Universally Composable Security

RAN CANETTI, Boston University

This work presents 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 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 sessions 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.

CCS Concepts: • **Theory of computation → Computational complexity and cryptography**; **Cryptographic protocols**; *Concurrency*; **Distributed computing models**; **Process calculi**; • **Security and privacy → Formal methods and theory of security**;

Additional Key Words and Phrases: Security modeling, specification, and analysis, security-preserving composition, universal composition, modular security

ACM Reference format:
Ran Canetti. 2020. Universally Composable Security. *J. ACM* 67, 5, Article 28 (September 2020), 94 pages.
https://doi.org/10.1145/3402457

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, devising 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

An extended abstract of this work appears in the *Proceedings of the 42nd Foundations of Computer Science Conference (FCS'01)*. Previous versions have appeared as IACR Eprint archive 2000/067 and ECCC TR 01-16 under the title "Universally Composable Security: A New Paradigm for Cryptographic Protocols."
Member of CPIIS. Supported by NSF Grants No. 1414119 and No. 1801564.

Author's address: R. Canetti, Department of Computer Science, Boston University, 111 Cummington Mall, Boston MA 02215, USA; email: canetti@bu.edu.

0004-5411/2020/09-ART28 $15.00
https://doi.org/10.1145/3402457

Corrected Version of Record v.1.1. Published October 14, 2020

well as the plethora of prevalent design techniques for distributed systems 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 of security that allow for modular design of cryptographic protocols and applications from simpler building blocks in a way that guarantees overall security. Addressing these challenges is the focal point of this work.

Initial definitions of security for specific cryptographic tasks (e.g., References [70, 71]) considered models that 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 were insufficient when used within contexts and applications of interest. Examples include: Encryption, where the basic notion of semantic security [70] was later augmented with several flavors of security against chosen ciphertext attacks [14, 55, 102, 113] to address general protocol settings; Commitment, where the original notions were later augmented with some flavors of non-malleability [52, 55, 58] and equivocation [13, 22] to address the requirement of some applications; Zero-Knowledge protocols, where the original notions [71] were shown not to be closed under composition, and new notions and constructions were needed [9, 56, 65, 68, 114]; Key Exchange, where the original notions allow protocols that fail to provide secure communication, even when combined with secure symmetric encryption and authentication protocols [15, 41, 66, 116]; Oblivious Transfer [57, 59, 112] and secure multiparty function evaluation [12, 23, 54, 63, 69, 96, 108], 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 [15, 41, 66], non-malleable commitments [55], concurrent zero-knowledge [56], and general concurrently secure protocols [11, 105], where the definitions explicitly model several adversarially coordinated sessions 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 session of the protocol *"in vitro."* Security *"in vivo,"* namely, in more realistic settings where a protocol session 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 that have not been explicitly considered.

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., References [12, 16, 23, 54, 69, 73, 96, 107, 108, 110]. These definitions are obvious candidates for such a general framework. However, 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 sessions may be running concurrently in an adversarially coordinated way. These works and their relation to the present one are further elaborated on in Appendix A.

This work proposes yet another framework for representing and analyzing the security of cryptographic protocols. Within this framework, it formulates a general methodology for expressing the security requirements of cryptographic tasks. Furthermore, it defines a 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 shall be seen, 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, this implies significantly stronger and more general variants of some standard security concerns, such as non-malleability and security under concurrent composition: Here security is preserved even with respect to an unbounded number of sessions 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 [25]. Many of these works are influenced by this work, and many of them influenced later versions of this work, this one included. Here let us mention only References [6, 33, 34, 77, 84, 86, 94, 97, 104, 111, 119]. 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 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

This section briefly sketches 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 it goes back to the seminal work of Goldreich, Micali, and Wigderson [67]: 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, the definitional framework of [23] is briefly sketched first. This framework provides a basic

[1] This work uses 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., Reference [63, Section 7.7.2]) the terminology is decoupled, with security being called *environmental security* and composition being articulated as concurrent.

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 Reference [63, Section 7.5.1].) Throughout the Introduction, these elements are referred to as machines.[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 machines 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 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 session 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 sessions that run concurrently are subject to a coordinated attack against them.

Indeed, there are natural protocols that meet the [23] definition but are insecure when as few as *two* sessions are active at the same time and subject to a coordinated attack against them. See References [23, 27] 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 References [27, 30].

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.” To better understand the new formulation, an alternative and equivalent of the formulation in [23] is first presented. 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*

[2] Formally, these elements are modeled as interactive Turing machines (ITMs). However, 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.

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 definition in Reference [23], where the set of corrupted parties is fixed in advance. In the case of adaptive corruption, the definition in Reference [23] allows some additional interaction between the environment and the protocol at the event of corrupting a party.) The main difference between the UC framework and the basic framework of Reference [23] 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 the "flow of information" between the protocol session 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-session concurrent execution environment. (See some concrete examples in Reference [27].)

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: 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 Reference [23] 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 machine, 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. 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., Reference [63, Section 7.7.1.3] or Section 7.4). Other differences from Reference [23], such as capturing different communication models and the ability to dynamically generate programs, are discussed in later sections.

1.2 Universal Composition

As mentioned earlier on, the universal composition operation can be thought of as a natural extension of the "subroutine substitution" operation from the context of sequential algorithms to the context of distributed protocols. Specifically, consider a protocol ρ 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 a session of ρ typically involves multiple machines; this, in particular, means that a single session of $\mathcal{F}$ operates as a subroutine of multiple machines.)

Furthermore, the framework allows ρ to call *multiple sessions* of $\mathcal{F}$, and even have multiple sessions of $\mathcal{F}$ run concurrently. The framework also provides ρ with a general mechanism for "naming" the sessions of $\mathcal{F}$ to distinguish them from one another, but leave it up to ρ to decide on the actual naming method. The sessions 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 $\rho^{\mathcal{F}\rightarrow\pi}$ by starting with protocol ρ, and replacing each call to a session of $\mathcal{F}$ with a call to a session of π. Specifically, an input given to a session of $\mathcal{F}$ is now given to a machine in the corresponding session of π, and outputs of a machine in a session of π are treated by ρ as outputs obtained from the corresponding session of $\mathcal{F}$.[3]

The universal composition theorem states that running protocol $\rho^{\mathcal{F}\rightarrow\pi}$ is "at least as secure" 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 $\rho^{\mathcal{F}\rightarrow\pi}$, or with $\mathcal{S}$ and parties running ρ. In particular, if ρ UC-realizes some ideal functionality $\mathcal{G}$, then so does $\rho^{\mathcal{F}\rightarrow\pi}$.

Essentially, the universal composition theorem considers an environment that, together with the adversary, runs a "coordinated attack" against the various sessions of π, along with the "high-level part of $\rho^{\mathcal{F}\rightarrow\pi}$." The theorem guarantees that any such coordinated attack can be translated to an attack against the "high-level part of ρ" plus a set of attacks, where each such attack operates separately against a single session 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 sessions, where the composed sessions 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., References [36, 42, 44, 56]). A more detailed taxonomy and discussion appears in References [27, 30].

All these composition methods can be captured as special cases of universal composition. That is, any such method for composing together protocol sessions can be captured via an appropriate "calling protocol" ρ that uses the appropriate number of protocol sessions as subroutines, provides them with appropriately chosen inputs, and arranges for the appropriate synchronization

[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, one often needs 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.

in message delivery among the various subroutine sessions. 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 sessions have some "joint state" or, more generally, "joint subroutines." Said otherwise, this is the case where a single session of some protocol γ serves as a subroutine of two or more different sessions 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 session separately *in vitro*, and deduce the security of the overall system - in very much the same way as the traditional case where the sessions 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 sessions use the same shared reference string or same randomly chosen hash function. Universal composition in such situations was initially investigated in Reference [46], for the case of protocols that share subroutines only with other sessions of themselves, and in Reference [34] for the case of protocols that share subroutines with arbitrary, untrusted protocols. See further discussion at the end of Section 2.

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 References [23, 54, 96, 108, 109].) 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 sessions 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: 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 sessions 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

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 (correctness, 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 that 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. This comes 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 sessions." (Here, the term "machines" is used 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 machine, obtaining output from a subroutine machine, and a machine that is a subroutine of multiple machines.

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. The ideal process is then 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 of 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 machines" that are subroutines of two or more other machines, where the latter machines represent the communicating entities. Different communication models are then captured via different programs for the channel machines, 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 level to which the formal model captures dynamically changing systems and networks. The first, simplistic variant of model postulates 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 sessions" are fixed ahead of time.

While this modeling is indeed simpler to present and argue about, it is not amenable to capturing realistic settings where the number, the identities, the programs, and the connectivity of computational entities changes as the system progresses. 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 that allow for an adversarially controlled number of independent sessions of a simple protocol where the number of sessions is not known to the protocol.

The more general model provides built-in mechanisms for capturing such situations. In particular, it allows representing dynamically generated processes, with dynamically generated programs, identities, and communication patterns. The model also provides a natural way to delineate an "session of a protocol" even when an *a priori* unbounded number of processes join the session dynamically throughout the computation, without global coordination.

The latter 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.

Another choice relates to the level of formalism: While this work strives to pin down the details of the model as much as possible, it does not provide much in terms of syntax and a "programming language" for expressing protocols and their execution. Instead, it relies on the basic minimal syntax of Turing machines. Developing more formal and abstract domain-specific programming languages, that will facilitate mechanized representation and analysis of protocols and ideal functionalities within this framework, is left to future work.

Finally, it is noted that the models of computation presented here are different than the one in the first public version of this work [24, version of 2000]. Indeed, the model has evolved considerably over time, with the main steps being archived at Reference [24, later versions]. In addition, a number of works in the literature provide different tradeoffs between simplicity and expressibility, e.g., References [33, 104, 116, 118, 119]. 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 Article

The restricted 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 sessions, 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 an environment, as well as the general notions of protocol emulation. It also presents some variants of the basic definition of protocol emulation and asserts relationships among them.

Section 5 presents the concept of an ideal functionality, and defines what it means for a protocol to realize an ideal functionality.

