here the full information regarding the identity and program of the process must be provided. In contrast, the second mode represents information passed to an existing process without intent to create a new one. Here there is no need to completely specify the identity and program of the recipient; one only needs to specify the recipient well enough so as to enable delivery. This flexibility is convenient in situations where the sender does not know the full code, or even the full identity, of the recipient.

Two additional comments are in order here: First it is stressed that the writing ITI is not notified whether the target ITI M' currently exists in the system. Indeed, incorporating such a "built-in" notification mechanism would be unnatural for a distributed system.[6]

Second, we note that the predicate-based mechanism for determining the recipient ITI in case that $f = 0$ allows much flexibility - all the way from completely determining the target extended identity to allowing almost any other ITI. One can restrict the set of predicates allowed by setting appropriate control functions. We also note that the convention of picking the *first*-created ITI that satisfies the given predicate P is convenient in that it guarantees consistency: If at any point in the execution a message with predicate P was delivered to an ITI M then all future messages that specify predicate P will be delivered to M. This holds even when there are multiple ITIs that satisfy P, and even when new ITIs that also satisfy P are added to the system.

Identifying the sending ITI. The external-write mechanism provides two modes regarding the information that the recipient ITI learns about the identity and program of the writing ITI: If the writing ITI sets the reveal-sender-id flag to 1, then the recipient ITI learns the extended identity of the sending ITI, If the flag is 0, then the receiving ITI does not get any information regarding the identity of the writing ITI.

These two modes represent two "extremes:" The first mode represents the more traditional "fixed links communication" where the recipient fully knows the identity and program of the sending entity. This makes sense, for instance, where the recipient ITI is the one that invoked the sender ITI as a subroutine, and the current message is an output of the subroutine, returned to its caller. (In this case, the target tape will be the subroutine-output tape.)

The other extreme represents situations where the recipient ITI has no knowledge of the sending ITI, such as an incoming message on a physical communication link coming from a remote and

[6] We remark that previous versions of this work did, unintentionally, provide such an implicit notification mechanism. Specifically, they did not allow the co-existence of ITIs with the same identity and different codes. This meant that an external-write to an ITI that differs from an existing ITI only in its code would fail. This allowed some unnatural 'model attacks' where an ITI A, that knows that an ITI B is planning to invoke an ITI C could affect the behavior of B by simply creating an ITI C' that has the same identity as C but different code. This would cause B's request to create C to fail. Such transmission of information from A to B without explicitly sending messages does not reflect realistic attacks, and interferes with the definitions of security in later sections.

unknown source.

It is of course possible to extend the formalism to represent intermediate situations, such as the natural situation where the recipient learns the identity of the sending ITI but not its code, or perhaps only some partial information on the identity and code. We chose not to do it for sake of simplicity, as the present two modes suffice for our purposes. (Also, one can capture these intermediate situations within the model by having the sending ITI perform a two-step send: The sending ITI M creates a new ITI M'' that receives the message from M with reveal-sender-id flag 1, and sends it to the recipient M' along with the partial information on M, with the reveal-sender-id 1. This way, the recipient learns the specified partial information on M. Seeing the code of M'' allows the recipient to trust that the partial information on M, provided by M'', is correct.)

Allowing the recipient to see the code of the sending ITI enables the recipient to make meaningful decisions based on some non-trivial properties of that code. (The mechanism proposed in the previous paragraph is an example of such usage, where M' verified properties of the code of M''.) We note that this requires writing code in a way that allows salient properties to be "recognizable" by the recipient. 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.

Jumping to the next received message. Recall that Definition 3 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: 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.

The control function as an ITM. The control function is a convenient mechanism, in that it allows separating the definition of the basic communication model from higher-level models that are then used to capture more specific concerns and definitions of security. Indeed, a number of other definitions (such as [DKMR05, CV12, CCL15, CSV16]) can be captured within the present basic framework, by using appropriate control functions.

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 input 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.

3.3.4 On capturing resource-bounded computations

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 5 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. This may indeed be regarded as a barrier to general applicability of these notions.

An alternative notion of time-bounded computation: Imposing an overall bound. Recall that it does not suffice 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 5, we might still have unbounded executions of systems even when all the ITMs are bounded. Definition 5 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 direct "linking" of 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 M. 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 M, together with the ITIs that M 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 6. 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 is 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.[7]

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

[7] 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.

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.

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 and Ralf were particularly instructive.

4 Defining security of protocols

This section presents the definition of protocols that securely realize a given ideal functionality. At high level, the definition is the same as the one in Section 2.2. However, it is formulated within the general model of computation of Section 3, and therefore applies to a wider class of protocols and computational environments.

Section 4.1 presents the model for protocol execution. Section 4.2 defines protocol emulation. Section 4.3 defines ideal functionalities and ideal protocols for carrying out a given functionality, followed by the definition of securely realizing an ideal functionality. Sections 4.4 and 4.5 present some simplified alternative formulations and variants of the definitions.

4.1 The model of protocol execution

The model of protocol execution extends the model of protocol execution from Section 2.2 to the formalism of Section 3.1. As in the case of Section 2.2, the same model to capture executions of protocols and also the ideal process for an ideal functionality. Furthermore, as there, the model does not explicitly represent "communication links" between parties. Also, as there, the model does not include an explicit provision for representing "corrupted parties." Section 6 discusses and exemplifies how to capture several common communication and party-corruption models on top of the basic model of execution presented in this section.

In addition, as in Section 2.2, the current model considers only protocols π where the subroutine structure is "hierarchical", in the following sense: No ITI M is a subsidiary of an instance of protocol π while at the same time being a subroutine of an ITI M' who is not a subsidiary of this instance of π. We leave modeling and arguing about the security of protocols that do not abide by this restriction out of scope.

Formally, the model of protocol execution is defined in terms of a system of ITMs, as formulated in Section 3.1.1. Recall that a system of ITMs consists of an initial ITM and a control function. The initial ITM will correspond to the environment. The control function will encode the adversary, the protocol, and the rules of how the various ITIs can communicate with each other.

Before proceeding to the actual definition, let us highlight some of the challenges in extending the definitional ideas from the setting of Section 2.2 to the present setting. Recall that the mechanism for ITI communication, namely the external-write mechanism, mandates that the writing ITI be

aware of the identity (and sometimes program) of the recipient. Furthermore, the mechanism sometimes allows the recipient ITI to know the identity and program of the sender. This appears to be incompatible with the "subroutine substitution" composition operation - at least as defined in Section 2. Indeed, subroutine substitution replaces the program of the subroutine with a different program, and furthermore the calling ITI should be oblivious to this replacement. The model will thus need to provide a way to reconcile these two contradicting requirements.

Furthermore, since our model involves a single environment machine that takes on the role of multiple processes (ITIs) other than those that belong to the analyzed protocol instance, we must also provide a mechanism for making inputs coming from the environment appear, to the recipient, as inputs coming from other ITIs.

The model of computation will reconcile these requirements, as well as other similar ones, via an appropriate control function.

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

An execution of protocol π, adversary $\mathcal{A}$, and environment $\mathcal{E}$ is a run of an extended, parameterized system of ITMs (see Section 3.1.1) with initial ITM $\mathcal{E}$, and the following control function $C^{\pi,\mathcal{A}}_{\text{EXEC}}$:

1. $\mathcal{E}$ may pass inputs to ITIs of its choice, as long as all of these ITIs have the same session ID. The control function sets the code of all these ITIs to π. $\mathcal{E}$ can also specify the code and ID of the writing ITI to values other than its own. In addition, it may pass inputs to $\mathcal{A}$, who is identified by having party ID $\perp$. The *test session ID, s,* is the session ID of the ITIs written to by $\mathcal{E}$. The *test instance of* π is the instance of π with session ID s.
2. $\mathcal{A}$ may write only to the *backdoor tapes* of *existing* ITIs. That is, $\mathcal{A}$'s external-write requests must have the forced-write flag *unset*.
3. external-write operations by ITIs other than $\mathcal{E}$ and $\mathcal{A}$ must always include the sender extended identity in the outgoing message, namely the "reveal-sender-id" flag must be set.

   These ITIs are allowed to write to the backdoor tape of $\mathcal{A}$. (Here the forced-write flag must be *unset,* and the recipient code must not be specified.)

   In addition, these ITIs may pass inputs and outputs to any ITI other than $\mathcal{A}$ and $\mathcal{E}$, with the following modification: If the writing ITI is a main party of the test session of π, the target tape is the subroutine-output tape, and the target ITI is not part of the extended test session of π, then the message is written to the subroutine-output tape of $\mathcal{E}$. Here $\mathcal{E}$ receives the sender identity, but not the sender code.

Figure 5: A summary of the model for protocol execution

The model. Given ITMs $\pi, \mathcal{E}, \mathcal{A}$, the model consists of the extended, parameterized system of ITMs $(\mathcal{E}, C^{\pi,\mathcal{A}}_{\text{EXEC}})$, where the initial ITM in the system is the environment $\mathcal{E}$, and the control function $C^{\pi,\mathcal{A}}_{\text{EXEC}}$ is defined below and summarized in Figure 5.

The effect of external-writes made by $\mathcal{E}$. The environment $\mathcal{E}$ may pass inputs (and only inputs) to ITIs of its choice, as long as all of these ITIs have the same session ID. The control function sets the code of all these ITIs to π. $\mathcal{E}$ can also specify the code and ID of the writing ITI to values other than its own. The adversary is identified by having party ID $\perp$. That is,

if the party ID of the target ITI is $\perp$ then the code of the target ITI is set by the control function to be $\mathcal{A}$.

More precisely, an external-write operation (f, M', t, r, M, m) by $\mathcal{E}$, where $f \in \{0,1\}$ is the forced-write flag, M' is an extended identity of a "target ITI", t is the tape name, r is the reveal-sender-id flag, M is an extended identity of a "source ITI", and m is the message, is handled as follows.

If t is not the input tape or $f = 0$, or the session ID of M' is different than the session ID of any other M' that appeared in an external-write operation of $\mathcal{E}$ in the execution so far, then the operation is rejected.

Else, if the PID of M' is $\perp$, then m is written to the input tape of the ITI whose identity is that of M' and whose code is $\mathcal{A}$. (As usual, if no such ITI exists then one is invoked.)

Else m is written to the input tape of the ITI whose identity is that of M' and whose code is π. In that case, if $r = 0$ then no sender identity appears on the recipient input tape. If $r = 1$ then M appears on the input tape of the recipient as the source extended identity.

Let the *test session ID, s,* be the session identifier of the ITIs written to by $\mathcal{E}$. Let the *test session of* π be the set of all ITIs whose code is π and whose session ID is s.

The effect of external-writes made by $\mathcal{A}$. The control function allows the adversary $\mathcal{A}$ (i.e., the ITI with party ID $\perp$) to write only to the *backdoor tapes* of ITIs. In addition, it can only write to tapes of existing ITIs; that is, $\mathcal{A}$'s external-write requests must have the forced-write flag *unset*.

The effect of external-writes made by other ITIs. external-write operations by ITIs other than $\mathcal{E}$ and $\mathcal{A}$ must always include the sender extended identity in the outgoing message - namely the "reveal-sender-id" flag must be set.

These ITIs are allowed to write to the backdoor tape of $\mathcal{A}$. Here the forced-write flag must be unset, and the recipient code must not be specified. (This way the message is delivered regardless of the code of the adversary.)

In addition, ITIs other than $\mathcal{E}$ and $\mathcal{A}$ may pass inputs and outputs to any ITI other than $\mathcal{A}$ and $\mathcal{E}$, subject to the following modification: If the target tape is the subroutine-output tape, the source ITI is a main party of the test session of π, and the target ITI is not part of the extended test session of π, then the value is written on the subroutine-output tape of $\mathcal{E}$, along with the target extended ID and sender identity (but not the sender code).[8]

Let $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(z) \stackrel{\text{def}}{=} \text{OUT}_{\mathcal{E},C^{\pi,\mathcal{A}}_{\text{EXEC}}}(z)$. (We think of the input z to the initial ITM $\mathcal{E}$ as representing some initial state of the environment in which the protocol execution takes place.)

Subroutine respecting protocols. The model of protocol execution is a highly simplified rendering of an actual execution environment where the test instance of the protocol runs alongside

[8] Deciding whether the target ITI is part of the extended test session of π may require a global view of the system. In particular, the writing ITI may not always know which is the case. Jumping ahead, we note that demonstrating that one protocol UC-emulates another might well require arguing about the behavior of the two protocols in such cases.

other ITIs. In particular, this model does not allow for ITIs other than $\mathcal{E}$, $\mathcal{A}$, and the extended test instance of π. Furthermore, neither $\mathcal{E}$ nor $\mathcal{A}$ are allowed to provide inputs to the ITIs which are not main parties of the test instance of π. This in particular means that ITIs that are subsidiaries of the test session are never faced with a situation where they might obtain an input or subroutine-output from an ITI which is not already a member of this extended session. Similarly, members of the extended test session cannot possibly generate inputs to an already-existing ITI that is not already a member of the extended test session.

In contrast, when executed in an actual system where other computational entities (ITIs) may coexist, such situations might indeed happen, and might lead to attacks. To make sure that such attacks are not possible, we restrict attention to protocols that guarantee hierarchical information flow of inputs and outputs. Concretely, we say that protocol π is subroutine respecting if all subsidiary ITIs of some session, s, of π satisfy the following two conditions, in any system (even in systems where π runs alongside other protocols):

1. The ITI ignores any input or subroutine-output value passed from an ITI which is not already a main party or subsidiary of session s. (Here and elsewhere, ignoring a message subject to a condition on the sender identity means that the ITI first reads the sender identity of each incoming message, and if the condition on this identity is not satisfied then the rest of the message is not read and the activation ends.)

2. The ITI does not pass any subroutine-output to an ITI that is not a member of the extended session of s, nor does it pass any input to an *existing* ITI that is not already a member of the extended session of s.[9]

We remark that in the simplified model of Section 2.2 there was no need to specifically define subroutine respecting protocols. Indeed, the static definition of protocols there guarantees that protocols are always subroutine respecting.

4.2 Protocol emulation

This section formalizes the general notion of emulating one protocol via another protocol, generalizing the definition of Section 2.2. We first define two technical restrictions on the environments we consider, and then proceed to the actual definition.

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 relative imports of inputs given to the parties. Recall that we have already restricted ourselves to parameterized systems where the import of input to each party must be at least the security parameter. Furthermore, the definition of protocol emulation will concentrate on the case where the import of the input to the environment is polynomial in the security parameter. However, these restrictions do not limit the relative imports of the inputs that the environment provides to the adversary and to the other ITIs; the difference can be any arbitrary polynomial in the security

[9] One way to implement this requirement is to make sure that all subroutine ITIs invoked by the main parties of s are s-compliant, where an ITI is s-compliant if: (a) It has s written in its code; (b) It ignores inputs and subroutine-outputs from ITIs that are neither main parties of session s nor s-compliant; (c) Generates inputs and subroutine-outputs only to the main parties of session s and to ITIs which are s-compliant.

parameter, and the ratio can be arbitrary. Consequently, the model still allows the environment to create situations where the import of the input to the protocol, hence the protocol's complexity and communication complexity, are arbitrarily large relative to the input import 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 8) becomes overly restrictive.[10]

To avoid such situations, we wish to consider only environments where the amount of resources given to the adversary (namely, the overall import of the inputs that the adversary receives) is comparable to the amount of resources given to the other ITIs in the system. To be concrete, we consider only environments where, at any point in time during the execution, the overall import of the inputs given to the adversary is at least the sum of the imports of all the other inputs given to all the other ITIs in the system so far. That is, if at a certain point in an execution the environment provided import $n_1, ..., n_k$ to k ITIs overall, then the overall import of the inputs to the adversary is at least $n_1 + ... + n_k$. We call such environments balanced. It is stressed that the import given to the adversary can still be arbitrarily (but polynomially) large relative to the overall imports given by the environment to the parties.

External identities and identity-bounded environments So far, the model of protocol execution allows the environment to assume any identity when providing inputs to the main parties of the test session. (That is, the environment can set the extended identity of the source of any input to any value.) We call these identities external. Allowing the environment to use external identities is essential for the meaningfulness of the model, in that it allows representing situations where the main parties of the protocol instance under consideration receive input from entities that are external to the protocol. Furthermore, requiring the protocol to consider such environments will be crucial for the composition theorem to hold. Still, it will be convenient to consider also restricted environments that can use only external identities of some form. Specifically, say that an environment is ξ-identity-bounded if it uses only external identities in the set (or, that satisfy the predicate) ξ.[11]

Distribution ensembles and indistinguishability. Towards the formal definition, we recall the definitions of 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

[10] Recall that 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 can no longer be meaningful; In particular, the simulated adversary may not even be able to run the code of the protocol in question. In fact, if such situations were allowed then there would exist simple protocols that do not even emulate themselves.

[11] The decision of which extended identities to designate as external ones can be viewed as part of the protocol design process. Jumping ahead, the more restricted the set ξ, the easier it will be to prove security of the protocol, and the harder it will be to use it. (This will become apparent in the notion of compliant protocols in Section 5.) The present formalism allows making the determination based on simple constructs, such as having the party or session ID be (or include) a specific string, or alternatively more complex criteria that involve the actual code of the ITI.

the security parameter. As we'll see, it will suffice to restrict attention to binary distributions, i.e. distributions over $\{0,1\}$.