Section 6 presents the universal composition operation, and then states and proves the universal composition theorem.

Section 7 exemplifies the use of the framework. It first proposes some conventions for expressing various party corruption operations. 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 RESTRICTED MODEL

This section presents a version of the definition of security and the composition theorem. Simplicity is obtained 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, this section also allows itself to be somewhat informal.

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 to be analyzed are *algorithms,* or *computer programs,* written for a distributed system. (The term *protocols* is often used 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 random inputs, and generates its own outputs. During their execution, the programs interact by transmitting information to each other. The rest of this subsection provides 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 we 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 an actual real-world entity (usually referred to as a "party" or "principal"): First, to enable the use of composition, we allow a "party" to consist of multiple machines, where some machines are treated as "subroutines" of other machines within that party. Second, the same construct (machines) is used 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. Machines can also be subroutines of multiple "caller machines," where some of these caller machines represent different real-world entities. This will be useful for capturing "multi-party" ideal functionalities. (For instance, the traditional notion of a

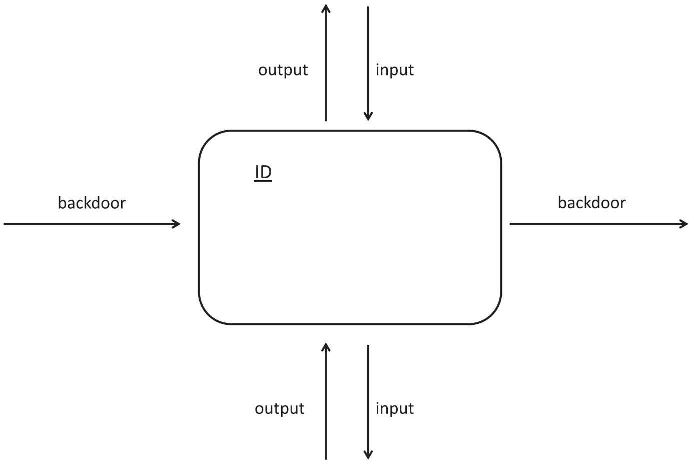

Fig. 1. The basic computing unit (machine). Each machine has an identity that remains unchanged throughout the computation. Information sent to other machines is labeled as either *input, output,* or *backdoor.* Information received from the outside (i.e., from other machines) is labeled consistently with the sender's labeling. Backdoor information is used to model information coming from and sent to the adversary. For graphical clarity, in future drawings, inputs are drawn as arrows facing downward, subroutine outputs as arrows facing upwards, and backdoor communication as arrows facing sideways. The communication set of the machine is not depicted.

communication channel between machines A and B will be captured via a "channel machine" that is a subroutine of both A and B.)

As for the formalism itself: With readability in mind, we almost completely refrain from setting syntax of machines. Still, the following constructs are defined. (Section 3 does propose some rudimentary syntax based on interactive Turing machine [61, 71], but any other reasonable model or syntax will do.)

First, each machine has a special value called the identity of the machine. The identity is visible to the machine during the computation, but remains unchanged throughout. Second, incoming information to a machine should be labelled either as input (representing inputs from a "calling machine") or as output (representing output from subroutines of the machine). A third form of incoming information, called backdoor, will be used to model information coming from the adversary (to be defined in Section 2.2).

Third, with each machine μ, a communication set is associatd, which lists the set of identities of machines that μ can send information to, including the type of information: input, 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 $L \in \{\text{input, output, backdoor}\}$. (Attention will soon be restricted to collections of machines where a machine μ can provide input to machine μ' if and only if μ' can provide 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, 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 1 for a graphical depiction of a machine.

Protocols. The next step is to define "multi-party protocols," namely, collections of programs that are designed for a joint goal, but are to be executed separately from each other "by different parties," while exchanging information. In our formalism a protocol will simply be a set of machines, with some minimal consistency requirements from the communication sets of these machines. (The reader is reminded that a machine models a computational module, and often does not correspond to the traditional concept of a "party" or "principal.")

We turn to defining when a collection of machines is called a protocol. To facilitate presenting the consistency requirements from the machines in a protocol, first formally define subroutines and callers: An identity ID is a *subroutine identity* for machine μ if the communication set of μ allows μ to provide input to identity ID (i.e., the communication set of μ contains the entry $\texttt{(input,ID)}$). Identity ID is a *caller identity* of μ if the communication set of μ allows μ to provide output to identity ID.

Now, consider a set of machines $\pi = (\mu_1, ..., \mu_n)$, where $\mu_i = (ID_i, C_i, \tilde{\mu}_i)$, $i = 1...n$. Machine $\mu_j \in \pi$ is a subroutine machine of machine $\mu_i \in \pi$ if ID_j is a subroutine identity of μ_i. Machine $\mu_i \in \pi$ is a caller machine of machine $\mu_j \in \pi$ if ID_i is a caller identity of μ_j. The set π is called a protocol if:

- No two machines in π have the same identity.
- For every two machines $\mu_i, \mu_j \in \pi$, μ_i is a subroutine of μ_j if and only if μ_j is a caller of μ_i.
- All the subroutine identities of all the machines in π correspond to actual machines in π. That is, if ID is a subroutine identity of some $\mu_i \in \pi$ then there exists $\mu_j \in \pi$ such that $ID = ID_j$.

Note that a protocol π may contain machines with caller identities that do not correspond to machines in π. If $\mu \in \pi$ has some caller identity ID, and no machine in π has identity ID, then say that μ is a main machine of π and ID is an external identity of π. The machines in π that are not main machines are called internal machines of π.

The formalism allows for "subroutine cycles": Say that machine $\mu_j \in \pi$ is a subsidiary of machine $\mu_i \in \pi$ if μ_j is a subroutine of μ_i or of another machine $\mu_k \in \pi$ that is a subsidiary of μ_i. Then protocols may contain machines that are subsidiaries (or even direct subroutines) of themselves. The reader is however cautioned that such "inherently non-hierarchical" protocols would not be conducive to modular analysis.

Using this terminology, translating the traditional notion of an m-party protocol into the present formalism results in a protocol that has m main machines, and potentially other internal machines. The internal machines that are subsidiaries of a single main machine represent modules within the program of the corresponding traditional party. Internal machines that are subsidiaries of more than one main machine naturally correspond either to shared physical resources (e.g., a communication links) of else to abstract ideal functionalities.

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. This approach is substantiated as follows.

First, formulate the process of executing a (potentially multi-party) protocol within an adversarial execution environment.

Next, as a preliminary step toward defining security, formulate a general notion of correspondence between protocols, called *protocol emulation.* This notion applies to any two protocols: Essentially, protocol π emulates protocol ϕ if π "successfully mimics the behavior of ϕ" within any execution environment. Said otherwise, if π emulates ϕ then, from the point of view of the rest of the system, interacting with π is "no worse" than interacting with ϕ.

Finally, define the ideal process for carrying out a distributed computational task by formulating a special "ideal protocol" for the task at hand. Then say that protocol π *securely realizes* a given task if π emulates the ideal protocol for the task.

2.2.1 Protocol Execution and Protocol Emulation. The formal model of protocol execution and the notion of protocol emulation is first presented in full, with minimal discussion. This is followed by a discussion of a number of aspects of the formalism, including how to represent network communication and corrupted parties on top of the basic model.

The model of execution for protocol π consists of the machines in π plus two additional machines, called the *environment* $\mathcal{E}$ and the *adversary* $\mathcal{A}$. The environment has identity 0, and its communication set allows it to provide inputs to $\mathcal{A}$ and to the *main machines* of π. The adversary $\mathcal{A}$ has identity 1, and its communication set allows it to provide backdoor information to all machines. (It is assumed that none of the identities of the machine in π, nor the external identities of the main machines in π, are 0 or 1.) The communication sets of the machines of π are augmented to include the ability to provide backdoor information to identity 1, namely, to $\mathcal{A}$. (As discussed in the Introduction, the environment and the adversary represent different aspects of the "rest of the system" and its interaction with π. The environment represents the interaction via inputs given to the protocol and outputs received from it, namely, via the "official channels." The adversary represent "side effects," namely, information leakage form the protocol execution and influence on it via other means of information transfer. Both machines are considered to be adversarial, in the sense that the definition of security will later quantify over all polynomial time $\mathcal{E}$ and $\mathcal{A}$.)

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 as follows: Once machine $\mu = (ID, C, \tilde{\mu})$ performs an instruction to transmit information to some identity $ID' \in C$, the execution of μ is suspended. Next:

(1) If $\mu = \mathcal{E}$, then the message (which in this case is an input) is added to the state of the machine μ' whose identity is ID' together with some source identity chosen by $\mathcal{E}$ out of the external identities of μ, along with the label `input`. If $\mu' = \mathcal{A}$, then no source identity is added. Next the execution of μ' begins (or resumes) until the point where μ' either pauses or instructs to transmit information to another machine.
(2) If $\mu \neq \mathcal{E}$ and the identity ID' exists in the system, then the message is added to the state of machine μ' whose identity is ID', along with the label and the source identity ID. Next μ' begins (or resumes) executing.
(3) Else (identity ID' is an external identity for π), the message is added to the state of $\mathcal{E}$, along with the identities ID and ID', and $\mathcal{E}$ resumes executing. (Observe that in this case μ is a main machine of π and the message is output.)
(4) If μ pauses without sending information, then the execution of $\mathcal{E}$ resumes.

The environment $\mathcal{E}$ is assumed to have a special binary *output* variable. The execution ends when the environment halts; The output of the execution is then the contents of the output variable of the environment. A graphical depiction of the model of protocol execution appears in Figure 2.

Let $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(z)$ denote the random variable (over the local random choices of all the involved machines) describing the output of 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 $\{\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(z)\}_{z\in\{0,1\}^*}$.

Protocol emulation. Next, define what it means for protocol π "emulate" 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 an adversary of choice, or with ϕ and some other adversary.