Definition 7 *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 $z \in \cup_{\kappa \leq k^d}\{0,1\}^\kappa$ we have:*

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

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 $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}} \stackrel{\text{def}}{=} \{\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(k,z)\}_{k\in\mathbf{N},z\in\{0,1\}^*}$. It is stressed that Definition 7 considers the distributions $X(k,z)$ and $Y(k,z)$ only when the import of z is polynomial in k. This essentially means that we consider only situations where the initial input to the environment is some polynomial function of the security parameter. We are finally ready to formally define UC protocol emulation:

Definition 8 *Let π and ϕ be subroutine-respecting 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:*

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

If the quantification is restricted to ξ-identity-bounded environments for some predicate ξ, then we say that π UC-emulates *ϕ* with respect to *ξ-identity-bounded environments.*

We refer the reader to Section 2.2 for interpretations of this definition and discussion.

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 3. 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). In this context, the PID of an ideal functionality is often 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. We extend the concept of dummy parties from Section 2.2 to the present model. Recall that the idea is to have a "wrapper" mechanism that makes an instance of an ideal functionality look, syntactically, like an instance of multi-party protocol. This "wrapper" mechanism is aimed at facilitating the process of designing protocols that use ideal functionalities as subroutines, and then replacing an ideal functionality with an instance of a distributed protocol that UC-emulates this functionality.

The dummy parties of an instance of an ideal protocol will have the same session ID as the ideal functionality. They will be distinguished via their party IDs. In more detail, the ideal protocol $\text{IDEAL}_{\mathcal{F}}$ for an ideal functionality $\mathcal{F}$ is defined as follows. Let (p, s) be the party and session ID. Then:

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 $id = (s, p)$: If the reveal-sender-identity flag not set, then do nothing. Else, pass input (v, eid_c) to an instance of $\mathcal{F}$ with identity $(s, \perp)$, with the forced-write reveal-sender-identity flags set. (Recall that setting the forced-write flag implies that if ITI $(\mathcal{F}, (s, \perp))$ does not exist, then one is invoked.)

2. When activated with subroutine-output $(v, (s, p), eid_t)$ from an ITI with code $\mathcal{F}$ and identity $(s, \perp)$, where v is the actual output value, pass output v to the ITI with extended identity eid_t with reveal-sender-identity and forced-write flags set.

3. Messages written on the backdoor tape, including corruption messages, are ignored. (The intention here is that, in the ideal protocol, the adversary should give corruption instructions directly to the ideal functionality. See more discussion in Section 6.1.3.)

Note that the ideal functionality $\mathcal{F}$ is, technically, a different protocol instance than $\text{IDEAL}_{\mathcal{F}}$. In particular, it cannot be accessed directly by the environment.

To make sure that $\text{IDEAL}_{\mathcal{F}}$ is formally PPT, without over-complicating the task of protocol design, we use the following convention. The polynomial bounding the runtime of $\text{IDEAL}_{\mathcal{F}}$ will be the same as the polynomial p bounding the runtime of $\mathcal{F}$. Upon receiving input with import n, $\text{IDEAL}_{\mathcal{F}}$ will pass as much of this import as it can to $\mathcal{F}$. That is, $\text{IDEAL}_{\mathcal{F}}$ will pass to $\mathcal{F}$ the largest import n' such that $p(n - n') \geq k$, where k is the number of steps taken by $\text{IDEAL}_{\mathcal{F}}$ in this activation. Similarly, upon receiving from $\mathcal{F}$ subroutine-output with import n, $\text{IDEAL}_{\mathcal{F}}$ will pass to the target ITI the largest import n' such that $p(n - n')$ still covers the number of steps done by $\text{IDEAL}_{\mathcal{F}}$ in this activation. (In both cases, if $p(n) < k$ then the activation ends without doing anything.) Notice that this mechanism guarantees that $n - n'$ can be made arbitrarily small by making p large enough. This convention allows the protocol designer to use the approximation $n' \sim n$ without sacrificing much in the precision of the analysis. In other words, the dummy parties are essentially 'transparent' in terms of the import of inputs and subroutine-outputs.[12]

Thanks. We note that previous formulations of the ideal protocol ignored the need to keep dummy parties polytime. Previous formulations of the dummy adversary had similar issues, leading to incorrect proofs of Claims 11 and 13. We thank Ralf Küsters for pointing out these errors.

[12] Alternatively, one can simply keep the definition of $\text{IDEAL}_{\mathcal{F}}$ as described in items 1-3, and allow $\text{IDEAL}_{\mathcal{F}}$ to not be polynomially bounded. This would not change the validity of the modeling and analysis, nor would it invalidate any of the results in this work. (In particular, the conclusion of Proposition 6 would hold even if the system included ideal protocols where $\text{IDEAL}_{\mathcal{F}}$ is as proposed here.) Still, for clarity and simplicity we make sure that $\text{IDEAL}_{\mathcal{F}}$ is polytime.

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* $\text{IDEAL}_{\mathcal{F}}$*, the ideal protocol for $\mathcal{F}$.*

4.4 Alternative formulations of UC emulation

We present some alternative formulations of UC emulation (Definition 8).

Environments with non-binary outputs. Definition 8 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 8. We leave the proof as an exercise.

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 8. Again, we omit the proof. Note however that this equivalence does *not* hold for the case of closed environments, where the environment has no input, other than import value.

4.4.1 Emulation with respect to the dummy adversary

We show that Definition 8 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 messages generated by the environment to the specified recipients, 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 with the protocol. It thus leaves the simulator with very little "wiggle room."

More specifically, the dummy adversary, denoted $\mathcal{D}$, proceeds as follows. When activated with an input $(n, (m, id, c))$ from $\mathcal{E}$, where n is the import of the input, m is a message to be delivered, id is an identity, and c is a code for a party, $\mathcal{D}$ writes m on the backdoor tape of the ITI with identity id and code c, subject to the runtime limitations described below. If $id = 0$ then the target ITI is taken to be $\mathcal{E}$ itself. When activated with a message m on its backdoor tape, adversary $\mathcal{D}$ passes m as output to $\mathcal{E}$, along with the extended identity of the sender. Again, this is done subject to the runtime limitations below.

To make sure that $\mathcal{D}$ is polynomially bounded, we add the following mechanism. $\mathcal{D}$ keeps a variable ν which holds the total imports of all inputs received so far, minus the total lengths of all inputs and all incoming messages, minus the imports of all the messages that $\mathcal{D}$ delivered to protocol parties. If at any activation the variable ν holds a value that is smaller than the security

parameter k then $\mathcal{D}$ ends this activation without sending any message. With this mechanism in place, $\mathcal{D}$ can be implemented in linear time.

Definition 10 *Let π and ϕ be subroutine-respecting PPT protocols. We say that π* UC-emulates *protocol ϕ* with respect to the dummy adversary *if there exists a PPT adversary $\mathcal{S}$ such that for any balanced PPT environment $\mathcal{E}$ we have $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}} \approx \text{EXEC}_{\pi,\mathcal{D},\mathcal{E}}$.*

We show:

Claim 11 *Let π, ϕ be subroutine respecting PPT protocols. Then π UC-emulates ϕ according to Definition 8 if and only if π UC-emulates ϕ with respect to the dummy adversary.*

We remark that Claim 11 and its proof naturally extend to emulation with respect to ξ-identity-bounded environments, without change in the bounding predicate ξ.

Proof: Clearly if π UC-emulates ϕ according to Definition 8 then it UC-emulates ϕ with respect to the dummy adversary. 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 $\text{EXEC}_{\phi,\mathcal{S}_\mathcal{D},\mathcal{E}} \approx \text{EXEC}_{\pi,\mathcal{D},\mathcal{E}}$ for all $\mathcal{E}$.) We show that π UC-emulates ϕ according to Definition 8. For this purpose, given an adversary $\mathcal{A}$ we construct an adversary $\mathcal{S}$ such that $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}} \approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$ for all $\mathcal{E}$. Adversary $\mathcal{S}$ runs simulated instances of $\mathcal{A}$ and $\mathcal{S}_\mathcal{D}$. $\mathcal{S}$ also maintains a variable ν that will hold the current balance of runtime tokens for the dummy adversary. That is, ν will hold the total imports of all inputs given to $\mathcal{S}_\mathcal{D}$ so far, minus the total imports of all messages generated by $\mathcal{S}_\mathcal{D}$ so far, minus the import of the message that $\mathcal{S}_\mathcal{D}$ has been instructed to deliver in this activation. Then:

1. When $\mathcal{S}$ obtains input x with import n from $\mathcal{E}$, it operates as follows:
   (a) $\mathcal{S}$ locally activates $\mathcal{A}$ with input x and import n, and runs $\mathcal{A}$ till $\mathcal{A}$ completes its activation. (Note that $\mathcal{S}$ does not reall spend import here, since $\mathcal{A}$ is simulated within $\mathcal{S}$.)
   (b) Let m be the outgoing message that $\mathcal{A}$ generates in this activation, and let id, c be the identity and code of the target ITI of this message. If $id \neq 0$ then $\mathcal{S}$ activates $\mathcal{S}_\mathcal{D}$ with input (m, id, c) whose import is $\max\{n, k - \nu\}$, where k is the security parameter. (This guarantees that $\mathcal{S}_\mathcal{D}$ never runs out of runtime.) If $id = 0$ (i.e., the target ITI is the environment), then $\mathcal{S}$ writes this message to the subroutine-output tape of $\mathcal{E}$. If $\mathcal{A}$ generates no outgoing message in this activation, then $\mathcal{S}$ ends the activation with no output.
   (c) If $\mathcal{S}_\mathcal{D}$ was activated in this activation of $\mathcal{S}$, then $\mathcal{S}$ follows the instructions of $\mathcal{S}_\mathcal{D}$ regarding generating an outgoing message. That is, $\mathcal{S}$ generates the same outgoing message to the same recipient as $\mathcal{S}_\mathcal{D}$ does.
2. When $\mathcal{S}$ obtains a message on its backdoor tape, it operates as follows:
   (a) $\mathcal{S}$ activates $\mathcal{S}_\mathcal{D}$ with the incoming message on the backdoor tape, and runs $\mathcal{S}_\mathcal{D}$ until $\mathcal{S}_\mathcal{D}$ completes its activation.

(b) If in this activation, $\mathcal{S}_\mathcal{D}$ generates a message to be written to the backdoor tape of an ITI other than $\mathcal{E}$, then $\mathcal{S}$ follows the instructions of $\mathcal{S}_\mathcal{D}$ regarding writing this message.

(c) If the message s generated by $\mathcal{S}_\mathcal{D}$ is directed at $\mathcal{E}$, then $\mathcal{S}$ parses $s = (m, id, c)$ and activates $\mathcal{A}$ with message m from ITI id with code c (written on $\mathcal{A}$'s backdoor tape). Next $\mathcal{S}$ runs $\mathcal{A}$ till $\mathcal{A}$ completes its activation.

(d) If in this activation $\mathcal{A}$ generates an outgoing message to $\mathcal{E}$, then $\mathcal{S}$ generates this message to $\mathcal{E}$. If the message generated by $\mathcal{A}$ is aimed at another ITI, then $\mathcal{S}$ activates $\mathcal{S}_\mathcal{D}$ with input (m, id, c) where m is the message and (id, c) are the identity and code of the recipient ITI. $\mathcal{S}$ then follows the instructions of $\mathcal{S}_\mathcal{D}$ regarding generating an outgoing message. This input of $\mathcal{S}_\mathcal{D}$ has import $\max\{0, k - \nu\}$.

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

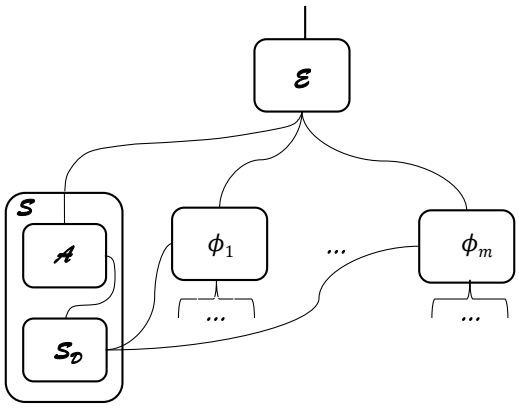

Figure 6: The operation of simulator $\mathcal{S}$ in the proof of Claim 11: Both $\mathcal{A}$ and $\mathcal{S}_\mathcal{D}$ are simulated internally by $\mathcal{S}$. The same structure represents also the operation of the shell adversary in the definition of black-box simulation (see Section 4.4.2).

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 (with some simulation overhead). When $\mathcal{S}$ has input with total import 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 $\mathcal{A}$. The runtime of the $\mathcal{S}_\mathcal{D}$ module is $p_{\mathcal{S}_\mathcal{D}}(n')$ where $p_{\mathcal{S}_\mathcal{D}}$ is the polynomial bounding the running time of $\mathcal{S}_D$ and n' is the total import of the inputs to $\mathcal{S}_\mathcal{D}$. In turn, n' is bounded by $\max\{n, n'\}$, where n' is the total import of the outgoing messages generated by $\mathcal{A}$. Since $max\{n, n'\} \leq p_\mathcal{A}(n)$, we have that the polynomial bounding the runtime of $\mathcal{S}$ is at most $p_\mathcal{S}(\cdot) = \mathcal{O}(p_{\mathcal{S}_\mathcal{D}}(p_\mathcal{A}(\cdot)))$ where $\mathcal{O}$ accounts for the simulation overhead (say, $\mathcal{O}$ is some linear function).

An alternative way to bound the runtime of $\mathcal{S}$ is to observe that without loss of generality we have $n' < p_\phi(n)$, where $p_\phi(\cdot)$ is the polynomial bounding the runtime of ϕ. This is the case since the overall import of inputs to the ITIs running ϕ is at most n (since $\mathcal{E}$ is balanced), and so additional communication beyond $p_\phi(n)$ can be truncated. Furthermore, the simulation overhead is at most

linear in the overall lengths of inputs and outputs of $\mathcal{A}$. It follows that:

$$p_{\mathcal{S}}(\cdot) = p_{\mathcal{A}}(\cdot) + \mathcal{O}(p_{\mathcal{S}_{\mathcal{D}}}(p_{\phi}(\cdot))). \tag{1}$$

We note that this fact is used in the proof of Claim 17 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 $\text{EXEC}_{\phi,\mathcal{S}_{\mathcal{D}},\mathcal{E}_{\mathcal{D}}} \not\approx \text{EXEC}_{\pi,\mathcal{D},\mathcal{E}_{\mathcal{D}}}$. Environment $\mathcal{E}_{\mathcal{D}}$ runs internally an interaction between simulated instances of $\mathcal{E}$ and $\mathcal{A}$. Similarly to $\mathcal{S}$, $\mathcal{E}_{\mathcal{D}}$ also maintains a variable ν that holds the current balance of runtime for the dummy adversary. That is, ν will hold the total imports of all inputs given to the external adversary so far, minus the total lengths of all inputs given to and all outputs received from the adversary so far, minus the imports of all the messages that the external adversary has been instructed to deliver to protocol parties. In addition:

1. When $\mathcal{E}$ generates an input x with import n to some ITI other than the adversary, $\mathcal{E}_{\mathcal{D}}$ does the same.
2. When $\mathcal{E}$ generates an input x with import n to its adversary, $\mathcal{E}_{\mathcal{D}}$ operates as follows:
   (a) $\mathcal{E}_{\mathcal{D}}$ activates $\mathcal{A}$ with input x and import n, and runs $\mathcal{A}$ till $\mathcal{A}$ completes its activation.
   (b) If in its activation $\mathcal{A}$ generates an output m to its environment, then $\mathcal{E}_{\mathcal{D}}$ activates $\mathcal{E}$ again with input m coming from the adversary. If $\mathcal{A}$ does not generate any outgoing message, then $\mathcal{E}_{\mathcal{D}}$ activates $\mathcal{E}$ again with no new incoming message. Else, Let m be the outgoing message that $\mathcal{A}$ generates in this activation, let id be the identity of the target ITI and let c be the code of the target ITI. Then, $\mathcal{E}_{\mathcal{D}}$ activates the external adversary with input (m, id, c) with import $\max\{n, k - \nu\}$, k being the security parameter. (This import guarantees that a dummy adversary will never run out of runtime tokens.)
3. When $\mathcal{E}_{\mathcal{D}}$ obtains, on its subroutine-output tape, an output value from an ITI other than the adversary, it forwards that output value to $\mathcal{E}$.
4. When $\mathcal{E}_{\mathcal{D}}$ obtains an incoming message (m, id, c) from the external adversary, it operates as follows:
   (a) $\mathcal{E}_{\mathcal{D}}$ activates $\mathcal{A}$ with incoming message m from ITI id with code c, and runs $\mathcal{A}$ till $\mathcal{A}$ completes its activation.
   (b) If in this activation $\mathcal{A}$ generates an outgoing message to its environment, then $\mathcal{E}_{\mathcal{D}}$ forwards this message to $\mathcal{E}$. If the message generated by $\mathcal{A}$ is aimed at another ITI, then $\mathcal{E}_{\mathcal{D}}$ activates its external adversary with input (m, id, c) where m is the message and id, c are the identity and code of the recipient ITI. This input message has import $\max\{0, k - \nu\}$.

We argue that environment $\mathcal{E}_{\mathcal{D}}$ is balanced. This is so since $\mathcal{E}$ is balanced, and at any point in time during the execution of $\mathcal{E}_{\mathcal{D}}$ the overall import that $\mathcal{E}_{\mathcal{D}}$ gives its external adversary is at least the overall import that $\mathcal{E}$ gives its own external adversary; Furthermore, the import that $\mathcal{E}_{\mathcal{D}}$ gives to each other ITI is at most the import that $\mathcal{E}$ gives this ITI.

It can also be verified that ensembles $\text{EXEC}_{\pi,\mathcal{D},\mathcal{E}_{\mathcal{D}}}$ and $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$ are identical. In particular, $\mathcal{D}$ never stops due to insufficient runtime.

Similarly, $\text{EXEC}_{\phi,\mathcal{S}_{\mathcal{D}},\mathcal{E}_{\mathcal{D}}}$ and $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}}$ are identical. In particular, the views of $\mathcal{S}_{\mathcal{D}}$ in the two experiments, including the import values, are identically distributed in the two experiments. □

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 somewhat 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.[13]

4.4.2 Emulation with respect to black-box simulation

Another alternative formulation of Definition 8 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 only imposes restrictions on the operation of the simulator. Specifically, an adversary $\mathcal{S}$ is called a *black-box simulator* if it consists of a main program or ITM (which we call a shell) and a subroutine α whose program is another ITM. Upon activation of $\mathcal{S}$, the shell is activated. The shell invokes and activates α on arbitrary inputs and obtains the outputs of α, but does not have access to the program or internal state of α. Furthermore α does not have access to the outgoing message tape of $\mathcal{S}$. We let $\mathcal{S}^{\mathcal{A}}$ denote the black-box simulator $\mathcal{S}$ when the code of the subroutine is $\mathcal{A}$.

Definition 12 *Say that protocol π* UC-emulates *protocol ϕ* with black-box simulation *if there exists a PPT adversary $\mathcal{S}$ such that for any PPT adversary $\mathcal{A}$ and any balanced PPT environment $\mathcal{E}$ we have* $\text{EXEC}_{\phi,\mathcal{S}^{\mathcal{A}},\mathcal{E}} \approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$.

We observe that UC-emulation with black-box simulation is equivalent as plain UC-emulation:

Claim 13 *Let π, ϕ be PPT multiparty protocols. Then π UC-emulates ϕ according to Definition 8 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, observe that simulator $\mathcal{S}$ in the proof of Claim 11 is in fact a black-box simulator, where the shell consists of the main program of $\mathcal{S}$ together with $\mathcal{S}_{\mathcal{D}}$. See Figure 6. □

[13] 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, since in this definition the environment is inherently un-balanced. See discussion in Footnote 10.

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 the black-box simulator controls also the random tape of $\mathcal{A}$ and can thus effectively "rewind" and "reset" $\mathcal{A}$ to arbitrary previous states in its execution. In contrast, here the communication between $\mathcal{S}$ and $\mathcal{A}$ is restricted to obtaining outputs of complete executions with potentially hidden randomness. Still, the present definition is equivalent to the plain (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 black-box simulation is not equivalent to the basic definition, due to different formalizations of probabilistic polynomial time.

4.4.3 Letting the simulator depend on the environment

Consider a variant of Definition 8, 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 emulation with respect to specialized simulators. A priori, it may appear that emulation with respect to specialized simulators is not sufficiently strong so as to provide the guarantees promised by UC security. Indeed, jumping ahead, we note that the proof of the UC theorem crucially uses the fact that the same simulator works for all environments. However it turns out that, in the present framework, emulation with respect to specialized simulators is actually equivalent to full-fledged UC security:

Claim 14 *A protocol π UC-emulates protocol ϕ according to Definition 8 if and only if it UC-emulates ϕ with respect to specialized simulators.*

Proof: Clearly, if π UC-emulates ϕ as in Definition 8 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 $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}} \approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$. Consider the "universal environment" $\mathcal{E}_u$ which expects its input to consist of $(\langle\mathcal{E}\rangle, z, t)$, 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 $\mathcal{E}$. (t is also the import of the input.) 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 $\mathcal{S}$ such that $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}_u} \approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}_u}$. We claim that $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}} \approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$ for *any* balanced PPT environment $\mathcal{E}$. To see this, fix a PPT machine $\mathcal{E}$ as in Definition 5, 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 $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}}(k,z)$ is identical to the distribution $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}_u}(k,z_u)$, where $z_u = (\langle\mathcal{E}\rangle, z, |z|^c)$. Similarly, the distribution $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(k,z)$ is identical to the distribution $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}_u}(k,z_u)$. Consequently, for any $d \in \mathbf{N}$ we have:

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

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, $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}} \approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$. (Notice that if $|z_u|$ were not polynomial in k then the last derivation would not hold.) □

Remark: Claim 14 is an extension of the equivalence argument for the case of computationally unbounded environment and adversaries, discussed in [C00]. A crucial element in the proof of this claim is the fact that the class of allowed environments 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]).

4.5 Some Variants of UC emulation

Next we present some variants of the basic notion of UC emulation, specifically statistical emulation, emulation with respect to closed environments, and two more quantitative notions of UC emulation. We also make some observations regarding the quantitative notions.

On statistical and perfect emulation. Definition 8 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 $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}}$ and $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$ are required to be identical then we say that π perfectly UC-emulates ϕ. Another variant allows $\mathcal{S}$ to have unlimited computational power, 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 8 considers environments that take input (of some polynomial length) that was generated in an arbitrary way, perhaps not even recursively. This input represents some initial joint state of the system and the adversary. Alternatively, one may choose to consider only "closed environments", namely environment that do not receive meaningful external input. Here the notion of security considers only environments whose external input contains no information other than its import. Such environments would choose the inputs of the parties using some internal stochastic process. We note that Claim 14 does *not* hold for closed environments. Indeed, jumping ahead, it can be seen that: (a) As long as there is a single simulator that works for all environments, the UC theorem holds even with respect to closed environments. (b) The UC theorem does not hold with respect to closed environments and specialized simulators.

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 5), and that a functional is a function from functions to functions. Then:

Definition 15 *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 (ϵ, 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 15 abandons the asymptotic framework and instead concentrates on a specific value of the security parameter k:

Definition 16 *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 16 still quantifies over all PPT environments and adversaries of all polynomial complexities. One can potentially formulate a definition that parameterizes also the run-times of the environment, adversary and simulator. That is, this weaker definition would quantify only over environments and adversaries that have specific complexity. It should be noted, however, that such a definition would be considerably weaker than Definition 16, 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 16 bounds the specified parameters for any arbitrarily complex environment and adversary.

Indeed, with such a fully parametric definition, the universal composition theorem, the dummy-adversary and black-box-simulation theorems would need to account for the appropriate quantitative degradation in the simulation overhead and the emulation slack.

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. That is:

Claim 17 *Let π and ϕ be protocols such that π UC-emulates ϕ with emulation slack ϵ and simulation overhead g, as in Definition 15. 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.

Proof. The claim follows from the proof of Claim 11. Indeed, the proof of Claim 11 shows how to construct, for any adversary $\mathcal{A}$, a valid simulator 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 polynomial in the complexities of π and ϕ (see Equation (1)). □

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 17 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.

5 Universal composition

This section presents the universal composition operation, and then states and proves the universal composition theorem, with respect to the definition of UC-emulation as formulated in Section 4. 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.

Both the composition operation and the proof of the composition theorem extend those in Section 2.3 so as to hold in the present model of execution. The extensions are significant; in particular, here the composition operation necessarily has to replace multiple instances of the subroutine protocol with another one, while making sure that the change is "seamless" from the point of view of both the calling protocol and the subroutine protocol. We elaborate within.

5.1 The universal composition operation and theorem

We present the composition operation in terms of an operator on protocols. Recall that this operator, called the universal composition operator is aimed to generalize the simple substitution operation in Section 2.3, which in turn generalizes the natural "subroutine substitution" operation on sequential algorithms.

Formally, the operator, denoted 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 $\rho^{\phi\to\pi} = \text{UC}(\rho, \pi, \phi)$ follows the instructions of ρ with two exceptions:

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

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

See a graphical depiction in Figure 4 on page 19. In other words, the program of $\rho^{\phi\to\pi}$ can be thought of as consisting of a "body" part that's identical to ρ, and a separate "shell" part that performs the translation between calling ϕ and calling π. It is stressed that the "body" part is not being made aware of the fact that the translations occur. The replacement is only applied when the code of target ITI is ϕ and the forced-write flag is set (in case 1) or when the code of source ITI is ϕ and the reveal sender code flag is set (in case 2). This is so even if there are other messages with the same session ID for which the replacement was applied. See more detailed account of the partitioning of protocols to "body" and "shell" in Section 6.1.[14]

Observe that if protocols ρ, ϕ, and π are PPT then $\rho^{\phi\to\pi}$ is PPT (with a bounding polynomial which is essentially the maximum of the individual bounding polynomials[15]).

When protocol ϕ is the ideal protocol $\text{IDEAL}_{\mathcal{F}}$ for some ideal functionality $\mathcal{F}$, we denote the composed protocol by $\rho^{\mathcal{F}\to\pi}$.

Comparison with the UC operation of Section 2.3. We note that the UC operation as defined here is different than the variant defined in Section 2.3: There, only a single instance of ϕ was replaced with π. Here is it inherently the case that *all* top-level instances of ϕ are being replaced. Indeed, in the present model it is not clear how to define a single-instance-replacement operation as a general operator on protocols; this is so since different ITIs in a system might have different views of the execution.

Compliant and subroutine-exposing protocols. In its general form, the UC theorem will state that protocol $\rho^{\phi\to\pi}$ UC-realized the original protocol ρ. However, to make this theorem work in our model we will need to impose a number of restrictions on ρ, π and ϕ. We start with the requirements from the calling protocol, ρ. Let ξ be a set of (or, predicate on) extended identities. We say that ρ is (π, ϕ, ξ)-compliant if:

- All external-writes made by parties and subsidiaries of ρ, where the target tape is the input tape, use the forced-write mode.

- No two external-write instructions, made by parties or subsidiaries of an instance of ρ, where one instruction has target code π, and the other instruction has target code ϕ, have the same target session ID.

[14] Alternatively, one could define the composition operation 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. While technically equivalent, we find the present formulation, where the protocol determines the code run by its subroutines, simpler and more intuitively appealing.

[15] Actually, $\rho^{\phi\to\pi}$ may make δ times more steps than ρ, where δ is the difference between the description length π and that of ϕ. This is so since $\rho^{\phi\to\pi}$ needs to write the code π whenever invoking a new main party of an instance of π. Notice however that in our model programs have constant description length, irrespective of the security parameter.

- All messages received on the subroutine-output tapes of parties and subsidiaries of ρ are expected to have reveal-sender-id flag on; other subroutine-outputs are ignored.
- The extended identities of all the ITIs in any extended instance of ρ, that have subroutines with code either π or ϕ, satisfy the predicate ξ.

We proceed to specify the requirements from protocols π and ϕ. Here we require that the subroutine structure of π and ϕ is public, in the sense that there should be a mechanism for the adversary to tell, given an extended identity of an ITI, whether this ITI is currently a party or a subroutine of a given instance of π (or ϕ).

For sake of concreteness we set the following mechanism for implementing this requirement. Say that protocol π is subroutine exposing if for each instance s of π there exists a special "directory" ITI $(\pi, s, \top)$ (where $\top$ is a special party ID that means "directory") that contains the list of the extended identities of all parties and subsidiaries of this instance of π, and returns this list to the adversary upon request. Each ITI that is a party or subsidiary of this instance notifies the directory ITI of its extended identity immediately upon its invocation, and also of each new subroutine ITI before invoking. When notified by an ITI M that it has been invoked, the directory ITI adds M to its database if M is a main party of session s, or if some ITI in the database invoked it. This way, the list contains the identities of all parties or subsidiaries of instance s, and only these identities. We note that a similar mechanism is used to facilitate the definition of party-corruption operations; see more details in Section 6.1.[16]

Theorem statement. We are now ready to state the composition theorem. First we state a general theorem, to be followed by two corollaries. A more quantitative statement of the UC theorem is discussed in Section 5.3.

Theorem 18 (Universal composition: General statement) *Let ρ, π, ϕ be PPT protocols and let ξ be a PPT predicate, such that ρ is (π, ϕ, ξ)-compliant, both ϕ and π are subroutine exposing, and π UC-emulates ϕ with respect to ξ-identity-bounded environments. Then protocol $\rho^{\phi \to \pi}$ UC-emulates protocol ρ.*

As a special case, we have:

Corollary 19 (Universal composition: using ideal functionalities) *Let ρ, π be PPT protocols, $\mathcal{F}$ be a PPT ideal functionality, and ξ be a PPT predicate, such that ρ is $(\pi, \text{IDEAL}_{\mathcal{F}}, \xi)$-compliant, π is subroutine exposing, and π UC-realizes $\mathcal{F}$ with respect to ξ-identity-bounded environments. Then protocol $\rho^{\phi \to \pi}$ 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 $\rho^{\mathcal{F} \to \pi}$ securely realizes $\mathcal{G}$.

[16] An alternative method for making sure that protocols are subroutine exposing, used in [HS11], mandates a hierarchical tree-like subroutine structure for protocol invocations, and furthermore requires that the hierarchical structure is represented in the session IDs. We note that this convention is sometimes overly restrictive, and also does not always suffice.

Corollary 20 (Universal composition: Realizing ideal functionalities) *Let $\mathcal{F}, \mathcal{G}$ be ideal functionalities such that $\mathcal{F}$ is PPT. Let ρ be a subroutine exposing, $(\pi, \text{IDEAL}_F, \xi)$-compliant protocol that UC-realizes $\mathcal{G}$ with respect to ξ'-identity-bounded environments, and let π be a subroutine exposing protocol that securely realizes $\mathcal{F}$ with respect to ξ-identity-bounded environments. Then the composed protocol $\rho^{\mathcal{F}\to\pi}$ securely realizes $\mathcal{G}$ with respect to ξ'-identity-bounded environments.*

Proof: Let $\mathcal{A}$ be an adversary that interacts with parties running $\rho^{\mathcal{F}\to\pi}$. Theorem 18 guarantees that there exists an adversary $\mathcal{A}'$ such that $\text{EXEC}_{\rho,\mathcal{A}',\mathcal{E}} \approx \text{EXEC}_{\rho^{\mathcal{F}\to\pi},\mathcal{A},\mathcal{E}}$ for any environment $\mathcal{E}$. Since ρ UC-realizes $\mathcal{G}$ respect to ξ'-identity-bounded environments, there exists a simulator $\mathcal{S}$ such that $\text{EXEC}_{\text{IDEAL}_\mathcal{G},\mathcal{S},\mathcal{E}} \approx \text{EXEC}_{\rho,,\mathcal{A}_\mathcal{F},\mathcal{E}}$ for any ξ'-identity-bounded $\mathcal{E}$. Using the transitivity of indistinguishability of ensembles we obtain that $\text{EXEC}_{\text{IDEAL}_\mathcal{G},\rho^{\mathcal{F}\to\pi},\mathcal{S},\mathcal{E}} \approx \text{EXEC}_{\rho^{\mathcal{F}\to\pi},\mathcal{A},\mathcal{E}}$ for any ξ'-identity-bounded environment $\mathcal{E}$. □

5.2 Proof of Theorem 18

We start with an outline of the proof, in Section 5.2.1. The full proof appears in Section 5.2.2.

We note that the proof here is significantly more complex than that of Section 2.3. One source of complication is the need to handle replacing multiple instances of ϕ by instances of π. (This seems inherent in our model; indeed it is not clear how to represent the operation of replacing some of the instances of ϕ but not others.) Another source of complication is that the composite simulator $\mathcal{S}$ needs to be able to identify the types of input and incoming backdoor messages; in the model of Section 2.3 this task is trivial.

5.2.1 Proof outline

The proof uses the formulation of emulation with respect to dummy adversaries (see Claim 11). While equivalent to the standard definition, this formulation considerably simplifies the proof.

Let ρ, ϕ, π and ξ be such that π UC-emulates ϕ with respect to ξ-identity-bounded environments and ρ is (π, ϕ, ξ)-compliant, and let $\rho^\pi = \rho^{\phi\to\pi} = \text{UC}(\rho, \pi, \phi)$ be the composed protocol. We wish to construct an adversary $\mathcal{S}$ so that no ξ-identity-bounded $\mathcal{E}$ 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

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

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 ξ-identity-bounded environment $\mathcal{E}_\pi$ we have:

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

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).

Construction of $\mathcal{S}$. Simulator $\mathcal{S}$ operates as follows. Recall that $\mathcal{E}$ expects to interact with parties running ρ. The idea is to separate the interaction between $\mathcal{E}$ and the parties into several parts. To mimic the sending of backdoor messages to the parties and subsidiaries of each session of

π, and the receiving of backdoor messages from them, $\mathcal{S}$ runs an instance of the simulator $\mathcal{S}_\pi$. To mimic the sending and receiving of backdoor messages to/from the rest of the ITIs in the system, $\mathcal{S}$ interacts directly with these parties, mimicking the dummy adversary. (Recall that these parties are the main parties of ρ and their subsidiaries which are not parties or subsidiaries of a session of π. We call these ITIs side-players.)