All machines are required to be polytime in the sense that the overall number of steps taken by each machine is bounded by a polynomial in a global security parameter, taken to be the length of the initial input to the environment. Jumping ahead, this will mean that an execution of any protocol can be simulated on a standard probabilistic Turing machine in time polynomial in the

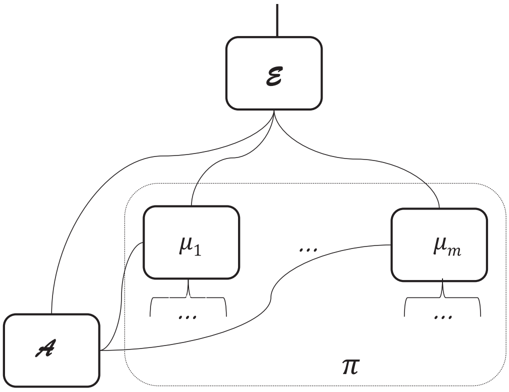

Fig. 2. The model of execution of protocol π. The environment $\mathcal{E}$ writes the inputs and reads the outputs of the main machines of π. In addition, $\mathcal{E}$ and $\mathcal{A}$ interact freely with each other. The main machines of π may have subroutines, to which $\mathcal{E}$ has no direct access.

security parameter and the number of machines. All other asymptotics are also taken over the security parameter.

Definition 1 (UC-emulation, restricted model). Protocol π UC-emulates protocol ϕ if for any polytime adversary $\mathcal{A}$ there exists a polytime adversary $\mathcal{S}$ such that, for any polytime 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.

Discussion. We discuss some aspects of the model of execution and the notion of protocol emulation. See additional discussion in Section 3.3.

On UC-emulation. UC-emulation guarantees strong correspondence between protocols. It requires the ability to turn any real-world attack on the emulating protocol (by $\mathcal{A}$) into an attack on the emulated protocols (by $\mathcal{S}$), so that the combined view of any environment from interacting with π on the input/output links, and at the same time interacting with π on the backdoor links (via $\mathcal{A}$) cannot be distinguished from its view of interacting with ϕ and $\mathcal{S}$.

This, in particular, means that the number of main machines in π, as well as their identities, are exactly the same as those of ϕ. In addition, it is guaranteed that outputs provided by π are distributed indistinguishably from those provided by 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 ϕ. When the output of ϕ is randomized, it is guaranteed that the joint distribution of the outputs of the main machines of π is indistinguishable from the joint distribution of their outputs of the main machines of ϕ; furthermore, this holds even when the outputs are viewed jointly with the output of $\mathcal{A}$ or $\mathcal{S}$, respectively.

As discussed in the Introduction, what sets this notion of emulation apart from previous ones is that the present notion considers an environment that takes an active role in trying to distinguish between the emulated protocol ϕ and the emulating protocol π. That is, the environment should be unable to distinguish between the process of running protocol π with adversary $\mathcal{A}$ and the process of running protocol ϕ with simulator $\mathcal{S}$ *even given the ability to interact with these processes as they evolve.* Indeed, the interactive nature of the definition plays a crucial role in the proof of the universal composition theorem.

Modeling 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 channel" between machines. Instead, communication channels 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 from machine μ_1 to machine μ_2. Machine $\mu_{[1,2]}$ proceeds as follows: (a) Upon receiving input m from machine μ_1, machine $\mu_{[1,2]}$ records m and sends m to $\mathcal{A}$ as backdoor information. (b) Upon receiving ok as backdoor, $\mu_{[1,2]}$ outputs m to μ_2. Indeed, the fact that $\mathcal{A}$ learns m means that the channel does not provide any secrecy guarantees; the ability of $\mathcal{A}$ to arbitrarily delay the delivery of the message represents the asynchrony of the communication; the fact that $\mu_{[1,2]}$ only delivers messages sent by μ_1 means that the communication is ideally authentic. Other types of communication are modeled analogously.

Arguably, 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 type of information the channel machine discloses to the adversary via the backdoor tape, and the way it responds to instructions coming from the adversary on the backdoor tape, determines the properties of the channel.

Modeling party corruption. The above model of protocol execution does not contain explicit constructs for representing adversarial behavior of parties (or, machines). As mentioned in Section 1.3, this too is done for sake of keeping the basic model and definition of security simple, while preserving generality and expressibility. It is now sketched 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 (dubbed *Byzantine*) behavior, honest-but-curious behavior, side-channel leakage, transient failures, coercion. For instance, adaptive Byzantine corruptions can be modeled by having the corruption instruction include a new program, and having the machine send its entire current state to the adversary, and from now on execute the new program 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 keep the definition of protocol emulation meaningful even in the presence of machine corruption, a mechanism will be needed that guarantees that the corruptions performed by $\mathcal{S}$ are "commensurate" to the corruptions performed by $\mathcal{A}$. One way to handle this issue is to postulate that the adversary (either $\mathcal{A}$ or $\mathcal{S}$) corrupts a machine only when specifically instructed to do so by the environment. However, this mechanism implicitly mandates that the internal machine structure of the emulating protocol π is identical to that of the emulated protocol ϕ, which is too restrictive (indeed, the analyst would typically like ϕ to be simpler than π).

Instead, a more general mechanism is devised, that allows the environment to obtain information regarding which machines are currently under control of the adversary: A special "record-keeping" machine is added to each protocol. This machine is notified by each machine of the protocol upon corruption. The environment can then learn about the current corruption activity by querying the special machine. The special machine can be programmed to disclose either full or partial information on the current corruption activity; determining which information to disclose to the environment should be viewed as part of the security specification of the protocol. See further discussion in Section 7.1.

A bit more generally, note that the above modeling of the traditional operation of "party corruption" heavily relies on using parts of the code of machines as "modeling pieces" rather than actual code that is to be executed in actual real-life machines. Indeed, this is a powerful modeling technique that allows keeping the model simple and at the same time flexible and expressive. However, to preserve meaningfulness one has to clearly delineate the separation between the "modeling code" and the "real code." Such delineation is formalized later on (Section 5.1) and used extensively.

On the order of activations and "true concurrency." Recall that the definition postulates a single-threaded and simplistic model of executing protocols: Only one machine is active at any point in time, and the next machine to be activated is determined exclusively by the currently active machine (subject to some basic model restrictions).

One might wonder whether this model adequately represents distributed systems where computation occurs concurrently in several locations. 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. It is also different than traditional mathematical modeling of concurrent executions of processes in distributed systems (see, e.g., References [1, 74, 87, 91, 92, 99]). Still, we argue that, in spite of its simplicity, the model does capture all salient situations and concerns. Let us elaborate.

Traditionally, executions of systems that include concurrent executions of physically separate processes are mathematically captured by first considering the un-ordered collection of the executions of the locally sequential processes, and then considering all possible ways to interleave the local executions to form a single "sequentialized" execution, subject to certain causality constraints. (This modeling is often dubbed "non-deterministic scheduling.") Here the granularity of "atomic events" (namely, events that are assumed to be executed at a single location and without interruption) is a key factor in determining the level of concurrency under consideration, and thus the expressive power of the model.

The model (or "mental experiment") considered here is simpler: Instead of determining the granularity of "atomic events" ahead of time, and then considering "all possible interleavings" of these atomic events, the present model lets the processes themselves determine both the granularity of atomic events (by deciding when to send a message to another machine) and the specific interleaving of events (by deciding which machine to send a message to). The result is a single-threaded sequential execution that is determined by the aggregate of the local decisions made algorithmically by each machine.

Observe however that the simplicity of the formalism does not restrict its expressive power: Indeed, it is possible to capture any granularity of atomic events, by programming the machines to send messages at the end of the desired atomic sequence of operations. Arbitrary and adversarial interleaving of events is captured by having the machines send messages to adversarial machines that represent the desired level of variability in timing and ordering of events (as exemplified by the modeling of asynchronous communication channels sketched in a previous comment). Furthermore, this modeling allows restricting attention to *computationally bounded* scheduling of events, as opposed to fully non-deterministic scheduling. This ability is crucial for capturing security guarantees that hold only against computationally bounded adversaries.

2.2.2 Realizing Deal Functionalities. Recall that security of protocols is defined by way of comparing the protocol execution to an *ideal process* for carrying out the task at hand, and that the ideal process takes the form of running a special protocol called the ideal protocol for the task. A key ingredient in the ideal protocol is the ideal functionality, which is a single machine that

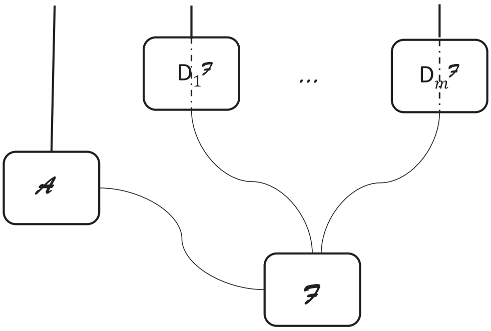

Fig. 3. The ideal protocol $\text{IDEAL}_{\mathcal{F}}$ for an ideal functionality $\mathcal{F}$. The main machines of $\text{IDEAL}_{\mathcal{F}}$, denoted $D_1^{\mathcal{F}}, \ldots, D_m^{\mathcal{F}}$, are "dummy parties": They only relay inputs to $\mathcal{F}$, and they relay outputs of $\mathcal{F}$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

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 the ideal functionality machine $\mathcal{F}$, 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 $\mathcal{F}$, along with a destination identity, a dummy party forwards the value to its destination.

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 $\text{IDEAL}_{\mathcal{F}}$.)

Defining when protocol π realizes an ideal functionality $\mathcal{F}$ is now straightforward:

Definition 2 (Realizing an Ideal Functionality, Restricted Model). Protocol π UC-realizes ideal functionality $\mathcal{F}$ if π UC-emulates IDEAL_F.

Discussion. We motivate some aspects of the design of ideal functionalities and ideal protocols:

Backdoor communication with the adversary. The backdoor communication between $\mathcal{F}$ and $\mathcal{A}$ provides a flexible and expressive way to "fine-tune" the security guarantees provided by $\mathcal{F}$. Indeed, as discussed in the Introduction, a natural way to represent tasks that allow some "disclosure of information" (say, via corruption or protocol messages) 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 (again, via corruption, message scheduling, or other means) can be represented by letting $\mathcal{F}$ take into account information received from $\mathcal{A}$.