More specifically, recall that $\mathcal{E}$ delivers, via the dummy adversary, backdoor messages to the parties of ρ, to the parties of all sessions of π, and to all their subsidiaries. In addition, $\mathcal{E}$ expects to receive all backdoor messages sent by these ITIs to the dummy adversary.

To address these expectations, $\mathcal{S}$ internally runs an instance of the simulator $\mathcal{S}_\pi$ for each session of ϕ in the system it interacts with. When activated by $\mathcal{E}$ with message m to be sent to ITI M, $\mathcal{S}$ first finds out if M is to be treated as a side-party, or else it should be handled by one of the instances of $\mathcal{S}_\pi$. This is done as follows:

If M is a main party of one of the sessions of π, then the answer is clear: m is to be handled by the corresponding instance of $\mathcal{S}_\pi$ (and if no such instance of $\mathcal{S}_\pi$ exists then one is created.) Else, $\mathcal{S}$ generates an input to each one of the instanced of $\mathcal{S}_\pi$ to check with the directory ITI of this session of π whether M is a member of that extended session. If one of the instances of $\mathcal{S}_\pi$ responds positively, then the input is to be handled by this instance.

In this case, $\mathcal{S}$ generates an input to the said instance of $\mathcal{S}_\pi$ with an instruction to deliver backdoor message m to ITI M, and continues to follow the instructions of this instance of $\mathcal{S}_\pi$ for the rest of the activation. Here $\mathcal{S}$ makes sure that the overall import of the inputs to $\mathcal{S}_\pi$ equals the overall import of the inputs to $\mathcal{S}$ so far.[17]

If none of the instances of $\mathcal{S}_\pi$ answers positively, then $\mathcal{S}$ treats M as a side party, namely the backdoor message m is delivered to ITI M. Note that since π is subroutine respecting, the situation where M is a member of two extended instances of π does not occur.

When activated with a backdoor message m sent by an ITI M, $\mathcal{S}$ again first finds out if M is to be treated as a side-party, or else it should be handled by one of the instances of $\mathcal{S}_\pi$. A similar mechanism is used: If M is a main party of one of the sessions of ϕ, then m is handed to the corresponding instance of $\mathcal{S}_\pi$, and if no such instance of $\mathcal{S}_\pi$ exist then one is created.

If M is not a main party of a session of ϕ, then $\mathcal{S}$ checks with the directory ITIs of all current top-level sessions of ϕ whether M is a member of that extended session. If one of them respond positively, then $\mathcal{S}$ activates this instance of $\mathcal{S}_\pi$ with an incoming backdoor message m from M, and continues to follow the instructions of this instance of $\mathcal{S}_\pi$ for the rest of the activation. If none of the directory ITIs responds positively, then $\mathcal{S}$ treats M as a side party, namely the message is forwarded to $\mathcal{E}$. Since ϕ is subroutine respecting, the situation where M is a member of two top-level extended instances of ϕ never occurs.

If instances of ϕ recursively use other instances of ϕ as subroutines, only the "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. Other instances of ϕ, if they exist, will be "handled" by the instance of $\mathcal{S}_\pi$ associated with the corresponding top-level instance of ϕ.

Figure 7 presents a graphical depiction of the operation of $\mathcal{S}$. A more complete description of the simulator is deferred to the detailed proof.

[17] Having the import of each $\mathcal{S}_\pi$ equal the import of $\mathcal{S}$ is done to make sure that the constructed environment $\mathcal{E}_\pi$ is balanced. This means that the polynomial bounding runtime of $\mathcal{S}$ should be t times the polynomial bounding the runtime of $\mathcal{S}_\pi$, where t is the maximum number of sessions of ϕ generated by ρ.

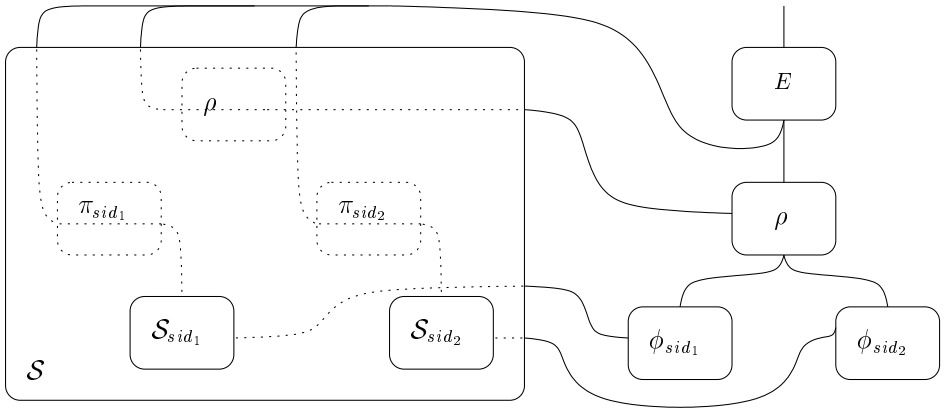

Figure 7: The operation of $\mathcal{S}$ in the proof of the composition theorem. Inputs from $\mathcal{E}$ that represent backdoor messages directed at the ITIs which are not part of an extended session of π or ϕ are forwarded to the actual recipients. Inputs directed at a session of π are re-directed to the corresponding instance of $\mathcal{S}$. Backdoor messages from an instance of $\mathcal{S}$ are directed to the corresponding actual session of ϕ. For graphical clarity we use a single box to represent a session of a multi-party protocol.

Analysis of $\mathcal{S}$. The validity of $\mathcal{S}$ is reduced to the validity of $\mathcal{S}_\pi$ via a traditional hybrid argument. However, while the basic logic of the argument is standard, applying the argument in our setting requires some care.

We sketch this argument. Let t be an upper bound on the number of sessions of π that are invoked in this interaction. Ideally, we would have liked the argument to proceed as follows: For $0 \leq l \leq t$ let ρ_l denote the protocol where the first l sessions of ϕ remain unchanged, whereas the rest of the sessions of ϕ are replaced with sessions of π.

However, it is not clear how such hybrid protocol ρ_l would look like; in particular, the ITIs running ρ_l might not know which is the lth session to be (globally) invoked. So we leave the protocol $\rho^{\phi \to \pi}$ as is, and instead define $t+1$ hybrid simulators $\mathcal{S}^0, ..., \mathcal{S}^t$, and $t+1$ hybrid models of execution (by way of defining $t+1$ control functions). The lth control function will replace the $t-l+1$ last top-level sessions of π back to being sessions of ϕ. Similarly, the simulator $\mathcal{S}^l$ will invoke copies of $\mathcal{S}_\pi$ only for the first l sessions of ρ.

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 $\rho^{\phi \to \pi}$. By construction, the former interaction is identical to an interaction with $\mathcal{S}^0$ in the 0-th hybrid model, whereas the latter interaction is identical to an interaction with $\mathcal{S}^t$ in the t-th hybrid model. It follows that, for a random $l \leftarrow \{1..t\}$, $\mathcal{E}$ distinguishes with probability ϵ/t between an interaction with $\mathcal{S}^l$ and $\rho^{\phi \to \pi}$ in the lth hybrid model, and an interaction with $\mathcal{S}^{l-1}$ and $\rho^{\phi \to \pi}$ in the $(l-1)$st hybrid model.

We then construct an environment $\mathcal{E}_\pi$ that uses $\mathcal{E}$ to distinguish with probability ϵ/t between an interaction with $\mathcal{D}$ and parties running a single session 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 $\rho^{\phi \to \pi}$, but with the following exception. $\mathcal{E}_\pi$ uses its actual interaction (which is either with ϕ and $\mathcal{S}_\pi$, or with ρ and $\mathcal{D}$) to replace the parts of the simulated execution that have to do with the interaction with the lth session of ϕ, denoted ϕ_l.

A bit more specifically, whenever some simulated side-player passes an input x to a party or

subsidiary of ϕ_l (i.e. the lth session of ϕ), the environment $\mathcal{E}_\pi$ passes input x to the corresponding ITI in the external execution. Outputs generated by an actual party running π are treated like outputs from ϕ_l to the corresponding simulated side-player.

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 $\mathcal{E}_\pi$.

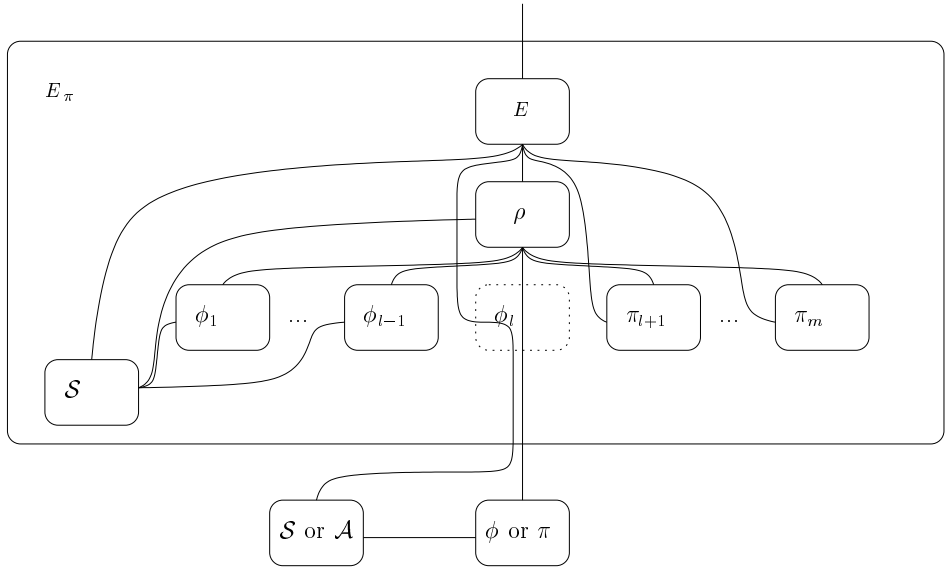

Figure 8: The operation of $\mathcal{E}_\pi$. An interaction of $\mathcal{E}$ with π is simulated, so that the first $l-1$ sessions of ϕ remain unchanged, the lth session is mapped to the external execution, and the remaining sessions of ϕ are replaced by sessions of π. For graphical clarity we use a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

Since $\mathcal{E}$ is balanced, then so is $\mathcal{E}_\pi$. (This is so since the overall import of inputs to the external instance of $\mathcal{S}_\pi$ equals the overall import to $\mathcal{S}$.) Furthermore, since ρ is (π, ϕ, ξ)-compliant then $\mathcal{E}_\pi$ is ξ-identity-bounded. 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 $\mathcal{S}^l$ and $\rho^{\phi\to\pi}$ in the lth hybrid model. 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 $\mathcal{S}^{l-1}$ and $\rho^{\phi\to\pi}$ in the $(l-1)$st hybrid model. The fact that π and ϕ are subroutine respecting is crucial for this argument, since it guarantees that there are no "side interactions", or "direct information flow" between ITIs that belong to different extended instances of π or ϕ.

5.2.2 A detailed proof

We proceed with a detailed proof of Theorem 18, substantiating the above outline.

Adversary $\mathcal{S}$

Adversary $\mathcal{S}$ locally runs multiple instances of $\mathcal{S}_\pi$ as described below.

1. When activated with input (m, M) (coming from the environment $\mathcal{E}$), where m is a message, and $M = (id, c)$ is an ITI with identity $id = (sid, pid)$ and code c, do:
   (a) Activate each existing instance s of $\mathcal{S}_\pi$ with an instruction, coming as input from the environment, to ask the directory ITI of session s of π whether M appears in its database. (Whenever an instance of $\mathcal{S}_\pi$ sends a query to the directory ITI of an instance of ϕ, $\mathcal{S}$ sends the query to the same directory ITI, and forwards the response to that instance of $\mathcal{S}_\pi$.)
   (b) If the directory of instance s of $\mathcal{S}_\pi$ responds positively then activate this instance of $\mathcal{S}_\pi$ with input (m, M), and follow the instructions of this instance of $\mathcal{S}_\pi$ for the rest of this activation — with 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 $\mathcal{S}$. The import of the input (m, M) is set to 2ι, where ι is the overall import of the inputs received by $\mathcal{S}$ so far, minus the overall import given to $\mathcal{S}_\pi$ so far. (Jumping ahead, we note that the reason for doubling ι is to make sure that the distinguishing environment $\mathcal{E}_\pi$ remains balanced.)
   (c) If all instances of $\mathcal{S}_\pi$ answer negatively and M is a main party of an instance s' of π, then invoke a new instance of $\mathcal{S}_\pi$ with session ID s', and activate this instance of $\mathcal{S}_\pi$ and follow it instructions as in Step 1b.
   (d) Else deliver the backdoor message m to ITI M, subject to the runtime restrictions of the dummy adversary.
2. When activated with backdoor message m from an ITI $M = (id, c)$,with $id = (sid, pid)$ do:
   (a) Query the directory ITI of each existing top-level instance s of ϕ whether M is in its directory. If the directory ITI of any instance s responds positively then activate instance s of $\mathcal{S}_\pi$ with incoming backdoor message m from ITI M, and follow the instructions of this instance of $\mathcal{S}_\pi$ for the rest of this activation, with the same exception that $\mathcal{S}$ mimics the time bounds of a dummy adversary.
   (b) If all directory ITIs answer negatively and M is a main party of an instance s' of ϕ, then invoke a new instance of $\mathcal{S}_\pi$, with session ID s', activate this instance of $\mathcal{S}_\pi$, and follow its instructions as in Step 2a.
   (c) Else forward (M, m) to $\mathcal{E}$, subject to the runtime restrictions of the dummy adversary.

Figure 9: The adversary for protocol ρ.

Construction of $\mathcal{S}$. Let ρ, ϕ, π and ξ be such that π UC-emulates ϕ with respect to ξ-identity-bounded environments and ρ is (π, ϕ, ξ)-compliant, and let $\rho^\pi = \rho^{\phi \to \pi} = \text{UC}(\rho, \pi, \phi)$ be the composed protocol. Let $\mathcal{S}_\pi$ be an adversary such that $\text{EXEC}_{\phi, \mathcal{S}_\pi, \mathcal{E}_\pi} \approx \text{EXEC}_{\mathcal{D}, \pi, \mathcal{E}_\pi}$ holds for any ξ-identity-bounded environment $\mathcal{E}_\pi$. Adversary $\mathcal{S}$ uses $\mathcal{S}_\pi$ and is presented in Figure 9.

Validity of $\mathcal{S}$. First, note that $\mathcal{S}$ is PPT. The polynomial $p(\cdot)$ bounding the running time of $\mathcal{S}$ can be set to $2t$ times the polynomial bounding the running time of $\mathcal{S}_\pi$, where t is a bound on the number of sessions of ϕ invoked by ρ. (Note that $t \leq n$, where n is the import of the input to $\mathcal{S}$.

This is so since $\mathcal{E}$ is balanced.)[18]

Now, assume that there exists a balanced environment machine $\mathcal{E}$ that violates the validity of $\mathcal{S}$ (that is, $\mathcal{E}$ violates Equation (2)). We construct a balanced ξ-identity-bounded 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

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

We show that

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

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

Towards constructing $\mathcal{E}_\pi$: The hybrid models and simulators. In preparation to constructing $\mathcal{E}_\pi$, we define the following distributions and variants of $\mathcal{S}$. 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 sessions of ϕ within ρ in this execution. (Say that a session of protocol π in an execution is top-level if it is not a subsidiary of any other session of π in that execution. The bound t is used in the analysis only. The parties need not be aware of t. Also, since $\mathcal{E}$ is PPT, t is polynomial in $k,|z|$.)

For $0 \leq l \leq t$, let the l-hybrid model for running protocol $\rho^{\phi\to\pi}$ 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:

The external-write requests to input tapes of the main parties of the first l top-level sessions of π to be invoked within the test session of $\rho^{\phi\to\pi}$ are redirected (back) to the corresponding sessions of ϕ. The external-write requests to the input tapes of the main parties of all other sessions of π are treated as usual. That is, let sid_i denote the SID of the ith top-level session of π to be invoked within the test instance of $\rho^{\phi\to\pi}$; then, given an external-write request made by some ITI to the input tape of ITI (π, id) where $id = (sid_i, pid)$ for $i \leq l$ and some pid, the control function writes the requested value to the input tape of ITI $(\phi, (sid_i, pid))$. If no such ITI exists then one is invoked. It is stressed that these modifications apply to external-write requests by *any* ITI, including ITIs that participate in sessions of ϕ and π, as well as subsidiaries thereof.

[18] The factor-t increase in the complexity of $\mathcal{S}$ results from the fact that the import that $\mathcal{S}$ gives each instance of $\mathcal{S}_\pi$ is comparable to the entire import of $\mathcal{S}$. This is done to account for the fact that our model allows different instances of ϕ to have very different imports, and $\mathcal{S}$ does not know the import of each instance. At the same time, the view of each instance of $\mathcal{S}_\pi$ should be consistent with an execution where its environment is balanced. We thus resolve this issue by setting the import of each instance of $\mathcal{S}_\pi$ to the maximal possible value, namely n. The additional factor of 2 will be needed to guarantee that environment $\mathcal{E}_\pi$, defined later, remains balanced. (Essentially, this factor accounts for the fact that the parties of the calling protocol ρ can obtain additional import via the backdoor messages they obtain from the adversary and so the import given by ρ to each individual instance of ϕ can be larger than the overall import that ρ received from its environment.)

In more restricted settings, where the imports given to the instances of ρ are known in advance, or alternatively where all instances of ϕ have roughly equal imports, or alternatively where the runtime of $\mathcal{S}_\pi$ depends only on the import of the parties running ϕ, and where in addition the import of the instances of ϕ is not larger than the the import of ρ, the polynomial bounding the complexity of $\mathcal{S}$ becomes the maximum of the polynomials bounding the runtimes of ρ, $\rho^{\phi\to\pi}$, and $\mathcal{S}_\pi$. Here the convention that the import is represented in binary rather than in unary becomes key.

Similarly, whenever $(\phi, (sid_i, pid))$ requests to pass output to some ITI μ, the control function changes the code of the sending ITI, as written on the subroutine-output tape of μ, to be π.

Let $\text{EXEC}^{l}_{\rho^{\phi\to\pi},\mathcal{A},\mathcal{E}}(k,z)$ denote the output of this system of ITMs on input z and security parameter k for the environment $\mathcal{E}$.

For each $0 \le l < t$, we also define a hybrid simulator $\mathcal{S}^l$. The hybrid simulator $\mathcal{S}^l$ is identical to $\mathcal{S}$, except that it invokes only up to l instances of $\mathcal{S}_\pi$. That is, in Steps (1c) and (2b) of Figure 9, if l instances of $\mathcal{S}_\pi$ are already active no new instance of $\mathcal{S}_\pi$ is activated, and the input (resp., incoming message) is treated as in Step (1d) (respectively, (2c)).

We observe that the output of $\mathcal{E}$ from an interaction with $\rho^{\phi\to\pi}$ and $\mathcal{S}^0$ in the 0-hybrid model is distributed identically to the output of $\mathcal{E}$ from an interaction with $\rho^{\phi\to\pi}$ in the standard model of computation, i.e. $\text{EXEC}^{0}_{\rho^{\phi\to\pi},\mathcal{S}^0,\mathcal{E}} = \text{EXEC}_{\rho^{\phi\to\pi},\mathcal{D},\mathcal{E}}$. Similarly, the output of $\mathcal{E}$ from an interaction with $\rho^{\phi\to\pi}$ and $\mathcal{S}^t$ in the t-hybrid model is distributed identically to the output of $\mathcal{E}$ from an interaction with ρ and $\mathcal{S}$ in the standard model of computation, i.e. $\text{EXEC}^{t}_{\rho,\mathcal{S}^t,\mathcal{E}} = \text{EXEC}_{\rho,\mathcal{S},\mathcal{E}}$.

Consequently, Inequality (4) can be rewritten as:

$$\text{EXEC}^{t}_{\rho^{\phi\to\pi},\mathcal{S}^t,\mathcal{E}}(k,z) - \text{EXEC}^{0}_{\rho^{\phi\to\pi},\mathcal{S},\mathcal{E}}(k,z) \ge \epsilon. \tag{6}$$

Construction and analysis of $\mathcal{E}_\pi$. Environment $\mathcal{E}_\pi$ 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, $\rho^{\phi\to\pi}$, and $\mathcal{S}$.) Also, since ρ is (π,ϕ,ξ)-compliant we have that $\mathcal{E}_\pi$ is ξ-identity-bounded.

Furthermore, we argue that $\mathcal{E}_\pi$ is balanced. This is so since $\mathcal{E}$ is balanced, and at any point during the execution we have that: (a) The overall import I_0 that $\mathcal{E}_\pi$ gave to the external adversary so far is at least twice the import I_1 that $\mathcal{E}$ gave its adversary so far, and (b) The overall import, I_2, that the main parties of any top-level session of π or ϕ receive in an execution of $\mathcal{E}_\pi$ is at most the overall import I_3 that the main parties of the test session of ρ receive from $\mathcal{E}$, plus the import I_4 that the parties of ρ received from $\mathcal{A}$ so far (via the backdoor messages). However the overall import received from $\mathcal{A}$ is bounded by the import that $\mathcal{E}$ gave its adversary so far, namely $I_4 \le I_1$, and since $\mathcal{E}$ is balanced we have $I_3 \le I_1$. Thus we have $I_2 \le I_3 + I_4 \le 2I_1 \le I_0$

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

$$\text{EXEC}_{\pi,\mathcal{D},\mathcal{E}^l_\pi}(k,z) = \text{EXEC}^{l-1}_{\rho^{\phi\to\pi},\mathcal{S}^{l-1},\mathcal{E}}(k,z) \tag{7}$$

and

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

Equations (7) and (8) follow from inspection of $\mathcal{E}_\pi$ and $\mathcal{S}$. Indeed, 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 $\rho^{\phi\to\pi}$ and $\mathcal{S}^{l-1}$ in the $(l-1)$-hybrid model. In particular, for any fixed value of the random choices, the messages that are directed by $\mathcal{E}^l_\pi$ to the external adversary in the left execution of (7) are exactly the messages delivered by $\mathcal{S}^{l-1}$ to the parties and subsidiaries of the session of π with session ID sid_l in the right execution of (7). The other messages that are sent by $\mathcal{E}^l_\pi$ in the left execution of (7) are exactly those messages that $\mathcal{E}$ sends to the parties and subsidiaries

Environment $\mathcal{E}_\pi$

Environment $\mathcal{E}_\pi$ proceeds as follows, given a value k for the security parameter, and input z. The goal is to distinguish between (a) the case where the test session runs π and the adversary is the dummy adversary, and (b) the case where the test session runs ϕ and the adversary is $\mathcal{S}_\pi$.
We first present a procedure called Simulate(). Next we describe the main program of $\mathcal{E}_\pi$.

Procedure Simulate(σ, l)

1. Expect the parameter σ to contain a global state of a system of ITMs representing an execution of protocol $\rho^{\phi\to\pi}$ in the l-hybrid model, with adversary $\mathcal{S}^l$ and environment $\mathcal{E}$. Continue a simulated execution from state σ (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 session of ϕ to be invoked within the test session of $\rho^{\phi\to\pi}$ in the simulated execution.
   (a) Some simulated ITI M passes input x to an ITI M' which is a party of session SID_l of π. In this case, save the current state of the simulated system in σ, pass input x from claimed source M to the external ITI M', and complete this activation.
   (b) The simulated $\mathcal{S}^l$ passes input (m, M) to the simulated adversary $\mathcal{S}_{\pi,l}$. In this case, pass the input (m, M) to the external 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 $l \stackrel{R}{\leftarrow} \{1..t\}$, and initialize a variable σ to hold the initial global state of a system of ITMs representing an execution of protocol $\rho^{\phi\to\pi}$ in the l-hybrid model, with adversary $\mathcal{S}^l$ and environment $\mathcal{E}$ on input z and security parameter k. Next, run Simulate(σ, l).
2. In any other activation, do:
   (a) Update the state σ. That is:
      i. If x, the new value written on the subroutine-output tape, 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 $\mathcal{S}_\pi$ that corresponds to session sid_l of ϕ.
      ii. If the new value x was written by another party, then interpret $x = (M, t, m)$ where M is an extended identity, t is a tape name and m is a value, and write m to tape t of the internally simulated ITI M. If no such ITI currently exists in the internal simulation, then one is invoked. (Recall that values written to the subroutine-output tape of the environment include an extended identity and target tape of a target ITI.)
   (b) Simulate an execution of the system from state σ. That is, run Simulate(σ, l).

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

of session sid_l of π in the right execution of (7). Similarly, the backdoor messages received by $\mathcal{E}_\pi^l$ from the external adversary in the left execution of (7) are exactly the backdoor messages that $\mathcal{S}^{l-1}$ receives from the parties and subsidiaries of session sid_l of π, and forwards to $\mathcal{E}_\pi^l$, in the right

execution of (7). The other backdoor messages that are received by $\mathcal{E}_\pi^l$ in the left execution of (7) are exactly those backdoor messages that $\mathcal{E}$ receives from the parties and subsidiaries of session sid_l of π in the right execution of (7).

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 $\rho^{\phi\to\pi}$ and $\mathcal{S}^l$ in the l-hybrid model. In particular, for any fixed value of the random choices, the messages that are directed by $\mathcal{E}_\pi^l$ to the external adversary in the left execution of (8) are exactly those messages that $\mathcal{S}^l$ forwards to the lth instance of $\mathcal{S}_\pi$ in the right execution of (8). The other messages that are sent by $\mathcal{E}_\pi^l$ in the left execution of (8) are exactly those messages that $\mathcal{E}$ sends to the parties and subsidiaries of session sid_l of ϕ in the right execution of (8). Similarly, the backdoor messages received by $\mathcal{E}_\pi^l$ from the external adversary in the left execution of (8) are exactly those backdoor messages that the lth instance of $\mathcal{S}_\pi$, running inside $\mathcal{S}^l$, outputs and $\mathcal{S}^l$ forwards to $\mathcal{E}_\pi^l$ in the right execution of (8). The other backdoor messages that are received by $\mathcal{E}_\pi^l$ in the left execution of (8) are exactly those backdoor messages that $\mathcal{E}$ receives from the parties and subsidiaries of session sid_l of ϕ in the right execution of (8). (Recall that, in the right execution of (8) $\mathcal{S}^l$ generates in addition directory queries to the lth instance of $\mathcal{S}_\pi$, and gets responses to these queries; but these responses are not forwarded to to $\mathcal{E}_\pi^l$).

It is stressed that a crucial reason for the validity of this analysis is that π and ϕ are subroutine respecting, namely that no subsidiary of any top-level instance of π or ϕ is passing input of subroutine-output to, or receiving inputs or subroutine-outputs from any ITI outside that extended instance.

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

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

in contradiction to the assumption that $\mathcal{S}_\pi$ is a valid simulator for 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 8) considers only environments that take external input that contains no information other than its import. We note that the UC theorem still holds even for this variant, with the same proof.

Composing multiple different protocols. The composition theorem (Theorem 18) is stated only for the case of replacing sessions of a *single* protocol ϕ with sessions of another protocol. The theorem holds also for the case where multiple different protocols $\phi_1, \phi_2, ...$ are replaced by protocols $\pi_1, \pi_2, ...$, respectively. (This can be seen either by directly extending the current proof, or by defining a single "universal" protocol that mimics multiple different ones.)

Nesting of protocol sessions. The universal composition operation can be applied repeatedly to perform "nested" replacements of calls to sub-protocols with calls to other sub-protocols. For instance, if a protocol π_1 UC-emulates protocol ϕ_1, and protocol π_2 UC-emulates protocol ϕ_2 using calls to ϕ_1, then for any protocol ρ that uses calls to ϕ_2 it holds that the composed protocol $\rho^{\phi_2 \to \pi_2^{\phi_1 \to \pi_1}} = \text{UC}(\rho, \text{UC}(\pi_2, \pi_1, \phi_1), \phi_2)$ UC-emulates ρ.

Recall that the UC theorem demonstrates that the simulation overhead grows under composition only by an additive factor that depends on the protocols involved. This means that security is preserved even if the nesting has polynomial depth (and, consequently, the UC theorem is applied polynomially many times).

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 μ in $\mathcal{C}$ such that, on any input and any random input, the output of a single instance of μ equals the output of (I, C). (Stated in these terms, Proposition 6 on page 29 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 8 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 21 *Let $\mathcal{C}$ be a self-simulatable class of ITMs, and let ρ, π, ϕ be protocols in $\mathcal{C}$ such that π UC-emulates ϕ with respect to class $\mathcal{C}$. Then protocol $\rho^{\phi \to \pi}$ UC-emulates protocol ρ 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.[19] 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 sessions of the ideal protocol for $\mathcal{F}$, and such that $\rho^{\mathcal{F} \to \pi}$ does not UC-emulate π. In fact, for *any* PPT π' we have that $\rho^{\mathcal{F} \to \pi'}$ does not emulate ρ.

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 $x \stackrel{R}{\leftarrow} s_{k,i}$ is computationally indistinguishable from $x \stackrel{R}{\leftarrow} \{0,1\}^k$. (b) $\mathcal{S}$ is evasive, that is for any non-uniform PPT algorithm A and for any $z \in \{0,1\}^*$, we have that $\text{Prob}[i \stackrel{R}{\leftarrow} \{0,1\}^k \ : \ A(z,i) \in s_{k,i}]$ is

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

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 $i \stackrel{R}{\leftarrow} \{0,1\}^k$ 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 $r \stackrel{R}{\leftarrow} s_{k,i'}$.

Protocol π for realizing $\mathcal{F}$ is simple: Given security parameter k it outputs $i \stackrel{R}{\leftarrow} \{0,1\}^k$. Given an input $x \in \{0,1\}^k$, it outputs $r \stackrel{R}{\leftarrow} \{0,1\}^k$. 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 sessions 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 $x_1 \stackrel{R}{\leftarrow} \{0,1\}^k$, 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, $\rho^{\mathcal{F} \to \pi}$ never outputs success. Furthermore, for *any* PPT protocol π' that UC-realizes $\mathcal{F}$, we have that $\rho^{\mathcal{F} \to \pi'}$ outputs `success` only with negligible probability.

6 UC formulations of some computational models

As discussed earlier, the basic model of computation provides no explicit mechanism for modeling communication over a network. It also provides only a single, limited mechanism for scheduling processes in a distributed setting, and no explicit mechanism for expressing adversarial control over, or infiltration of, computational entities. It also does not provide explicit ways to express leakage of information from computing devices. Indeed, in of itself, the model is not sufficiently convenient for representing realistic protocols, attacks, or security requirements.

This section puts forth mechanisms for capturing realistic protocols, attacks and security requirements, by way of setting conventions on top of the basic model of Section 4.1. It also formulates a number of basic ideal functionalities that capture common abstractions, or models of communication; As motivated in the Introduction, these abstract models allow composing protocols that use the ideal functionality as an abstract model with protocols that realize the functionality using less abstract modeling, while preserving overall security.

In addition to capturing some specific conventions and ideal functionalities, this section exemplifies how the basic model can be used as a platform for more fine-tuned and expressive models. It also provides a general toolbox of techniques for writing ideal functionalities that capture other situations, concerns, and guarantees.

We start with presenting, in Section 6.1, a mechanism for writing protocols in a way that provides clear distinction between modules that represent "artificial model instructions" that are only used in the analysis, and modules that represent "real code" that is meant to be actually executed in a physical implementation. Using this mechanism, we then demonstrate how to capture various corruption models, as well as some useful conventions for writing ideal functionalities.

Next we present, in section 6.2, ideal functionalities that capture some commonplace abstract models of communication, specifically authenticated, secure, and synchronous communication. Fi-

nally, in Section 6.3 we present an ideal functionality that captures non-concurrent protocol execution.

6.1 Writing conventions for protocols and ideal functionalities

Section 6.1.1 presents and exemplifies the set of conventions for separating "real code" from "model instructions". Section 6.1.2 defines a variety of party-corruption models. Section 6.1.3 presents some useful conventions for writing ideal functionalities.

6.1.1 A basic convention: Bodies and Shells

Our first convention provides a way to distinguish between protocol instructions that represent "real code" and instructions 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. Roughly, the body will contain the "real code", and the shell will contain the model-related instructions.

Formally, the shell can read and modify the state of the body; however the body does not have access to the state of the shell. An activation of a compliant protocol starts by running the shell, which may (or may not) execute the body sub-process. In case the body executes, it keeps executing until it reaches a special "end of activation state", at which point the shell resumes executing. In particular, while the body may prepare all the information necessary for executing an external-write operation, it may not execute this operation (as this would end the overall activation of the ITI). Only the shell can execute external-write. For simplicity we assume that the runtime of protocols includes the runtime of both the shell and the core.)

Throughout this section we assume all protocols and ideal functionalities are compliant. We use different shells to capture a variety of conventions and models.

Subroutine exposing protocols. As an example for the use of body and shell, we present a specific implementation of the notion of subroutine exposing protocols (see Section 5.1 on page 54). We say that a protocol π is subroutine exposing if its shell includes the following mechanisms:

1. Let $\top$ be a special identifier, interpreted as "directory". Before performing a forced-write to an input tape of ITI M, the shell of ITI (π, s, p) first sends input `(invoking` M`)` to $(\pi, s, \top)$. Upon receiving subroutine-output `ok` from $(\pi, s, \top)$, it resumes processing its external-write instruction.

2. In the first activation of an ITI with code π, session ID s and party ID $p \neq \top$, the shell sends input `started` with reveal-sender-identity and forced-write flags set to a special directory ITI $(\pi, s, \top)$. Upon receiving `ok` from that directory ITI, it resumes processing its first activation.

3. The shell includes the following two instructions in the shell code of any subroutine ITI M it invokes: (a) An instruction to report to ITI $(\pi, s, \top)$ upon first activation and before any forced-write to input tape of another ITI, in the same way as in steps (1) and (2) above; (b) An instruction to embed these two instructions in the shells of all the subroutines of M.

   It is stressed that the subsidiaries of instance s of π use the same directory ITI as instance s of π. That is, the directory serves all the ITIs in the extended instance (π, s).

4. If the local identity is $(\pi, s, \top)$, then the body is never activated. Instead, when activated by an ITI M with input `started`, where either M' is a main party of instance s of π, or M is recorded as eligible, then the shell records M as a member of the extended session of s, and outputs `ok` to M.