Capturing stateful and reactive tasks. An ideal functionality 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.

The role of the dummy parties. At first glance, the dummy parties may look redundant: One could potentially have the ideal protocol consist of only $\mathcal{F}$: All inputs would be sent directly to $\mathcal{F}$, and $\mathcal{F}$ would directly send all outputs to their destination machines. However, in that case the ideal protocol would be easily distinguishable from any distributed implementation of it, since the environment could tell whether it is interaction with a single machine (in particular, a single identity) or else with multiple machines. Indeed, the role of the dummy parties is to allow the environment

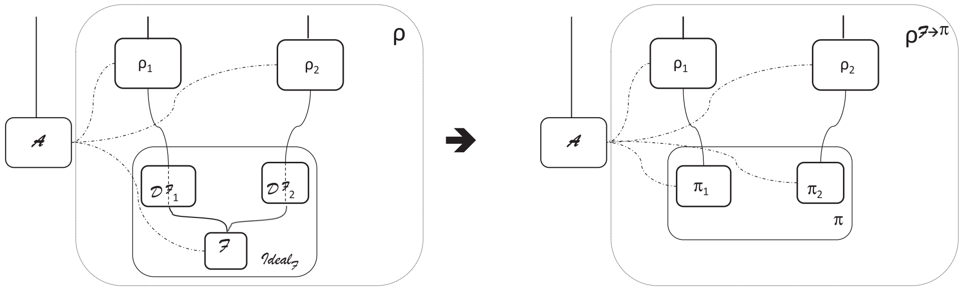

Fig. 4. The universal composition operation for the case where ρ has two main machines and uses an ideal protocol $\textsc{ideal}_{\mathcal{F}}$ with two main machines (left figure). $\textsc{ideal}_{\mathcal{F}}$ is replaced by a two-machine protocol π (right figure). The solid lines represent inputs and outputs. The dashed lines represent backdoor communication.

to treat the ideal process as a distributed process, which consists of multiple separate computational entities, each taking inputs and generating outputs separately from the other entities, and at the same time have the actual computational process be done centrally in $\mathcal{F}$. The dummy parties are not meant to play any other role. In particular, they ignore backdoor information. This means that the description of $\mathcal{F}$ captures all the specification information for the task at hand.

2.3 Universal Composition

This subsection presents the universal composition operation and theorem. It concentrates on the case of composing general protocols, noting that the case of ideal functionalities and ideal protocols follows as a special case.

Subroutine protocols. To present the composition operation, first define subroutine protocols. Let ρ be a protocol, and let $\phi \subset \rho$, namely, ϕ is a subset of the machines in ρ. Say that ϕ is a subroutine protocol of ρ if ϕ is a valid protocol when considered as a set of machines in and of itself. In this case, call the set $\rho \setminus \phi$ the caller part of ρ with respect to ϕ. It is stressed that ϕ may contain some of the main machines of ρ. (In fact, ρ is a subroutine of itself.)

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, say that protocol π is compatible with protocol ϕ if there is an identity-preserving injective correspondence between the main machines of π and those of ϕ. Furthermore, the external identities in the communication set C of each main machine of π, namely, the identities in C that are not part of π, appear also in the communication set of the corresponding main machine of ϕ.

Let ρ, ϕ and π be protocols such that ϕ is a subroutine protocol of ρ, π and ϕ are compatible, and no machine in π has the same identity as any machine in $\rho \setminus \phi$. The composed protocol, denoted $\rho^{\phi\rightarrow\pi}$, is identical to ρ, except that the subroutine protocol ϕ is replaced by protocol π. Given the interpretation of protocols as sets of machines, it holds that $\rho^{\phi\rightarrow\pi} = (\rho \setminus \phi) \cup \pi$. Since π is compatible with ϕ, it holds that $\rho^{\phi\rightarrow\pi}$ is a valid protocol. For notational simplicity, $\rho^{\mathcal{F}\rightarrow\pi}$ is used instead of $\rho^{(\text{IDEAL}_{\mathcal{F}})\rightarrow\pi}$. Figure 4 presents a graphical depiction of the composition operation:

The composition theorem. Say that protocol π is identity-compatible with protocols ρ and ϕ if no machine in π has the same identity as a machine in $\rho \setminus \phi$. Then:

Theorem 3 (universal composition for the restricted 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 $\rho^{\phi\rightarrow\pi}$ UC-emulates ρ.*

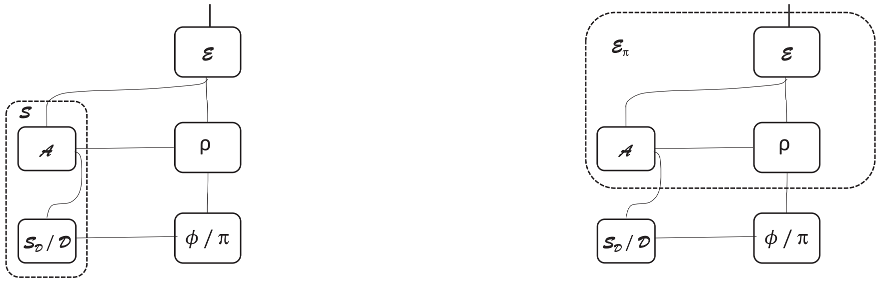

Fig. 5. The operation of the simulator $\mathcal{S}$ (left figure) and of the environment $\mathcal{E}_\pi$ (right figure). For visual clarity, each (potentially multi-party) protocol is depicted as a single box.

Discussion. See Section 1.2 for interpretation and discussion of universal composition. The following corollaries hold: (a) If protocol π UC-realizes an ideal functionality $\mathcal{F}$, and ρ uses as subroutine protocol IDEAL$_\mathcal{F}$ (i.e., the ideal protocol for $\mathcal{F}$), then the composed protocol $\rho^{\mathcal{F}\rightarrow\pi}$ UC-emulates ρ. (b) If ρ UC-realizes an ideal functionality $\mathcal{G}$, then so does $\rho^{\mathcal{F}\rightarrow\pi}$.

PROOF (SKETCH): The main observation that underlies the proof is that, since UC-emulation allows unrestricted exchange of information between the environment and the adversary, the interaction between $\mathcal{A}$ and $\rho^{\phi\rightarrow\pi}$ can be separated out to two distinct (interleaved) interactions: An interaction with π, and an interaction with the caller part of $\rho^{\phi\rightarrow\pi}$ with respect to π. Simulating the overall interaction can then be done by separately simulating each one of the two interleaved interactions: The first interaction is simulatable because π UC-emulates ϕ, and the second interaction is trivially simulatable, because the caller part of $\rho^{\phi\rightarrow\pi}$ with respect to π is identical to the caller part of ρ with respect to ϕ.

To make use of this observation, first define a special adversary, called the dummy adversary, that merely serves as a "transparent channel" between $\mathcal{E}$ and the protocol. That is, $\mathcal{D}$ expects to receive in each input a request to deliver a given message to (the backdoor tape of) a given machine. $\mathcal{D}$ carries out these requests. In addition, any incoming backdoor message (from some protocol machine) is forwarded by $\mathcal{D}$ to its environment, along with the identity of the sending machine. Note that $\mathcal{D}$ is stateless and, in particular, "separable" to many independent adversaries.

The proof of the theorem now proceeds as follows (see Figure 5). Since π UC-emulates ϕ, it follows that there exists an adversary ("simulator") $\mathcal{S}_{\pi,\mathcal{D}}$, such that no environment can tell whether it is interacting with π and $\mathcal{D}$ or with ϕ and $\mathcal{S}_{\pi,\mathcal{D}}$. Given an adversary $\mathcal{A}$ (that is geared to interact with $\rho^{\phi\rightarrow\pi}$), construct the following simulator $\mathcal{S}$ (that is geared to interact with ρ). Simulator $\mathcal{S}$ runs $\mathcal{A}$ and channels the communication between $\mathcal{A}$ and the environment without any change. Similarly, the communication between $\mathcal{A}$ and the caller part of ρ is channeled without change. The communication between $\mathcal{A}$ and the machines of ϕ is "pipelined" via $\mathcal{S}_{\pi,\mathcal{D}}$; that is, messages generated by $\mathcal{A}$ to the machines of π are forwarded to $\mathcal{S}_{\pi,\mathcal{D}}$ as inputs from the environment; incoming messages from the machines of ϕ are forwarded to $\mathcal{S}_{\pi,\mathcal{D}}$ without change. Messages generated by $\mathcal{S}_{\pi,\mathcal{D}}$ to the machines of ϕ are forwarded without change, and outputs of $\mathcal{S}_{\pi,\mathcal{D}}$ to its environment are forwarded to $\mathcal{A}$ as messages coming from the machines of π.

Intuitively, the simulation makes sense, since the instance of $\mathcal{A}$ run by $\mathcal{S}$ behaves exactly like the environment that $\mathcal{S}_{\pi,\mathcal{D}}$ expects. More concretely, the validity of the simulation is demonstrated via a reduction to the validity of $\mathcal{S}_{\pi,\mathcal{D}}$: Given an environment $\mathcal{E}$ that distinguishes between an execution of ρ with $\mathcal{A}$, and an execution of $\rho^{\phi\rightarrow\pi}$ with $\mathcal{S}$, construct an environment $\mathcal{E}_\pi$ that distinguishes between an execution of π with $\mathcal{D}$ and an execution of ϕ with $\mathcal{S}_{\pi,\mathcal{D}}$. 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. It follows that if $\mathcal{E}_\pi$ interacts with π, then $\mathcal{E}$, run by $\mathcal{E}_\pi$, is in effect interacting with $\mathcal{D}$ and $\rho^{\phi\rightarrow\pi}$. Similarly, if $\mathcal{E}_\pi$ interacts with an session of ϕ, then $\mathcal{E}$ is in effect interacting with ρ and $\mathcal{S}$ (see Figure 5). Here, it is crucial 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 6.)

Note that $\mathcal{D}$ is in fact the "hardest adversary to simulate," in the following sense: If, for some protocols π and ϕ, it is possible to successfully simulate the dummy adversary for any environment, then it is possible to successfully simulate *any* polytime adversary, for any environment. 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. □

Finally, note that the UC theorem can be applied repeatedly to substitute multiple subroutine protocols of ρ with protocols that UC-realize them. Furthermore, repeated applications of the theorem may use nested subroutine protocols. Also, note that the simulation overhead is additive, namely, the runtime of $\mathcal{S}$ is bounded by the runtime of $\mathcal{A}$ *plus* some constant times the runtime of $\mathcal{S}_{\pi,\mathcal{D}}$. This means that the UC theorem cam be applied polynomially many times while keeping the simulation overhead polynomial.

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. While helpful in keeping the model simple, these restrictions do not allow representing many realistic situations, protocols, and threats.

The coming four sections extend the treatment of Section 2 to account for fully dynamic and evolving distributed systems of computational entities. This section extends the definition of machines and protocols from Section 2.1 and defines a general model of distributed computation within dynamically evolving systems, taking into account computational limitations. Building on the basic model developed here, Sections 4 and 5 then extend the definition of protocol emulation and realizing ideal functionalities from Section 2.2. Section 6 then extends Section 2.3.

Formulating a general model of computation, separately from the process of protocol execution, which underlies the notion of protocol emulation, is, in a way, a non-essential "detour." Still, we hope that this detour will help clarify and motivate the definitional choices made along the way. Furthermore, this general model may be of independent interest. In particular, it may potentially be used as a basis for different notions of security and correctness of distributed computation.

This section strives 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 [94, 100] 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, longer discussions and comparisons with other models are postponed to Section 3.3.

3.1 The Basic Model

As in Section 2, this section starts 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 the identities, connectivity, and even programs of computing elements are 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. Note that many of the choices here are new, and require re-thinking of basic concepts such as addressing of messages, identities, "protocol sessions," and resource-bounded computation.

The presentation proceeds in two main steps. Section 3.1.1 first defines a *syntax,* or a rudimentary "programming language" for protocols. This language, which extends the notion of *interactive Turing machine* [71], contains data structures and instructions needed for operating in a distributed system. Section 3.1.2 next defines the *semantics* of a protocol, namely, an execution model for distributed systems that consist of one or more protocols as sketched above. To facilitate readability, most of the motivating discussion is postponed to Section 3.3. The relevant parts of the discussion are pointed out along the way.

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 toward capturing *pairs* of interacting machines, is given in Reference [71] (see also Reference [61, Volume I, Chapter 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, this formalism is extended to accommodate protocols written for systems with *multiple* computing elements, and where multiple concurrent executions of various protocols co-exist. For this purpose, a somewhat richer syntax for ITMs is defined. The semantics of the added syntax are described as part of the model of execution of systems of ITMs, in Section 3.1.2.

Note that the formalism given below aims to use only minimal syntax programming abstractions. 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 3.3.1.

Definition 4. An interactive Turing machine (ITM) μ is a multitape[4] Turing machine (as in, say, Reference [117]) 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). This description is called 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.

[4] Recall that multitape Turing machines have multiple work tapes, each equipped with its own head. This variant of the model has the advantage that it allows combining two (or more) Turing machines into a single one that runs both machines, in an arbitrary interleaving of the operations, and with runtime and space that is essentially the sum of the individual runtimes and space requirements, plus some small additive overhead. Note that the Circuit, RAM or PRAM models have this compositionality property as well. See more discussion in Section 3.3.1.

(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, assume that each message ends with a special end-of-message (eom) character.) [5]

Definition 5. 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 in each tape. 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.) 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 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).

[5] 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 3.3.1.

Throughout this work, an ITM instance (ITI) is treated as a run-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. As will be seen, 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. In the language of Section 2, the control function corresponds to the collection of all the communication sets of all the machines in the system; it is, however, significantly more general, providing greater flexibility, expressive power, and clarity in defining models of distributed computing. (See more discussion in Section 3.3.) We proceed to the formal description.

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

Executions of systems of ITMs. 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 code I followed by identity 0, the input tape contains the value x, and a sufficiently long random string is written on the random tape.[6] In accordance, the ITI $(I, 0)$ is called the initial ITI in this execution.

An execution ends when the initial ITI halts (that is, when a halting configuration of the initial ITI is reached). 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 that 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 subroutine-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

[6] 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.

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, say that M invoked M'.

(2) If $f = 0$, then M' is interpreted as a predicate P on extended identities. Let M'' be the first ITI (considered by order of invocation) such that $P(M'')$ holds. Then, the message m is delivered to M'' as in Step 1a. 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', say that M sends backdoor message x to M'. When M writes a value x onto the input tape of M', say that M passes input x to M'. When M' writes x to the subroutine-output tape of M, say that M' passes subroutine-output x (or simply outputs x) to M.

It will be convenient to formally define what it means for an ITI to reject an incoming message: Say that ITI M rejects (or, ignores) an incoming message if, after reading the message, M returns to its state prior to reading the message and ends its activation without further action.

Notation. Let $\textsc{out}_{I,C}(z)$ 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 ITIs in the system. Let $\textsc{out}_{I,C}$ denote the ensemble $\{\textsc{out}_{I,C}(z)\}_{z\in\{0,1\}^*}$.

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 identity 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 mechanisms for ITIs to communicate, while specifying the code and identity of the ITIs it transmits information to. It also provides mechanisms for an ITI to know the identity (and sometimes the code) of the ITIs that transmitted information to it. These constructs provide a basic, yet sufficient formalism for representing distributed computational systems.

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 session with a session 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 used 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.* The definition of a control function is extended so that it can also *modify* the external-write requests made by ITIs. 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.

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 external-write operations only in very specific cases and in very limited ways.

Subroutines, etc. If M has passed input to M' in an execution and M' has not rejected this input, or M' has passed subroutine-output to M and M has not rejected this subroutine-output, then say that M' is a subroutine of 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 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 4. As discussed, the goal is to capture an *algorithm* written for a distributed system where physically separated participants engage in a joint computation; namely, the ITM is a static object that 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 formalism would add unnecessary notational complexity to the basic model.)

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

Traditionally, a session 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 sessions 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 sessions 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 session 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 a session typically run 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 session intends that ITI participates in a joint interaction with the other ITIs in that session."

The frameworks also provides a mechanism for an invoker of an ITI to specify a protocol session for the ITI to participate in. The construct used for this purpose is the identity string. That is, interpret (via some standard unambiguous encoding) the identity of an ITI as *two* strings, called the session identifier (SID) and the party identifier (PID). The idea is that a protocol session will consist of all the ITIs that have the same SID and the same code. The PIDs are used to differentiate between ITIs within a protocol session; they can also be used to associate ITIs with "clusters," such as physical computers in a network. Additional discussion on the SID/PID mechanism appears in Section 3.3.1.

We coin some additional terminology. Consider some execution prefix of a system of ITMs. A session s of protocol π in that prefix is the set of ITIs that have code π and SID s. (The shorthand protocol session (s, π) is also used with the same meaning.) The ITIs in protocol session (s, π) are called the parties of that session. The terms main parties or main ITIs of the protocol session are also used with the same meaning.

The extended session s of protocol π (or, extended session (σ, π)) in an execution prefix is defined inductively by inspecting the ITIs in this prefix by order of invocation: Each newly invoked ITI M is added to the extended session if (a) M is a main party of this session, or (b) M is a subroutine of an ITI that is already in the extended session, or (c) M was invoked by an ITI that is already a sub-party of this extended session. An ITI is a sub-party of an extended session if it is in the extended session but is not a main party of the session. (Informally, the extended session (s, π) includes the transitive closure of the invocation relation rooted at the main ITIs of the session, if we disregard invocations made by the main ITIs via their subroutine-output tapes.)[7]

Comparison with the modeling of Section 2.1. Recall that in Section 2.1 protocols are defined differently: There, a protocol is a set of machines, with restriction that all subsidiaries of a machine in a protocol should also be machines in the same protocol. However, this simple approach is no longer meaningful in the present model where ITIs are created dynamically, and furthermore it may not be known at the time of creating an ITI μ whether μ will later become a subsidiary of another machine μ'. We are thus forced to define protocol sessions and extended sessions in a more "myopic" way that is consistent with the dynamic evolution of the system. Still, observe that the *main parties* of a protocol in Section 2.1 roughly correspond to the main parties of the protocol here. The prsent definitions are further discussed and motivated in Section 3.3.2.

3.2 Polynomial Time ITMs and Parameterized Systems

This subsection adapts 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 sessions 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 will be seen, 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, 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, one would like to examine the number of steps required as a function of the complexity of

[7] In previous versions of this work the extended session was defined differently—it was the set of main parties of the session and their subsidiaries. The present definition simplifies the formalism in situations where one would like to include in the extended session of a protocol also ITIs that are not subsidiaries of the main parties of the session. See further discussion in Sections 5.2 and 7.3.1.

the input, often interpreted as the input length. Another purpose is bounding the power of *feasible computation,* often for the purpose of security. Here it typically does not matter whether the computation is using "asymptotically efficient algorithms"; one is only concerned with what can be done within the given resource bounds.

At first glance, it appears that for security one should be only interested in the second interpretation. However, recall that to argue security one often provides an algorithmic reduction that translates an attacker against the scheme in question to an attacker against some underlying construct that is assumed to be secure. This reduction should be efficient *in the former, algorithmic sense.* Furthermore, the very definition of security, formulated later, will require presenting an *efficient transformation* (namely, a simulator) from one feasible computation to another. In conclusion, a good model should capture both interpretations.

Let $T : \mathbf{N} \rightarrow \mathbf{N}$. 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 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-bounded, consumes an unbounded number of resources. This is so since ITIs may send each other messages of repeatedly increasing lengths. In Reference [71] this problem was solved by setting n to be the length of the input only. Indeed, in the setting of Reference [71], 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-bounded. Furthermore, infinite "chains" of ITIs can be created, where each ITI in the chain invokes the next one, again causing potentially infinite runs.