   When activated by an already-registered ITI M with input `(invoking` M'`)`, the shell records M' as eligible and outputs `ok` to M.

   Upon receiving a backdoor message `query` from the adversary, the shell returns to the adversary the full list of registered ITIs.

To make sure that the directory ITI remains polytime, we require that each `(invoking` M`)` input should carry enough import to cover the cost of registering M, and each query from the adversary must contain sufficient import to cover the cost of the processing of that query.

6.1.2 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, adversarial ("Byzantine") behavior, resilience to viruses and exploits, resilience to side channel attacks, incoercibility, 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 backdoor 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.

One issue that needs to be addressed by any mechanism for modeling party corruption within the current framework is to what extent should the environment be made aware of the corruption operation. On one extreme, if the environment remains completely unaware of party corruptions, then our notion of protocol emulation would become rather loose; in particular protocol π could emulate protocol ϕ even if attacks mounted on π without corrupting anyone can only be emulated by attacks on ϕ that corrupt all participants. On the other extreme, if the environment learns the entire extended ID of all corrupted parties then the emulated and emulating protocol would need to be identical.

We provide the following mechanism for party corruption. This mechanism determines how the information regarding which ITIs are currently corrupted is collected and made available to the environment. The mechanism also allows the protocol analyst to determine the amount of information that the environment learns about the corruption operations, thereby determining the level of security provided by UC emulation.

To corrupt an ITI M, the adversary writes $(\texttt{corrupt}, cp)$ on the backdoor tape of M, where cp is some parameter of the corruption. Let $f : \{0,1\}^* \to \{0,1\}^*$ be an "identity masking" function, which partially masks the party's identity. (See more discussion on this function below.) Say that a protocol π is standard f-revealing corruption if the shell of π includes the following instructions: (a) Upon receipt of a backdoor message $(\texttt{corrupt}, cp)$ message, the shell of ITI (π, s, p) first passes a $(\texttt{corrupt}, cp, f(\pi, s, p))$ output to a special corruption aggregation ITI $(\pi, s, \perp)$ where $\perp$ is a special identifier. (b) When (π, s, p) invokes a subroutine ITI M, the shell of ITI (π, s, p) includes in

the shell of M instructions to report corruptions to $(\pi, s, \perp)$, and to further instruct the shells of subsidiaries of M' to do the same.

The corruption aggregation ITI $(\pi, s, \perp)$ proceeds as follows: When invoked by with a corruption notification $(\texttt{corrupt}, cp)$ from ITI M, it records (p, M) and returns control to M. When invoked with a `(report)` input (presumably from the environment), $(\pi, s, \perp)$ returns the list of all notifications of corruption.

Once the control returns to the corrupted ITI, the behavior of the shell changes according to the specific corruption type. See some examples below.

On the identity masking function. The identity masking function is a mechanism that allows specifying how much information the environment obtains about the corrupted ITIs, and in particular how much information it obtains about the actual subroutine structure of the analyzed protocol. This, in turn, determines the degree by which the subroutine structure of π can diverge from the subroutine structure of ϕ, and still have π UC-emulate ϕ.

For instance, consider the case where $f(\pi, id) = id$, i.e. f outputs the identity (but not the code) of the corrupted ITI. This means that the environment receives the identities of all the corrupted parties. In this case, if protocols π and ϕ are standard f-revealing corruption and π UC-emulates ϕ, then it must be the case that for each ITI in an extended instance of π there exists an ITI in the extended instance of ϕ with the same identity.

A somewhat more relaxed identity-masking function returns only the *pid* of the corrupted ITI. This makes sense when ITIs are naturally grouped into "clusters" where all the ITIs in a cluster have the same *pid*, and allows hiding the internal subroutine structure within a cluster. (A cluster may correspond to a single physical computer or a single administrative entity.) This identity-masking function is instrumental in modeling *pid-wise corruptions,* discussed in Section 6.1.3.

Another natural identity-masking function considers the case where the *sid* is constructed in a hierarchical way and includes the names of parent instances in the "subroutine graph". Here the identity-masking function returns the *pid*, plus information of some ancestors of the current *sid*. This allows capturing cases where π and ϕ consist of multiple "sessions" of another protocol, where the number and identity of sessions is the same in π and in ϕ, but the internal subroutine structure within each session in π is different than in ϕ.

Finally we note that π and ϕ need not be standard f-revealing corruption with respect to the same f. In particular, when ϕ is the ideal protocol for some ideal functionality, its behavior in case of party corruption will often be very different. See more details in Section 6.1.3.

Byzantine corruption. Perhaps the simplest form of corruption to capture is total corruption, often called Byzantine corruptions. A protocol (or, rather, a shell) is Byzantine-corruptions if, upon receiving the (`corrupt`) message, the shell first complies with the above standard-corruption requirement. From this point on, upon receiving value m on the backdoor tape, the shell external-writes m (Where m presumably includes the message, the recipient identity and all relevant flags). In an activation due to an incoming input or subroutine-output, the shell sends the entire local state to the adversary. The setting where data erasures are not trusted can be modeled by restricting to *write once* protocols, i.e. protocols where each data cell can be written to at most once. Note that here the body of π becomes inactive from the time of corruption on.

Non-adaptive (static) corruptions. The above formulation of Byzantine-corruption shells 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 if 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 a party to a (`corrupt`) message from the adversary, as follows. A protocol π is passive corruptions if, upon receiving a (`corrupt`) message, a corrupted flag is set. Upon receipt of an input or subroutine-output, the shell activates the body, and at the end of the activation, if the corrupted flag is set, then it 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 and remains inactive in all future activations. When activated due to another incoming message from the adversary, the shell forwards the message to the body, and delivers any message that the body prepares to write in this activation.

We make two additional remarks: First, the variant defined here allows the adversary to learn whenever a passively corrupted party is activated; it also allows the adversary to make the party halt. Alternative formulations are possible, where the adversary only learns the current state of a corrupted party, and is not allowed to make the party halt.

Second, the variant defined here does not allow the adversary to modify *input values* to the party. Alternative formulations, where the adversary is allowed to modify the inputs of the corrupted parties, have been considered in the literature. Such formulations can be naturally represented here as well. (Note however that with non-adaptive corruptions the two variants collapse to the same one.)

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. In a way, leakage can be thought of as a more nuanced variant of passive corruption, where the corrupted party discloses only some function of its internal state, and furthermore does it only once (per leakage instruction).

In of itself, this modeling is somewhat limited in that it allows the adversary to only obtain leakage information from individual processes (or, ITIs). To capture realistic settings where side-channel attacks collect joint information from multiple protocol sessions that run on the same physical device, and protocols that are resilient to such attacks, one needs to augment the formal modeling of side channel attacks with a mechanism that allows for joint, non-modular leakage from multiple ITIs. Such a mechanism is described in [BCH12].

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 incoming messages and inputs to the adversary, and stops following the adversary's instructions. `(recover)` messages are reported to the corruption aggregation ITI defined above in the same way as corruption messages.

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 coercion mechanism considered here 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 verify consistency with the public transcript of the protocol.

Resilience to coercion is meaningful in settings where the participants are humans that are susceptible to social pressure; Voting schemes are a quintessential example.

The idea is to provide the entities running the protocol with a mechanism by which they can provide the attacker with "fake input" and "fake random input" that will be indistinguishable for the adversary from the real input and random input that were actually used in the protocol execution. This way, the attacker will be unable to tell whether the attacked party used the fake input or perhaps another value.

In the present framework, coercion attacks can be modeled as follows, along the lines of [CG96, CGP15]. We assume that the protocol description includes a "faking algorithm" F. Furthermore, each ITI M has a "caller ITI", which represents either an algorithm in of itself or a human user. We say that a protocol is coercion compliant if, upon receipt of a coercion instruction, the shell of the recipient ITI notifies its caller ITI that a coercion instruction was received. Then, if the caller ITI returns a "cooperate" message, then the coerced ITI discloses its entire internal state to the adversary. If the parent ITI returns a "report fake input v" message, then the coerced ITI runs F on v and its current internal state, and sends the output of F to the adversary.

Incoercibility, or resilience to coercion attacks, is then captured by way of realizing an ideal functionality $\mathcal{F}$ that guarantees 'ideal incoercibility' as follows: Upon receiving an instruction to coerce some party P, the ideal functionality forwards this request to the caller ITI of P. If in return the caller ITI inputs an instruction to cooperate, the ideal functionality reports to the adversary the true input of P; If the caller ITI returns an instruction to fake an input v, then the ideal functionality simply reports v to the adversary.

We then say that π is incoercible if it is coercion compliant, and in addition it UC-realizes an

ideal functionality $\mathcal{F}$ that guarantees ideal incoercibility.

6.1.3 Writing conventions for ideal functionalities

We present a number of conventions and mechanisms for writing ideal functionalities in a way that captures some commonplace security requirements.

Determining the identities of parties that provide input and receive outputs. Recall that the framework provides a way for an ideal functionality $\mathcal{F}$ to learn the extended identities of the ITIs that provide inputs to it, and to determine the extended identities of the ITIs that obtain output from it. In fact, this holds not only with respect to the immediate callers of $\mathcal{F}$, which are the dummy parties in the ideal protocol for $\mathcal{F}$. Rather, this holds also with respect to the ITIs that provide inputs to these dummy parties, and the ITIs that obtain subroutine-outputs from them. We note that this feature of the framework is crucial for the ability to capture realistic tasks. (A quintessential example for this need is $\mathcal{F}_{\text{AUTH}}$, the message authentication functionality, described in the next section.)

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 which received the input from an ITI with extended identity P". Similarly we say "generate output v for party P", meaning "perform an external-write operation of value v to a dummy party that will in turn write value v on the subroutine-output tape of ITI with extended identity P." Note that the dummy ITI, as well as ITI P, may actually be *created* as a result of this write instruction.

We also 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 $\mathcal{F}$.

Behavior upon party corruption. In the ideal protocol $\text{IDEAL}_{\mathcal{F}}$, corruption of parties is modeled as messages written by the adversary on the backdoor tape of the ideal functionality $\mathcal{F}$. (Recall that, by convention, backdoor 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 first restrict attention to the case where $\mathcal{F}$ only accepts corruption instructions for identities that match the identities of the existing dummy parties, or in other words the identities of the main parties of $\text{IDEAL}_{\mathcal{F}}$. Specifically, we say that an ideal functionality $\mathcal{F}$ is standard PID-wise corruption if the following holds:

1. Upon receiving a `(corrupt` p`)` message from the adversary, where p is a party ID of a dummy party for the present session of $\text{IDEAL}_{\mathcal{F}}$, $\mathcal{F}$ marks p as corrupted and returns 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; $\mathcal{F}$ then uses v' instead of v. Also, all output values intended for p are sent to the adversary instead.

2. Upon receiving a `(Report)` input from some caller ITI, $\mathcal{F}$ returns the list of corrupted identities to the caller.

The above set of instructions captures the standard behavior of the ideal process upon corruption of a party in existing definitional frameworks, e.g. [C00, G04]. Note that here the "granularity" of corruption is at the level of party ID for the main parties of the instance. That is, a party can be either uncorrupted or fully corrupted. This also means that the security requirements from any protocol π that realizes $\mathcal{F}$ is only at the granularity of corrupting main parties. This is so even if the main parties of π have subroutines and these subroutines are corrupted individually. (In particular, the identity-masking function of π can only output identities of main parties of π.)

Alternatively, ideal functionalities might be written so as to represent more refined corruption mechanisms, such as corruption of specific subroutines or sub-sessions, forward secrecy, leakage, coercion, etc. Furthermore, ideal functionalities may change their global behavior depending on the identity or number of corrupted parties. We leave further discussion and examples out of scope.

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 M if it engages in the following interaction: Instead of simply outputting v to M, $\mathcal{F}$ first sends to the adversary (on the backdoor tape) a message that it is ready to generate an output to M. If the output is public, then the value v is included in the note to the adversary. If the output is private then v is not mentioned in this note. Furthermore, the note 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 (say, by echoing the unique identifier on $\mathcal{F}$'s backdoor tape), $\mathcal{F}$ outputs the value v to M.

Running arbitrary code. It is often convenient to let an ideal functionality $\mathcal{F}$ receive a description of an arbitrary code c from the adversary, and then run this code while inspecting some properties of it. One use of this "programming 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, $\mathcal{F}$ will output an error message to the relevant parties. Examples of this use include the signature and encryption functionalities as formalized in [CH11, C05]. Other examples exist in the literature. Another use for this technique is to enable expressing the requirement that some adversarial processes be carried out in isolation from the external environment the protocol runs in. An example for this use is the formulation of non-concurrent security in Section 6.3.

At first glance, this technique seems problematic in that $\mathcal{F}$ is expected to run algorithms of arbitrary polynomial runtime, whereas $\mathcal{F}$'s own runtime is bounded by some fixed polynomial. We get around this technicality by having $\mathcal{F}$ not run c directly, and instead invoke a subroutine ITI γ for running c, where the polynomial bounding the runtime of γ appropriately set by $\mathcal{F}$. The import to γ would be provided by the adversary, namely it would be included in the request to run c (which is written on $\mathcal{F}$'s backdoor tape) and then handed over to γ by $\mathcal{F}$.

demonstrate how to use In the rest of Section 6.1 we demonstrate how to use this set of conventions for expressing a number of traditional party-corruption models.

6.2 Some communication models

We turn to capturing, within the UC framework, some abstract models of communication. We consider four commonplace models: Completely unprotected (or, adversarially controlled) communication, authenticated point-to-point communication, secure point-to-point communication, and synchronous communication.

We first note that the present bare framework, without ant additional ideal functionalities, already provides a natural way for modeling communication over an unprotected communication medium that provides no guarantees regarding the secrecy, authenticity, or delivery of the communicated information. Specifically, sending of a message over such a communication medium amounts to forwarding this message to the adversary. Receiving a message over such a medium amounts to receiving the message from the adversary. (Expressing this high level specification in terms of body and shell may proceed as follows: When the body completes an activation with an outgoing message $(\texttt{network}, m)$, the shell writes $(\texttt{network}, m)$ on the backdoor tape of the adversary. Similarly, when activated with a message $(\texttt{network}, m)$ on the backdoor tape, the shell activates the body with incoming message $(\texttt{network}, m)$.)

Capturing the other three abstractions requires more work. Sections 6.2.1, 6.2.2, and 6.2.3 present ideal functionalities for capturing authenticated, secure, and synchronous communication, respectively.

6.2.1 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 completed the protocol.*

We assume that the sender S knows R, namely the identity of the receiver, at the onset of the protocol. However, if S gets corrupted during the course of the protocol execution then the actual receiver identity R' may be different than the original intended identity, R. Furthermore, R' may be determined adversarially and adaptively during the protocol execution. The receiver may not have any knowledge of S ahead of time, yet it learns the sender identity by the end of the protocol.

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. In its first activation, $\mathcal{F}_{\text{AUTH}}$ expects its input to be of the form $(\texttt{Send}, sid, R, m)$ with sender ITI S. (Here S and R are extended identities, namely codes and identities. We stress that these are not the dummy parties for $\mathcal{F}_{\text{AUTH}}$. Rather, S is the ITI that provides input to the sender-side dummy parties and R is the ITI that gets output from the receiver-side dummy party. $\mathcal{F}_{\text{AUTH}}$ then generates a public delayed output $(\texttt{Send}, sid, S, R, m)$ 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 R. (More precisely, $\mathcal{F}_{\text{AUTH}}$ outputs this value to a dummy party with identity (sid, R); that dummy party then outputs this value to R.)

$\mathcal{F}_{\text{AUTH}}$ is a standard corruption functionality as defined in Section 6.1.3 above. In addition, if the adversary instructs to corrupt the sender before the output value was actually delivered to

R, then $\mathcal{F}_{\text{AUTH}}$ allows the adversary to provide new, arbitrary message m' and recipient extended identity R'. $\mathcal{F}_{\text{AUTH}}$ outputs $(\texttt{Send}, sid, m')$ to R'.

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

1. Upon receiving an input $(\texttt{Send}, sid, R, m)$ from party S, do: If this is the first $(\texttt{Send...})$ input then record R, m and send $(\texttt{Sent}, sid, S, R, m)$ to the adversary; else do nothing. When receiving $(\texttt{ok})$ from the adversary for the first time, output $(\texttt{Sent}, sid, S, R, m)$ to R and halt.
2. Upon receiving $(\texttt{Corrupt}, sid, P, m', R')$ from the adversary, where $P \in \{S, R\}$, disclose m to the adversary. Next if $P = S$ and not yet halted, then record R', m' instead of R, m.

Figure 11: The Message Authentication functionality, $\mathcal{F}_{\text{AUTH}}$

We highlight several points regarding the security guarantees provided by $\mathcal{F}_{\text{AUTH}}$. First, $\mathcal{F}_{\text{AUTH}}$ allows the adversary to change the contents of the message and destination, 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. This provision captures the fact that in general the received value is not determined until the point where the recipient actually generates its output.[20]

Second, $\mathcal{F}_{\text{AUTH}}$ reveals the contents of the message, as well as the extended identities of the sender and the receiver, to the adversary. This captures the fact that secrecy of the message and of the sender and receiver identities is not guaranteed. (One might argue that revealing the code of the sender and receiver is not called for and exposes too much about the participants; however, this modeling issue can be easily overcome by designing protocols such that the codes of the actual sender and receiver ITIs do not contain sensitive information.)