This "infinite runs" problem is prevented via the following simple mechanism. Each message is expected 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. Now, define the run-time 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.* An ITI is T-bounded if, at any configuration, the number of steps it took, since invocation is at most $T(n)$. As will be seen, 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 that 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 run-time. 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 6 (T-bounded, PPT). Recall that the import of a message is the value written in the import field of the message. Let $T : \mathbf{N} \rightarrow \mathbf{N}$. 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.

T-bounded ITMs are defined inductively: A locally T-bounded ITM that does not make external-writes with forced-write is T-bounded. In addition, a locally T-bounded ITM, all of whose external-writes with forced-write specify a recipient ITM that is T-bounded, is T-bounded as well. ITM 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 6 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 run-time." 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 is not an input message.

Second, here the import is represented in binary, instead of the traditional "length of message" convention. This allows the import of a message to be separate from its length and the amount of information it contains. Furthermore, it provides additional flexibility in distributing run-time among multiple subroutines. (For instance, consider the following two situations: In one situation, consider 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 $m \ll n$. 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. Section 3.3.4 infomally discusses 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 $T : \mathbf{N} \rightarrow \mathbf{N}$ is super-additive if $T(n + n') \geq T(n) + T(n')$ for all n, n'. We have:

Proposition 7. *Let $T : \mathbf{N} \rightarrow \mathbf{N}$ 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 the control function does not increase the import in any external-write request, and 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. First observe 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 resource messages as providing "tokens" that allow run-time. 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, prior to 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, it holds 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 $i = \sum_{M\in m_i}(\text{\# steps taken by } M) \le \sum_{M\in m_i} T(n_{M,i})$. By super-additivity of T, it holds that $i \le \sum_{M\in m_i} T(n_{M,i}) \le T(\sum_{M\in m_i} n_{M,i})$. However, $\sum_{M\in m_i} n_{M,i} \le n$. Thus, $i \le T(n)$.

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. Since T is super-additive and the control function does not increase the imports specified in external-write instructions, the overall number of steps taken by μ is at most $T(n)$, plus 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. □

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 simplify the formalization of security properties and, in particular, to allow for an asymptotic treatment based on a single parameter that tends to infinity, we will want to bound the variability in the computing powers of different ITMs. To do that, we will restrict attention to systems where there is a common value, called the security parameter, that serves as a minimum value for the initial run-time budget of each ITI. More specifically, say that an ITM is parameterized with security parameter k if it does not start running unless its overall import is at least k. (Can think of k as being part of the code of the machine, i.e., written on the identity tape.) 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, and in addition 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.[8]

3.3 Discussion

This subsection further discuss the model of computation and compare it with other general models in the literature that are aimed at capturing distributed computation with concurrently running processes—some of which explicitly aim at modeling security of protocols. A very incomplete list of such models includes the CSP model of Hoare [74], the CCS model and π-calculus of Milner [99, 100] (that is based on the λ-calculus as its basic model of computation), the *spi*-calculus of Abadi and Gordon [1] (that is based on the π-calculus), the framework of Lincoln et al. [87] (that

[8] The restriction to parameterized systems is by no means essential for a sound treatment of security. In fact, it is somewhat restrictive in that it does not allow modeling natural situations where different computational entities have vastly different computational powers. We adopt it only for sake of simplicity and readability. Extending the present formulation to capture situations where there is no *a priori* bound on the relative computational powers of entities in a system is left for future work.

uses the functional representation of probabilistic polynomial time from [101]), the I/O automata of Merritt and Lynch [91], the probabilistic I/O automata of Lynch, Segala, and Vaandrager [92, 115], the Abstract Cryptography model of Maurer and Renner [94], and the equational approach of Micciancio and Tessaro [97]. (Other approaches are mentioned in the Appendix.)

The discussion is paritioned to four parts: The use of ITMs, the use of identities, the external-write mechanism, and the modeling of resource-bounded computation. 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 as a "thought experiment" as in functional languages (such as 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 side-effects that result from the physical aspects of the computation. 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 represents the complexity of computations is that it facilitates the modeling of adversarial, yet computationally bounded, *scheduling of events* in a distributed system.

Finally, our imperative formalism naturally allows for a 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 downside, 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 References [2, 32, 40, 98]. We leave this line of research for future work.

One technical caveat here is that the traditional, single-tape Turing machine model imposes a significant overhead on the natural operation of running a number of programs in an interleaved manner. Instead, it is preferable to work in a model that allows for expressing the natural operation of combining two (or more) sequential programs to a single sequential program that simply executes the two programs side by side, following some arbitrary schedule of interleaving the executions, and where the runtime and space of the combined program is the sum of the individual programs, plus perhaps some small additive factor. The treatment is thus restricted to the *multitape* Turing machine model (see, e.g., Reference [117]), or alternatively to the RAM or PRAM models.

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 of a new ITI 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—with sufficient information for identifying this process from other ones. Furthermore, it has been demonstrated that externally chosen, unique identities are essential for performing basic tasks such as broadcast and Byzantine agreement within a general distributed computing framework such as the present one [90].

Allowing the process that creates a new ITI to determine, in an algorithmic way, also the *code* of the newly invoked ITI is a direct reflection of prevalent practice (akin to “downloading an app”). It is also a powerful analytical tool. The protocol of Reference [8] is an example for how this ability can be meaningfully used, and then analyzed, within the present model.

Another consequence of the present formalism is that if protocol π intends to call another protocol π' as subroutine, where π' intends to call yet another subroutin eπ'', and so on, then the description of π must essentially include the description of π' and π'' and all other subroutines.

Note that preventing an ITI from modifying its own identity is done mainly to simplify the delineation of individual computational processes in a system. Indeed, no expressive power appears to be lost (indeed, ITIs can always invoke other ITIs with related identities and programs).

Using the code in the external write operation. Recall that the semantics of the external-write operation crucially depends on the code of the sending and recipient ITIs. This convention might appear a bit odd at first, since the code of the sender (respectively, the recipient) of a message sent over a network is typically not known to the recipient (repsectively, sender). Furthermore, 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. Also, verifying practically any interesting property of arbitrary code is bound to be impossible in general.

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 run-time is bounded, and so on. 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 will be seen, 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, note that such situations are indeed expressible within the present model—simply consider protocols that ignore a certain portion of the identity. (Note that other formalisms, such as the IITM model [84–86], *mandate* having part of the identity inaccessible to the program.)

Furthermore, the model allows multiple ITIs in an execution to have the same (non-extended) identity—as long as they have different programs. This again underlines the fact that (non-extended) identities need not be unique.

On the SID mechanism. The SID mechanism provides a relatively simple and flexible way to delineate individual protocol sessions in a dynamically changing distributed system. In particular, it allows capturing, within the formal model, the intuitive notion of "creating a session 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 session 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 sessions 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 sessions is to set all the protocol sessions 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 session have the same SID.

Another, more dynamic method for designing systems with multiple individual protocol sessions is to have each protocol session 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 session be created indigenously from within the protocol session 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 session (π, s) is created, by way of an incoming input that specifies code π and SID s. All the other ITIs of this session are created, with the same SID and code, by way of receiving communication from other ITIs in this session. (Jumping ahead, 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 7 are written in this manner.

Alternatively, one may wish to design a system where protocol sessions are created dynamically, but computational processes that make up a new protocol session are created "hierarchically" via inputs from existing processes rather than autonomously from within the protocol session itself. Here again the SID mechanism is a natural formalism. Indeed, if the existing processes (ITIs) have sufficient information to create new ITIs that have the same SID, then the creation of a new protocol session can be done without additional coordination. When this is not the case, additional coordination might be needed to agree on a common SID. See References [8, 10] for a protocol and more discussion of this situation.

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

One can also formulate alternative conventions regarding the delineation of protocol sessions. For example one may allow the SIDs of the parties in a protocol session 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 session. (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 session.) 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, note that other frameworks, such as Reference [77], put additional restrictions on the format of the SIDs. Specifically, in Reference [77] the SID of a protocol session is required to include the SID of the calling protocol session, 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, then 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 are 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 an abstract construct that corresponds to a name of a channel. These channel names are typically treated as static system parameters, in the sense that they are not mutable by the programs running in the system. They also provide pre-set matching between the sending process and the receiving one. Furthermore, sending information on a channel is often 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 does not 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 sessions that run in the systems are not known in advance.

Indeed, such situations are commonplace. Examples include open peer-to-peer protocols (such as, e.g., public blockchain systems), client-initiated interaction with a server where the server learns that the client exists only via a message of the protocol itself, and programs or software updates 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 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.[9]

Second, 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. 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

[9] Note 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.

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 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 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''.) 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 4 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 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.)

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 the context of Zero-Knowledge protocols [106].

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 can be captured within the present basic framework but using approptiate control functions [33, 47, 48, 85].

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 any other 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 (subject to C's runtime limitations, and without increasing the imports of delivered messages).

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 say that the notion is *efficiently recognizable.*

It can be readily seen that the notion of PPT in Definition 6 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. Note that other notions of PPT protocols, such as those in References [77, 79] 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 run-time 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 6, it is still possible that unbounded executions of systems even when all the ITMs are bounded. Definition 6 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 run-time 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 run-time 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 that 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, then 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. This it holds that whether I accepts depends on information that should not be "legitimately available" to I in a distributed system.)

Jumping ahead, 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 run-time 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 on Section 2 as well as in References [5, 6, 23, 32, 93, 108].

Bounding the run-time this way is simpler than the method used here. It also allows proving a proposition akin to Proposition 7. However, it has a number of drawbacks. First, it does not allow capturing algorithms and protocols that 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 is 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 is determined by an arbitrary PPT adversary, and thus cannot be bounded by any specific polynomial in the security parameter. Consequently, bounding the run-time 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.[10]

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 Reference [78]).

Another drawback of bounding the run-time 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 run-time can vary with the length of the input, it is possible to have a single PPT ITM U that can “simulate” the operation of *all* PPT ITMs, when given sufficiently long input. (As the name suggests, U will be the universal Turing machine that receives the description of the ITM to be simulated, plus sufficiently long input that allows completing the simulation.) This universality is at the heart of the notion of “feasible computation.” Also, this property turns out to be useful in gaining assurance in the validity of the definition of security, defined later in this work.