Third, $\mathcal{F}_{\text{AUTH}}$ 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 or not achievable, it might suffice to use an appropriately relaxed variant of $\mathcal{F}_{\text{AUTH}}$ (see e.g. [CSV16]).

Finally, we highlight two modeling choices 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.[21]

[20] Early 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].

[21] We point out that this non-interactive formulation of $\mathcal{F}_{\text{AUTH}}$ makes crucial use of the fact that the underlying

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, with any PPT environment and adversary, no party ever generates output with non-negligible probability . Then, we have:

Claim 22 ([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 $\mathcal{F}_{\text{AUTH}}$ algorithmically, given some other abstractions on the system. Following the same definitional approach, these abstractions are again formulated by way of ideal functionalities. One such ideal functionality (or, rather, family of ideal functionalities) allows the parties to agree on secret values in some preliminary stage, thus capturing a "pre-shared key" or perhaps "password" mechanisms. Another family of ideal functionalities provide the service of a trusted "bulletin board", or "public ledger", where parties can register public values (e.g., public keys), and where potential receivers can correctly obtain the public values registered by a party.

In this context, different abstractions (ideal functionalities) represent different physical, social and algorithmic mechanisms for providing authentication, or binding between long-term entities and cryptographic constructs that can be used in message authentication. Indeed, different ideal functionalities lead to different authentication protocols and mechanisms.

An important aspect of the modeling of these methods of binding between identities and keys (whether these are based on pre-shared keys, on public-key infrastructure, or other means) is the fact that realistic binding methods are typically long-lived, and are in particular used for authentication of multiple messages. This appears to be incompatible with the formulation of $\mathcal{F}_{\text{AUTH}}$ as an ideal functionality that handles a single message. Indeed, a protocol that UC-realizes $\mathcal{F}_{\text{AUTH}}$ using long-term-binding module (say, a digital signature algorithm) cannot be subroutine-respecting, unless each instance of the protocol uses a new instance of the long-term-binding module - which does not capture reality.

Two main mechanisms appear in the literature to address this issue, while maintaining the compositional aspects of the framework. One mechanism, called universal composition with joint state (JUC), provides a way to analyze multiple instances of some protocol π, where all these instances use some common module (e.g., a common instance of an ideal functionality $\mathcal{G}$ which satisfied some special properties) "as if" each one of the instances of π has its own exclusive instance of $\mathcal{G}$ [CR03, KT13]. This mechanism can be used within the current UC framework.

Another mechanism, called Generalized UC security (GUC), provides a more general treatment of the task of performing modular security analysis for systems that are not "completely decomposable" to separate components (that take the form of subroutine-respecting protocols) [CDPW07]. In particular, GUC security allows arguing about the security of protocols that use some "globally accessible" modules (or services, or ideal functionalities), whereas these global services can

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.

be used by arbitrary other protocols in the systems. For instance, GUC security allows arguing about the security properties of protocols that use public-key infrastructure (or some other mechanism) that is accessible to *anyone,* not only to the parties running the protocol. See for instance [DKSW09, CSV16].

We note however, that it is possible to model systems that provide multiple instances of some service (e.g. authenticated communication) with access to a "globally accessible module" (e.g. public-key infrastructure, or PKI) also within the plain UC framework presented in this work, and obtain similar security and de-composability guarantees.[22]

6.2.2 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.)

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 $(\texttt{Send}, sid, R, m)$ from party S, do: If this is the first `(Send...)` input then record R, m and send $(\texttt{Sent}, sid, S, R, l(m))$ to the adversary; else do nothing. When receiving `(ok)` from the adversary for the first time, output $(\texttt{Sent}, sid, S, R, m)$ to R.
2. Upon receiving $(\texttt{Corrupt}, sid, P, m', R')$ from the adversary, where $P \in \{S, R\}$, disclose m to the adversary. Next if $P = S$ and have not yet received `(ok)` from the adversary, then record R', m' instead of R, m.

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

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 $\mathcal{F}_{\text{SMT}}$. Following our convention regarding party corruption, when either the sender or the receiver are corrupted, $\mathcal{F}_{\text{SMT}}$ discloses (R, m) to the adversary. In addition, like $\mathcal{F}_{\text{AUTH}}$, $\mathcal{F}_{\text{SMT}}$ allows the adversary to change the contents of the message

[22] To do that, consider the protocol that provides a single instance of the service (authenticated transmission of a single message), *plus the service provided by the globally accessible module* (PKI), and show that this protocol UC-realizes an ideal functionality that provides a single ideal instance of authenticated message transmision, plus the PKI service. Then, by iterating this process for each transmitted message, one demonstrates that the real system UC-emulates a system with only ideal authentication, alongside a globally accessible PKI. A similar process can be used to realize joint-state composition (JUC).

and the identity of the recipient as long as the sender is corrupted *before message delivery.* This is so even if the sender was uncorrupted at the point when it sent the message.

Another difference between $\mathcal{F}_{\text{SMT}}$ and $\mathcal{F}_{\text{AUTH}}$ is that $\mathcal{F}_{\text{SMT}}$ remains active even after the message was delivered. This is done to capture information leakage that happens when either the sender or the receiver are compromised even long after the protocol completed its execution.

Stronger variants. *Forward Secrecy* is the requirement that the message should remain secret even if the sender and/or the receiver are compromised — as long as the compromise happened *after* the protocol execution has ended. 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 if the corruption happened after the message has been delivered. The rest of the code of $\mathcal{F}_{\text{SMT}}$ remains unchanged.

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 learns that a message was sent but not the sender or receiver.

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 session of $I_{\mathcal{F}_{\text{AUTH}}}$ with input $(\texttt{Send}, sid'', \texttt{init-smt})$, where $sid'' = (S, R, sid')$, 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 23 *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 corruptions.*

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

As in the case of $\mathcal{F}_{\text{AUTH}}$, it is possible to realize multiple instances of $\mathcal{F}_{\text{SMT}}$ using a single

session of a more complex protocol, in a way that is considerably more efficient than running multiple independent sessions 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 however 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]. Using the UC with joint state mechanism [CR03], one can still design and analyze protocols that employ multiple independent instances of $\mathcal{F}_{\text{SMT}}$, in spite of the fact that all these instances are realized by a single (or few) instances of an encryption protocol.

6.2.3 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.

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 within the general, unmodified framework by having access to an ideal functionality $\mathcal{F}_{\text{SYN}}$ that provides the same guarantees as the ones that are traditionally provided in synchronous networks. We first present $\mathcal{F}_{\text{SYN}}$, and then discuss and motivate some aspects of its design.

Specifically, $\mathcal{F}_{\text{SYN}}$ is aimed at capturing a basic variant of the synchronous model, which provides the following two guarantees:

Round awareness. All participants have access to a common variable, representing the current round number. The variable is non-decreasing.

Synchronized message delivery. Each message sent by an uncorrupted party is guaranteed to arrive in the next round. In other words, all the messages sent to a party at round $r - 1$ are received before the party sends any round-r messages.

The second guarantee necessarily implies 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.

We note that the first requirement is not essential, i.e. there exist meaningful notions of synchronous communication which do not imply common knowledge of the round number. Still, we impose it here for sake of simplicity. It is possible to capture within the present formalism also weaker notions of synchronous communication.

Finally, we stress that 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.

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 ITIs (i.e., extended identities) among which synchronization is to be provided. It proceeds as follows.

1. At the first activation, initialize a round counter $r \leftarrow 1$, initialize $L_P^0 = \perp$ for all $P \in \mathcal{P}$.
2. Upon receiving input $(\texttt{Send}, sid, \mu)$ from a party $P \in \mathcal{P}$, where $M = \{(m_i, R_i)\}$ is a set of pairs of messages m_i and recipients $R_i \in \mathcal{P}$, record (P, M, r) and send (sid, P, M, r) to the adversary.
3. Upon receiving input $(\texttt{Receive}, sid, r')$ 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 provided their messages for round r, then:
      i. Let N be 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.)
      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 $(\texttt{Received}, sid, L_P^{r'})$ to P.
   (b) If $r' < r$ then output $(\texttt{Received}, sid, L_P^{r'})$ to P.
   (c) Else (i.e. $r' > r$ or $r' = r$ and some uncorrupted party did not provide its round-r messages), output $(\texttt{Round } r \texttt{ incomplete})$ to P.

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

$\mathcal{F}_{\text{SYN}}$, presented in Figure 13, expects its SID to include a list $\mathcal{P}$ of parties among which synchronization is to be provided. It also assumes that all parties in $\mathcal{P}$ are notified of the existence of the present instance of $\mathcal{F}_{\text{SYN}}$ in other means. (Alternatively, $\mathcal{F}_{\text{SYN}}$ could send a delayed public output `init` to all parties in $\mathcal{P}$ - but in this case guaranteed delivery would be lost.)

At the first activation, $\mathcal{F}_{\text{SYN}}$ initializes a round number r to 1. Next, $\mathcal{F}_{\text{SYN}}$ responds to two types of inputs: Given input of the form $(\texttt{Send}, sid, \mu)$ from party $P \in \mathcal{P}$, $\mathcal{F}_{\text{SYN}}$ interprets μ as a list of messages to be sent to other parties in $\mathcal{P}$. The list μ is recorded together with the sender identity and the current round number, and is also forwarded to the adversary. (This is the only point where $\mathcal{F}_{\text{SYN}}$ yields control to the adversary. Notice that guaranteed delivery of messages is not harmed since in its next activation, $\mathcal{F}_{\text{SYN}}$ will continue without waiting for the adversary's response.) $\mathcal{F}_{\text{SYN}}$ is a standard corruption functionality. That is, 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.

First, if r' is the current round number and all uncorrupted parties have sent their messages for this round, then $\mathcal{F}_{\text{SYN}}$ prepares the list of messages sent to each party at this round, and advances the current round number r. Else r remains unchanged.

Next, if $r' < r$ then $\mathcal{F}_{\text{SYN}}$ returns to the requesting party the messages sent to it in this round. If $r' \geq r$ (i.e., round r' is still incomplete) then an error message is returned.[23]

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 way, the functionality can make a set of values available to multiple parties at the same time, thus guaranteeing both fairness and guaranteed delivery of messages. Indeed, a protocol that uses $\mathcal{F}_{\text{SYN}}$ can be guaranteed that as soon as all uncorrupted parties 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.

Using $\mathcal{F}_{\text{SYN}}$. 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. 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.[24] In either case, each party of π first initializes a round counter to 1, and inputs to $\mathcal{F}_{\text{SYN}}$ a list μ 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 $(\texttt{Receive}, sid, r)$, 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 $(\texttt{Send}, sid, \mu)$ where μ contains the outgoing messages for this round. If $\mathcal{F}_{\text{SYN}}$ returns `(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, $\mathcal{F}_{\text{SYN}}$ guarantees 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.

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 and realistic settings such as synchronous execution of a protocol within a larger asynchronous environment, or several protocol executions where each execution is internally synchronized 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"

[23] The formulation of $\mathcal{F}_{\text{SYN}}$ in earlier versions of this work was slightly different: It explicitly sent a notification message to the adversary at any advancement of the round number, and waited for a confirmation from the adversary before advancing the round number. This allowed the adversary to block the advancement of the round number, which meant that the functionality did not guarantee delivery of messages. This flaw is pointed out in [KMTZ13], where an different fix to the one used here is proposed.

[24] We note that such an adaptive initialization mechanism might, if implemented incorrectly, interfere with the guaranteed delivery of messages. Still, it can also be implemented without giving up on guaranteed delivery, using the same logic as for incrementing the round number. That is, the ideal functionality will output appropriate notifications to added participants; in its next activation, the ideal functionality will then continue as before, without waiting for confirmation from the added participant.

way. That is, inputs may be received at any time and there is no guarantee that all or most inputs are received within the same round. Still, protocols that use $\mathcal{F}_{\text{SYN}}$ can deploy standard mechanisms for guaranteeing that the actual computation does not start until enough (or all) parties have inputs.

Finally we remark that including the set $\mathcal{P}$ of participating ITIs within the SID is aimed at capturing situations where the identities of all participants are known to the initiator in advance. Situations where the set of participants is not known a priori can be captured by letting parties join in as the computation proceeds, and having $\mathcal{F}_{\text{SYN}}$ update the set $\mathcal{P}$ accordingly.

On composing $\mathcal{F}_{\text{SYN}}$-hybrid protocols. Within the present framework, where protocols are bound to be subroutine-respecting, an instance of $\mathcal{F}_{\text{SYN}}$ cannot be used as a subroutine by two different protocols instances (π, sid) and (π', sid'), unless one instance is a subroutine of the other. This means that if we want to consider a (perhaps composite) protocol where the communication is synchronous across all parties of an instance of the protocol, then we must analyze the entire protocol as a single unit and cannot meaningfully de-compose this protocol to smaller units that can be analyzed separately.

Composability (or, rather, de-composability) can be regained via using either the Universal Composition with Joint State (JUC) theorem or alternatively via the Generalized UC (GUC) framework [CR03, CDPW07]. The JUC theorem provides a way to analyze individual instances of some protocol π, where each instance uses its own instance of $\mathcal{F}_{\text{SYN}}$, and then argue that the overall behavior does not change even if all instances of π use the same instance of $\mathcal{F}_{\text{SYN}}$. (Here care must be taken to account for protocols that take different number of rounds to complete.) The GUC framework allows modeling the behavior of protocols that assume a "globally synchronous" system, i.e. a system where *all* components, even arbitrary ones that are not part of the current design, have access to a single, global, instance of $\mathcal{F}_{\text{SYN}}$ (or a variant thereof). As sketched in Footnote 22, de-composability can also be regained within the present framework by treating $\mathcal{F}_{\text{SYN}}$ as part of both the protocol and the ideal functionality that the protocol UC-realizes.

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. One may relax this "timeliness" guarantee as follows. $\mathcal{F}_{\text{SYN}}$ may only guarantee 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 *non-blocking asynchronous communication model* where message are guaranteed to be delivered, but with unknown delay (see, e.g., [BCG93, CR93]).

6.3 Non-concurrent Security

One of the main features of the UC framework is that it guarantees security even when protocol sessions 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 of protocols within the present framework. That is, we present a general methodology for writing protocols so that no attacks against the protocol, that involve executing other protocols concurrently with the analyzed protocol, will be expressible in the model.

Recall that the main difference between the UC model and models that guarantee only non-concurrent security is that in the UC model the environment expected to be able to interact with the adversary at any point in the computation, whereas in non-concurrent models the environment receives information from the adversary only once, at the end of the computation. The idea is to write protocols in a way that essentially forces the UC environment to behave as if it runs in a non-concurrent model.

In fact, we demonstrate how to transform *any* given protocol π into a protocol π_{NC}, such that π_{NC} provides essentially the same functionality as π, except that π_{NC} forces the enviroment to behave non-concurrently. The idea is to replace all interaction between π and the adversary for interaction between π and a special ideal functionality, called $\mathcal{F}_{\text{NC}}$, that mimics the adversary for π, and interacts with the actual adversary only in a very limited way.

That is, let π_{NC}, the non-concurrent version of π, be identical to π except that any external-write to the backdoor tape of the adversary is replaced be an input to an instance of $\mathcal{F}_{\text{NC}}$ with the same session ID as the local one; similarly, subroutine outputs coming from this instance of $\mathcal{F}_{\text{NC}}$ are treated like messages coming on the backdoor tape. Incoming messages from the actual trapdoor tape are ignored.

Functionality $\mathcal{F}_{\text{NC}}$ is presented in Figure 14. It expects to receive from its activator a session ID s. It then notifies the adversary of its session ID and waits to receive a code $\hat{\mathcal{A}}$ on its backdoor tape. ($\hat{\mathcal{A}}$ represents an adversary in a non-concurrent model). $\mathcal{F}_{\text{NC}}$ then behaves in the same way that adversary $\hat{\mathcal{A}}$ would in the non-concurrent security model. That is, $\mathcal{F}_{\text{NC}}$ runs $\hat{\mathcal{A}}$, feeds it with all the inputs messages from the parties of this extended instance of π, and follows its instructions with respect to sending information back to the parties. (Of course, $\hat{\mathcal{A}}$ sends this information as subroutine outputs rather than backdoor messages.) In addition, $\mathcal{F}_{\text{NC}}$ verifies that $\hat{\mathcal{A}}$ stays within the allowed boundaries of the model, namely that it only delivers backdoor messages to existing ITIs that are parties or subsidiaries of the session s of the calling protocol. (For this purpose, we assume that π is subroutine-exposing.) As soon as $\hat{\mathcal{A}}$ generates an output$v$ to the environment, $\mathcal{F}_{\text{NC}}$ sends v to the external adversary and halts.

Notice that the above formalism applies also to protocols that assume some idealized communication model, say by using an ideal functionality that represents that model (e.g., $\mathcal{F}_{\text{AUTH}}$ or $\mathcal{F}_{\text{SYN}}$). Indeed, when applied to protocols that use an ideal functionality such as $\mathcal{F}_{\text{AUTH}}$ or $\mathcal{F}_{\text{SYN}}$, the above generic transformation would modify the ideal functionality (e.g. $\mathcal{F}_{\text{AUTH}}$ or $\mathcal{F}_{\text{SYN}}$) so that it will interact with $\mathcal{F}_{\text{NC}}$ instead of interacting with the adversary.