Bounding the run-time 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 4.3, some of the properties of the notion of security defined here no longer hold when the run-time of ITMs is bounded this way.[11]

[10] 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.

[11] 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

4 PROTOCOL EXECUTION AND UC-EMULATION

This section formulates the model of protocol execution, and then presents and studies the definition of UC-emulation, which will be a central tool in defining security of protocols. At a 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. To allow for a more in-depth study of several formulations and variants of protocol emulation, we defer formulating the notion of realizing an ideal functionality to the next section (Section 5).

Section 4.1 presents the model for protocol execution. Section 4.2 defines protocol emulation. Sections 4.3 and 4.4 present some 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 more expressive formalism of Section 3.1. As there, the model does not explicitly represent communication channels between machines, nor does it include an explicit provision for representing corrupted parties. Section 7 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.

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), it must also provide a mechanism for making inputs coming from the environment appear, to the recipient, as inputs coming from ITIs other than the environment.

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

The model. Given ITMs $\pi, \mathcal{E}, \mathcal{A}$, the model consists of the extended, parameterized system of ITMs $(\mathcal{E}, C_{\text{EXEC}}^{\pi,\mathcal{A}})$, where the initial ITM of the system is the environment $\mathcal{E}$, the input z to $\mathcal{E}$ represents some initial state of the actual environment in which the protocol execution takes place, and the control function $C_{\text{EXEC}}^{\pi,\mathcal{A}}$ is defined below and summarized in Figure 6.

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 SID. The control function sets the code of all these ITIs to π. $\mathcal{E}$ can also specify the extended identity of the writing ITI to values other than its own. The adversary is identified by having PID $\diamond$.

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.

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 SID. The control function sets the code of all these ITIs to π. $\mathcal{E}$ can also specify the code and extended identity of the writing ITI to values other than its own. In addition, $\mathcal{E}$ may pass inputs to $\mathcal{A}$, who is identified by having PID $\diamond$. The *test SID, s,* is the SID of the ITIs written to by $\mathcal{E}$. The *test session of* π is the session of π with SID s.

(2) $\mathcal{A}$ may write only to the *backdoor tapes* of *existing* ITIs. In particular, $\mathcal{A}$'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,* the recipient code must not be specified, and the import must be 0.)
In addition, these ITIs may pass inputs and subroutine-outputs to any ITI other than $\mathcal{A}$ and $\mathcal{E}$, with the following modification: If the writing ITI is a main ITI of the test session of π, the target tape is the subroutine-output tape, and the target ITI does not exist in the system, 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.

Fig. 6. A summary of the model for protocol execution.

That is, if the PID of the target ITI is $\diamond$, 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 SID of M' is different than the SID 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 $\diamond$, 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.

The session identifier of the ITIs written to by $\mathcal{E}$ is called the *test SID.* The *test session of* π is the set of all ITIs whose code is π and whose SID is the test SID.

The effect of external-writes made by $\mathcal{A}$. The control function allows the adversary $\mathcal{A}$ (i.e., the ITI with PID $\diamond$) 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.) Furthermore, these messages cannot have any import.[12]
>
> In addition, ITIs other than $\mathcal{E}$ and $\mathcal{A}$ may pass inputs and subroutine-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 does not exist in the system, then the value is written on the subroutine-output tape of $\mathcal{E}$, along with the target extended identity and sender identity (but not the sender code).

Let $\textsc{exec}_{\pi,\mathcal{A},\mathcal{E}}(z) \overset{\text{def}}{=} \textsc{out}_{\mathcal{E},C^{\pi,\mathcal{A}}_{\text{EXEC}}}(z)$.

Observe that the set of ITIs invoked in an execution of protocol π, other than $\mathcal{E}$ and $\mathcal{A}$, consist of a single extended session of π. Jumping ahead, note that this makes the model of protocol execution a bit of an over-simplification of a general execution environment. Section 5.2 bridges the gap, via the notion of *subroutine-respecting protocols.*

4.2 UC-emulation

This section formalizes the general notion of one protocol emulating another protocol, extending the definition of Section 2.2 to the present model. We start by setting language for expressing more nuanced variants of the concept of an *environment machine.* Specifically, we formulate the notions of *identity-bounded environments* and *balanced environments.*

External identities and identity-bounded environments. Recall that the model of protocol execution lets the environment determine the extended identity of the source of any input it sends to a main party of the protocol. We call these extended identities, determined by the environment, external identities. Allowing the environment to use external identities is indeed essential - this is how the main parties get inputs from and provide subroutine-outputs to external entities. Still, when designing protocols it will often be convenient to be able to represent natural situations where the protocol in question only needs to be analyzed where the external identities take some form. (For instance, this would allow the protocol designer to set some "reserved identities" that can be used by the protocol for subroutine ITIs. Furthermore, knowing that the identities used by the environment are only of a certain form might help design better protocols.)

This extra lenience is captured as follows. For a set ξ of identities, an environment is ξ-identity-bounded if it uses only external identities in ξ. More generally, the set of allowed identities can be determined dynamically depending on the execution so far. That is, ξ can be a (PT) predicate that takes as input an entire configuration of the system at the moment where the environment provides input to a protocol party, and determines whether to accept the source identity assumed by the environment.[13]

Balanced environments. To keep the notion of protocol emulation from being unnecessarily restrictive, the environment is required to satisfy some basic conditions regarding the relative imports given to the protocol ITIs and the adversary.

[12] Allowing only the environment to transfer import to the adversary serves to simplify the modeling and analysis of security. It also simplifies the proofs of structural results about the model such as Lemma 11 and Theorem 22. Crucially, these restrictions do not appear to affect the expressive power of the model.

[13] Jumping ahead, we note that the more restricted the predicate ξ, the easier it will be to prove that π emulates ϕ (for some given protocols π, ϕ), and the harder it will be to use this fact later on—specifically in the context of the composition theorem. Indeed, the decision of which extended identities to designate as external ones can be viewed as part of the protocol design process. See more discussion following the definition of *compliant protocols* in Section 6.

Recall that the treatement is already restricted to parameterized systems where the import given to each ITI must be at least the security parameter. Furthermore, the definition of protocol emulation concentrates 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 parameter. Consequently, the model still allows the environment to create situations where the computational resources of the protocol, hence the protocol's cmoputational and communication complexity, are arbitrarily large relative to the computational resources available to the adversary. Such situations seem unnatural; for instance, with such an environment no polytime adversary will be able to read even a fraction of the protocol's communication. Indeed, if such situations are allowed, then the definition of UC-emulation (Definition 9 below) would become overly restrictive.[14]

To avoid such situations, attention is restricted to 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. Specifically, say that an environment is balanced if, 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 $n_1, \ldots, n_k$ to k ITIs overall, then the overall import of the inputs to the adversary is at least $n_1 + \cdots + n_k$. 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 protocol parties (ITIs).

Distribution ensembles and indistinguishability. Toward the formal definition, 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 the security parameter. As will be seen, it will suffice to restrict attention to binary distributions, i.e., distributions over $\{0,1\}$.

Definition 8. Two binary probability distribution ensembles X and Y are indistinguishable (written $X \approx Y$) if for any $c, d \in \mathbf{N}$ there exists $k_0 \in \mathbf{N}$ such that for all $k > k_0$ and all $z \in \cup_{\kappa \le k^d}\{0,1\}^\kappa$, it holds that:

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

The probability distribution ensembles considered in this work represent outputs of systems of ITMs, namely, outputs of environments. More precisely, these are ensembles of the form $\textsc{exec}_{\pi,\mathcal{A},\mathcal{E}} \overset{\text{def}}{=} \{\textsc{exec}_{\pi,\mathcal{A},\mathcal{E}}(k,z)\}_{k\in\mathbf{N}, z\in\{0,1\}^*}$. It is stressed that Definition 8 considers the distributions $X(k,z)$ and $Y(k,z)$ only when the length of z is polynomial in k. This essentially means that only situations where the length of the initial input to the environment is some polynomial function of the security parameter are considered. (Recall that the import of the initial input

[14] Indeed, in such a case the adversary may not even be able to run the prescribed protocol. To exemplify this point further, for any protocol π let $\tilde{\pi}$ be identical to π except that π' instructs the party to first send to the adversary an initial random message of length proportional to the party's import, and expects the adversary to echo the message back to it before continuing (the message carries no import). Then π does not UC-emulate $\tilde{\pi}$ according to Definition 9: Let $\mathcal{A}$ be the linear-time adversary that delivers all protocol messages until it hits its run-time bound and halts. Now, to mimic the behavior of $\mathcal{A}$, the simulator $\mathcal{S}$ needs to have additional run-time to handle the additional message. But for any polytime $\mathcal{S}$ there would exist an environment that gives the ITIs running $\tilde{\pi}$ more import than the run-time of $\mathcal{S}$.

is taken to the environment be its length.) We are finally ready to formally define UC protocol emulation:

Definition 9. Let π and ϕ be PPT protocols, and let ξ be a predicate on extended identities. Say that π UC-emulates ϕ with respect to ξ-identity-bounded environments (or, π ξ-UC-emulates ϕ) if for any PPT adversary $\mathcal{A}$ there exists a PPT adversary $\mathcal{S}$ such that for any balanced, PPT, ξ-identity-bounded environment $\mathcal{E}$, it holds that

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

If π ξ-UC-emulates ϕ with $\xi = \{0,1\}^*$, then simply say that π UC-emulates ϕ.

Essentially all the discussion and interpretations of UC emulation, presented in Section 2.2 for the restricted model, apply here as well.

4.3 Alternative Formulations of UC-emulation

This subsection presents some alternative formulations of UC-emulation (Definition 9).

Environments with non-binary outputs. Definition 9 quantifies only over environments that generate binary outputs. One may consider an extension to the model 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 [121] (see also Reference [61]). It is easy to see, however, that this extra generality results in a definition that is equivalent to Definition 9.