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 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

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

1. Upon invocation, expect the input to include a session ID s. Report s to the adversary.
2. When receiving message `(Start,`$\hat{\mathcal{A}}$`)` 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`.
3. When receiving input `(Backdoor,`m`)` from party (i.e., ITI) P, verify that the state is `running`, and that P is a member of session s of π. If verification succeeds then activate $\hat{\mathcal{A}}$ with backdoor message m from P. Then:
   (a) If $\hat{\mathcal{A}}$ instructs to deliver a backdoor message m' to party P' then verify that P' is a member of session s of π. If so, then output `(Backdoor,`m'`)` to P'.
   (b) If $\hat{\mathcal{A}}$ generates an output v to the environment, then write v to the backdoor tape of the external adversary and halt.

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

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 of the [C00] notion, 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; furthermore, it ignores all outputs from the adversary other than the first one. Then:

Definition 24 *Let π and ϕ be subroutine respecting and subroutine exposing 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, balanced PPT environment $\mathcal{E}$, we have $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}} \approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$.*

We argue (informally) that NC-emulation captures the essence of the notion of [C00]. In particular, we conjecture that the existing security analysis of known protocols (e.g., the protocol of [GMW87], see [G04]) for realizing a general class of ideal functionalities with any number of faults, assuming authenticated communication as the only set-up assumption, is essentially tantamount to demonstrating that these protocols NC-emulate the corresponding ideal functionalities. This stands in contrast to the impossibility results regarding the realizability of the same functionalities in the unconstrained UC framework, even with authenticated communication.

Formally, what we show is that, for any protocol π and task ϕ, considering whether π NC-emulates ϕ is the same as considering whether the transformed protocol π_{NC} UC-emulates ϕ:

Proposition 25 *Let π and ϕ be subroutine respecting, PPT protocols. Then π_{NC} UC-emulates ϕ if and only if π NC-emulates ϕ.*

Notice that Proposition 25, together with the UC theorem, provide an alternative (albeit somewhat indirect) formulation of the non-concurrent composition theorem of [C00]. In fact, the present result is significantly more general, since it applies also to reactive protocols with multiple rounds of inputs and outputs.

Proof: We first show that if π_{NC} UC-emulates ϕ then π NC-emulates ϕ. Let $\mathcal{A}$ be an adversary (geared towards interacting with π in a non-concurrent environment). We need to show a simulator $\mathcal{S}_{\mathcal{A}}$ such that $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}} \approx \text{EXEC}_{\phi,\mathcal{S}_{\mathcal{A}},\mathcal{E}}$ for any non-concurrent environment $\mathcal{E}$.

We construct $\mathcal{S}_{\mathcal{A}}$ in two steps. First, we consider the adversary $\hat{\mathcal{A}}$ which is the version of $\mathcal{A}$ geared towards working with π_{NC}. That is, upon receiving the first input from the environment, $\hat{\mathcal{A}}$ sends the code $\mathcal{A}$ to π_{NC}, and from then on behaves like the dummy adversary. We have that, as long as $\mathcal{E}$ is non-concurrent, the ensembles $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$ and $\text{EXEC}_{\pi_{\text{NC}},\hat{\mathcal{A}},\mathcal{E}}$ are identical. We now let $\mathcal{S}_{\mathcal{A}}$ be the simulator for π_{NC} and $\hat{\mathcal{A}}$, that is $\mathcal{S}_{\mathcal{A}}$ is such that $\text{EXEC}_{\pi_{\text{NC}},\hat{\mathcal{A}},\mathcal{E}} \approx \text{EXEC}_{\phi,\mathcal{S}_{\mathcal{A}},\mathcal{E}}$ for all $\mathcal{E}$. This direction follows.

It remains to show that if π NC-emulates ϕ then π_{NC} UC-emulates ϕ. In fact, it suffices to demonstrate that π_{NC} UC-emulates ϕ with respect to the dummy adversary. Furthermore, we will use Claim 14 and only show that π_{NC} UC-emulates ϕ with respect to specialized simulators (i.e., when the simulator depends on the environment).

Let $\mathcal{E}$ be a general UC environment, and consider the environment $\mathcal{E}_{\text{NC}}$ that is identical to $\mathcal{E}$ except that $\mathcal{E}_{\text{NC}}$ passes to its own adversary only the first input that $\mathcal{E}$ provides to its adversary. All other values that $\mathcal{E}$ provides to its adversary are ignored. Furthermore, as soon as $\mathcal{E}_{\text{NC}}$ receives an output value from its adversary, it hands this value to $\mathcal{E}$, outputs whatever $\mathcal{E}$ outputs, and halts. The interaction between $\mathcal{E}_{\text{NC}}$ and the parties of π is the same as that of $\mathcal{E}$. Clearly, $\mathcal{E}_{\text{NC}}$ is a non-concurrent environment.

Next we interpret the first input that $\mathcal{E}$ provides its adversary as a program $\hat{\mathcal{A}}$ for an adversary, and conclude that there exists a simulator $\mathcal{S}$ such that $\text{EXEC}_{\pi,\hat{\mathcal{A}},\mathcal{E}_{\text{NC}}} \approx \text{EXEC}_{\phi,\mathcal{S},\mathcal{E}_{\text{NC}}}$. However, $\text{EXEC}_{\pi,\hat{\mathcal{A}},\mathcal{E}_{\text{NC}}}$ is identical to $\text{EXEC}_{\pi_{\text{NC}},\mathcal{D},\mathcal{E}}$, since the view of $\mathcal{E}$ is the same in the two executions. Similarly, consider the simulator $\hat{\mathcal{S}}$ that is identical to $\mathcal{S}$ except that $\hat{\mathcal{S}}$ ignores all inputs from its environment other than the first one, and withholds all outputs to the environment other than the very last one before halting. We then have that $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}_{\text{NC}}}$ is distributed identically to $\text{EXEC}_{\phi,\hat{\mathcal{S}},\mathcal{E}}$, since the view of $\mathcal{E}$ is the same in the two executions. We conclude that $\text{EXEC}_{\pi_{\text{NC}},\mathcal{D},\mathcal{E}} = \text{EXEC}_{\pi,\hat{\mathcal{A}},\mathcal{E}_{\text{NC}}} \approx \text{EXEC}_{\phi,\mathcal{S},\mathcal{E}_{\text{NC}}} = \text{EXEC}_{\phi,\hat{\mathcal{S}},\mathcal{E}}$. □

Modeling partial concurrency. Finally we remark that the methodology presented here can be extended to analyzing "partial concurrency" of protocols, where "partial concurrency" can come in multiple flavors. For instance, one can model *bounded concurrency* by allowing $\mathcal{F}_{\text{NC}}$ only limited number of interactions with the external adversary, or alternatively only a limited number of bits sent to (or received from) the external adversary. Alternatively, one can consider composite protocols where some components cannot be run concurrently to each other, but concurrency of other components (or of sub-components within a component) is allowed.

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 particularly grateful to Oded Goldreich who, as usual, gave me both essential moral support and invaluable technical, presentational, and practical advice. In particular, Oded's dedicated advice reshaped the presentation of this paper, especially Sections 1 and 2.

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, Rita Vald, Mayank Varia, 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. I am also extremely grateful to an anonymous referee who has meticulously read the paper over and over, pointing out scores of bugs - small and large.

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)

[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. . 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.

[B+91] R. Bird, I. S. Gopal, A. Herzberg, P. A. Janson, S. Kutten, R. Molva, M. Yung. Systematic Design of Two-Party Authentication Protocols. CRYPTO 1991: 44-61

[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.

[C00A] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. IACR ePrint Archive, Report 2000/067 and ECCC report TR01-016.

[C01] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. FOCS 2001: 136-145

[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] R. 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] R. Canetti. Security and composition of cryptographic protocols. Chapter in *Secure Multi-party 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

[CCL15] R. Canetti, A. Cohen, Y. Lindell. A Simpler Variant of Universally Composable Security for Standard Multiparty Computation. Crypto, 2015. See also IACR Cryptology ePrint Archive 2014: 553.

[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.

[CF01] R. Canetti and M. Fischlin. Universally Composable Commitments. Crypto '01, 2001.

[CG96] R. Canetti and R. Gennaro. Incoercible multi-party computation. *37th Symp. on Foundations of Computer Science (FOCS)*, IEEE, 1996.

[CGP15] R. Canetti, S. Goldwasser, O. Poburinnaya. Adaptively Secure Two-Party Computation from Indistinguishability Obfuscation. TCC (2) 2015: 557-585. See also IACR Eprint archive 2014: 845.

[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).

[CHMV17] R. Canetti, K. Hogan, A. Malhotra and M. Varia. Universally Composable Network Time. *Computer Security Foundations (CSF)* 2017.

[CJS14] R. Canetti, A. Jain, A. Scafuro. Practical UC security with a Global Random Oracle. ACM Conference on Computer and Communications Security 2014: 597-608

[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.

[CK02] R. Canetti and H. Krawczyk. Universally Composable Key Exchange and Secure Channels . In *Eurocrypt '02*, pages 337–351, 2002. LNCS No. 2332. Extended version at http://eprint.iacr.org/2002/059.

[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.

[CLOS02] R. Canetti, Y. Lindell, R. Ostrovsky, A. Sahai. Universally composable two-party and multi-party secure computation. *34th STOC*, pp. 494–503, 2002.

[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.

[CSV16] R. Canetti, D. Shahaf, M. Vald. Composable Authentication with Global PKI. PKC 2016. See also IACR Cryptology ePrint Archive 2014: 432.

[CV12] R. Canetti, M. Vald. Universally Composable Security with Local Adversaries. SCN 2012: 281-301. See also IACR Cryptology ePrint Archive 2012: 117.

[CM89] B. Chor, L. Moscovici. Solvability in Asynchronous Environments (Extended Abstract). FOCS 1989: 422-427

[CN99] B. Chor, L. Nelson. Solvability in Asynchronous Environments II: Finite Interactive Tasks. SIAM J. Comput. 29(2): 351-377 (1999)

[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.

[DKSW09] Y. Dodis, J. Katz, A. D. Smith, S. Walfish. Composability and On-Line Deniability of Authentication. TCC 2009: 146-162

[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).

[G08] O. Goldreich. *Computational complexity - a conceptual perspective*. Cambridge University Press 2008, ISBN 978-0-521-88473-0.

[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.

[GL01] O. Goldreich and Y. Lindell. Session-key generation using human passwords only. *CRYPTO '01*, 2001.

[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 multi-party 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. J. Cryptology 28(3): 423-508 (2015)

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

[K74] G. Kahn. The Semantics of Simple Language for Parallel Programming. IFIP Congress 1974: 471-475

[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. Küsters, M. Tüngerthal. 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.

[MT13] D. Micciancio and S. Tessaro. An equational approach to secure multi-party computation. ITCS 2013: 355-372

[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.

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

[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] A. 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.

[W05] D. Wikström. On the security of mix-nets and hierarchical group signatures *PhD Thesis*, KTH, 2005.

[W16] D. Wikström. Simplified Universal Composability Framework. TCC (A1) 2016: 566-595.

[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 that led to the present framework or directly affect it. This includes works that affect the first version (from December 2000), as well as works that influenced subsequent revisions. Still, we omit many works that use this framework, study it and extend it. The review sections in [C01, C06, C07, C08, C13] cover some of these works. For simplicity of exposition we mostly present the works in a rough chronological order rather than in thematic order. Also we concentrate on contributions to the definitional aspects cryptographic protocols rather than protocol design (although the two naturally go hand in hand).

Prior work. Two works that laid the foundations of general notions of security for cryptographic protocols are the work of Yao [Y82], which explicitly expressed the need for a general "unified" framework for expressing the security requirements of secure computation 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 UC framework 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 general definitional framework for secure protocols 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].

The first ideal-process based definition of computational security against resource bounded adversaries is given in [C95]. In [C00] the framework of [C95] is strengthened 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 any point in time. (See more details in Section 6.3.) In addition, security of protocols in that framework is shown to be preserved under modular composition. A closely related formulation appears in [G04, Section 7.7.2].

[C00] also sketches how to strengthen the definition there to support concurrent composition. The UC framework implements these sketches in a direct way.

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, *which is specific to the setting where the communication between parties is ideally private.* 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, it is not clear how to extend their definition and modeling to settings where the adversary has access to the communication between honest parties.

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.

The modeling of randomized distributed computation in an asynchronous, event-driven setting is an important component of this work. Works that considerably influenced the present modeling include Chor and Moscovici [CM89], Chor and Nelson [CN99], Bird et al. [B+91], and Canetti and

Krawczyk [CK01].

Concurrent work. 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. Their model provides a rich set of methods for scheduling events in an execution. Still, they postulate a static system where the number of participants and their identities are fixed in advance (this is somewhat similar to the model of Section 2 in this work). In particular, 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). Other technical differences include the notion of polynomial time computation (all entities are bounded by a fixed polynomial in the security parameter regardless of the input length - see discussion in Section 3.3.4), and scheduling of events.

Subequent work. 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 (still it is otherwise static as in [PW00]). 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.

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 process calculus for probabilistic polynomial time computation, 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]. We note that all of these frameworks postulate a static execution model which is most similar to the one in Section 2.

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.

Micciancio and Tessaro [MT13] provide an alternative, simplified formalism for composable

simulation-based security of protocol. The formalism, which is a generalization of Kahn networks [K74], allows for equational (rather than temporal) representation and analysis of protocols and their security.

Küsters and Tüngerthal [K06, KT13] formulate 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 an ITM-based model, allowing for succinct and clear presentation of composition theorems and proofs. In particular the accounting of the import of messages in this work is influenced by the modeling of [K06].

Hofheinz, Müller-Quade and Unruh [HMU09] give an alternative definition of polynomial time ITMs (see discussion in Section 3.3.4). 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 session IDs 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 21).

Nielsen [N03], Hofheinz and Müller-Quade [HM04a], and Katz et al. [KMTZ13] formulate synchronous variants of the UC framework. Wikström [W05, W16], as well as Canetti, Cohen and Lindell [CCL15] present simplified formulations of the UC framework, geared as simplifying the presentation and analysis of protocols in more "standard" multiparty computation settings.

Previous versions of this work. Finally, we note that the present framework has evolved considerably over the years; We highlight the main advances. (In addition, each revision corrects multiple inaccuracies and modeling discrepancies in previous versions. See more details in [C00A, Appendix B of Version of 2013].) The first versions of this work [C00A, Versions of 2000 and 2001] do not formulate a separate, rigorous model of distributed computation, and have different models for the execution of a protocol and for the ideal process. Also different communication and corruption models are treated as variants of the basic model.

The next revision [C00A, Version of 2005] introduces the notion of a system of ITMs and is the first to treat communication models as additional constructs on top of a single basic model, where the UC theorem is stated in the basic model. This version also moves from the restricted notion of "polynomial time in the security parameter" to a more expressive notion that takes into account input size, formally defines security with respect to the dummy adversary and demonstrates its equivalence with plain UC security, and presents a more detailed proof of the composition theorem.

The next revision [C00A, Version of 2013] is the first to treat corruption models as additional constructs on top of the basic model. It also provides an improved treatment of identities and the need to translate identities in the composition operation, simplifies the notion of polynomial time, and introduces the notions of balanced environments and subroutine respecting protocols.

The present version further improves the treatment of identities and subroutine respecting protocols, simplifies the definition of systems of ITIs and the model of protocol execution, and introduces the simplified, "static" model of computation of Section 2. More specifically, the main changes from the version of 2013 are summarized in Appendix B.

B The main differences from the 2013 version

We list the main differences from [C00A, Version of 2013]. Many of these changes are the result of comments of colleagues, including an anonymous referee to whom I am greatly thankful.

Changes to Section 3 (the model of computation):

- Renamed the incoming communication tape to be the backdoor tape.
- Allowed co-existence of ITIs with the same IDs, as long as their extended-IDs (i.e., their codes) are different.
- Added the reveal sender code and forced write parameters to the external-write operation.
- Introduced the notion of the import of messages as a basis for bounding the runtime and defining polynomial-time ITM.

Changes to Section 4 (protocol emulation and related claims):

- Changed the model of protocol execution as follows:
  - Allowed the adversary to be invoked at any time during the execution (rather than at the beginning).
  - Restricted the adversary to write only to the *backdoor tapes* of *existing* parties (i.e., without forced write).
- Introduced the notion of external identities and identity-bounded environments.
- Changed the definition of the dummy parties and dummy adversary to make sure they remains PT, and included a missing argument in the proof of the dummy adversary claim (Claim 11).

Changes to Section 5 (the composition operation and theorem):

- Added the notion of compliant (calling) protocols, and modified the UC theorem accordingly.
- Modified the definition of subroutine exposing protocols, introducing the concept of the directory ITI.

Changes to Section 6 (the modeling of party corruption and ideal functionalities):

- Modified the process of notification upon party corruption, using the directory ITI as a registry of corrupted ITIs.
- Modified the definition of $\mathcal{F}_{\text{AUTH}}$ so as to let the receiver know the *extended* identity of the sender.
- Modified the definition of $\mathcal{F}_{\text{SYN}}$ so as to guarantee that communication rounds always advance.