Deterministic environments. Since environments 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 9. Again, the proof is omitted. Note however that this equivalence does *not* hold for the case of closed environments, where the environment has no input other than the import value.

4.3.1 Emulation with Respect to the Dummy Adversary. Definition 9 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 backdoor messages generated by the environment to the specified recipients, and delivers to the environment all backdoor messages generated by the protocol parties. Said otherwise, 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 $(i, (m, id, c, i'))$ from $\mathcal{E}$, where i is the import of the input, m is a message to be delivered, id is an identity, c is a code for an ITI, and i' is the import to be given out, $\mathcal{D}$ writes (i', m) on the backdoor tape of the ITI with identity id and code c, subject to the run-time limitations described below. When activated with a message m on its backdoor tape, adversary $\mathcal{D}$ passes m as subroutine-output to $\mathcal{E}$, along with the extended identity of the sender. (Recall that these messages carry no import.)

To make sure that $\mathcal{D}$ is polynomially bounded, we add the following mechanism. $\mathcal{D}$ keeps a variable v, which holds the total import received so far from $\mathcal{E}$, minus the import given out on the backdoor tape, minus the total lengths of all inputs and all incoming messages on the backdoor tape. If at any activation the variable v 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. (Note that the target identity *id* in an input message might be the environment itself, in which case the message is written back to the environment - albeit with no import. These messages can be used by the the environment to provide the adversary with additional import, without having it deliver any message.)

Definition 10. Let π and ϕ be PPT protocols and let ξ be a predicate on extended identities. Say that π ξ-UC-emulates protocol ϕ with respect to the dummy adversary if there exists a PPT adversary $\mathcal{S}$ such that for any balanced, PPT, ξ-identity-bounded environment $\mathcal{E}$, it holds that $\textsc{exec}_{\phi,\mathcal{S},\mathcal{E}} \approx \textsc{exec}_{\pi,\mathcal{D},\mathcal{E}}$.

We show:

Claim 11. *Let π, ϕ be PPT protocols, and let ξ be a predicate on extended identities. Then π ξ-UC-emulates ϕ (as in Definition 9) if and only if π ξ-UC-emulates ϕ with respect to the dummy adversary.*

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, this notion is almost always easier to work with. However, we chose not to present this formulation as the main notion of protocol emulation, since we feel it is somewhat less intuitively appealing than Definition 9. In other words, we find it harder to get convinced that emulation with respect to the dummy adversary captures the security requirements of a given task. In particular, it looks farther away from the the basic notion of security in, say, Reference [23]. Also, it is less obvious that this definition has some basic closure properties such as transitivity (see Claim 18).

At the same time, UC-emulation with respect to the dummy adversary might make it easier to see that UC security is a natural relaxation of the notion of *observational equivalence* of processes (see, e.g., Reference [99]). Indeed, observational equivalence essentially fixes the entire system that interacts with either π or ϕ, whereas UC-emulation with respect to the dummy adversary allows the analyst to insert a simulator that translates between the adversarial interface provided by ϕ and the adversarial interface provided by π, to make sure that the rest of the external system cannot distinguish between π and ϕ.

Proof. Fix some predicate ξ on extended identities. For the rest of the proof, we consider and construct only ξ-identity-bounded environments. Clearly, if π ξ-UC-emulates ϕ as in Definition 9, then π ξ-UC-emulates ϕ with respect to the dummy adversary. The idea of the implication 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 $\textsc{exec}_{\phi,\mathcal{S}_\mathcal{D},\mathcal{E}} \approx \textsc{exec}_{\pi,\mathcal{D},\mathcal{E}}$ for all balanced PPT $\mathcal{E}$.) We show that π UC-emulates ϕ according to Definition 9. For this purpose, given an adversary $\mathcal{A}$, construct an adversary $\mathcal{S}$ and show that $\textsc{exec}_{\phi,\mathcal{S},\mathcal{E}} \approx \textsc{exec}_{\pi,\mathcal{A},\mathcal{E}}$ for all balanced PPT $\mathcal{E}$. Adversary $\mathcal{S}$ runs simulated instances of $\mathcal{A}$ and $\mathcal{S}_\mathcal{D}$. 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 it completes its activation. (Note that $\mathcal{S}$ does not really "give away run-time tokens" here, since $\mathcal{A}$ is simulated within $\mathcal{S}$.)
   (b) Next $\mathcal{S}$ activates $S_\mathcal{D}$ with input $(i, (m, id, c, i'))$, where m is the outgoing message that $\mathcal{A}$ generated in this activation, id, c are the identity and code of the target ITI of this

message, and i' is the import of this message. (Recall that this message can be either a subroutine-output to the environment, or a backdoor message to some other ITI. If $\mathcal{A}$ did not generate a message in this activation, then $\mathcal{S}$ sets $(m, id, c, i') = (, 0, 0, 0)$, i.e., $\mathcal{S}$ inputs to $S_{\mathcal{D}}$ an instruction to send empty subroutine-output to the environment.) The import i of this input to $S_{\mathcal{D}}$ is set as $i = p_{\pi,\phi}(\hat{n}) - p_{\pi,\phi}(\hat{n} - n)$, where $p_{\pi,\phi}(\cdot)$ is the maximum between the polynomials bounding protocols π and ϕ, and $\hat{n}$ is the overall import in the inputs received by $\mathcal{S}$ in the execution so far. (Observe that this way, the overall import received by $\mathcal{S}_{\mathcal{D}}$ at any point is $p_{\pi,\phi}(\hat{n})$, which is assumed to be at least $\hat{n}$. This fact will be later used to argue the validity of $\mathcal{S}$.)[15]

Next $\mathcal{S}$ follows the instructions for $\mathcal{S}_{\mathcal{D}}$. In particular, if $\mathcal{S}_{\mathcal{D}}$ instructs to write some message to some tape of some ITI, then $\mathcal{S}$ writes that message to that tape of that ITI.

(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 $\mathcal{S}$ writes that message to the backdoor tape of that ITI.
   (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). If the subroutine-output of $\mathcal{S}_{\mathcal{D}}$ carries any import, then $\mathcal{S}$ halts. (This last instruction guarantees that $\mathcal{S}$ remains PPT even when $\mathcal{S}_{\mathcal{D}}$ is faulty.)
   (d) Next $\mathcal{S}$ runs $\mathcal{A}$: 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 $S_{\mathcal{D}}$ with input $(0, (m, id, c, i))$, where m is the message generated by $\mathcal{A}$, i' is the import of this message, and (id, c) are the identity and code of the recipient ITI. The input to $\mathcal{S}_{\mathcal{D}}$ does not contain any import. $\mathcal{S}$ then follows the instructions of $\mathcal{S}_{\mathcal{D}}$ for generating an outgoing message.

A graphical depiction of the operation of $\mathcal{S}$ appears in Figure 7.

Analysis of $\mathcal{S}$. We first argue that $\mathcal{S}$ is PPT. The running time of $\mathcal{S}$ is dominated by the run-time of the $\mathcal{A}$ module plus the run-time of the $\mathcal{S}_{\mathcal{D}}$ module (with some simulation overhead). Consider a state of $\mathcal{S}$: Let $\hat{n}$ be the overall import of the inputs received so far by $\mathcal{S}$, and recall that $\mathcal{A}$ and $\mathcal{S}$ receive no import on their backdoor tapes. This means that the overall running time of the $\mathcal{A}$ module is at most $p_{\mathcal{A}}(\hat{n})$, where $p_{\mathcal{A}}$ is the polynomial bounding the run-time of $\mathcal{A}$. The overall import of inputs received by the $\mathcal{S}_{\mathcal{D}}$ module within $\mathcal{S}$ is $p_{\pi,\phi}(\hat{n})$. It follows that the run-time of $\mathcal{S}_{\mathcal{D}}$ is bounded by

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

where $p_{\mathcal{S}_{\mathcal{D}}}$ denotes the polynomial bounding the run-time of $\mathcal{S}_{\mathcal{D}}$. Note that this fact is used also 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, ξ-identity-bounded environment $\mathcal{E}$ such that $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}} \not\approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$. Construct a balanced, ξ-identity-bounded environment $\mathcal{E}_{\mathcal{D}}$ such that $\text{EXEC}_{\pi,\mathcal{D},\mathcal{E}_{\mathcal{D}}} = \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$, and $\text{EXEC}_{\phi,\mathcal{S}_{\mathcal{D}},\mathcal{E}_{\mathcal{D}}} = \text{EXEC}_{\phi,\mathcal{S},\mathcal{E}}$. This will mean that $\text{EXEC}_{\phi,\mathcal{S}_{\mathcal{D}},\mathcal{E}_{\mathcal{D}}} \not\approx \text{EXEC}_{\pi,\mathcal{D},\mathcal{E}_{\mathcal{D}}}$, in contradiction to the premise that π ξ-UC-emulates ϕ with respect to the dummy adversary.

[15] We thank Andrew Miller for pointing out the need to activate $\mathcal{S}_{\mathcal{D}}$ even when $\mathcal{A}$ generates an subroutine-output to the environment or no message at all, to make sure that the environment $\mathcal{E}_{\mathcal{D}}$ (constructed later in the proof) remains balanced. This step was indeed missing in previous versions.

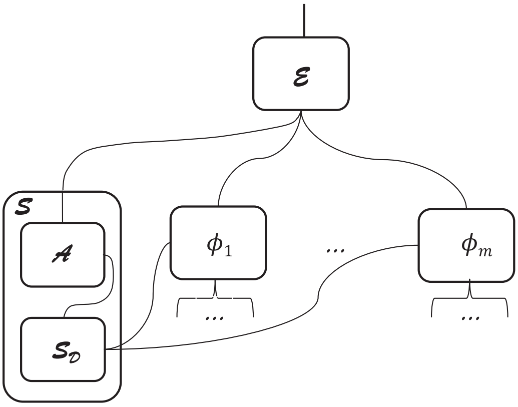

Fig. 7. The operation of simulator $\mathcal{S}$ in the proof of Claim 11. Both $\mathcal{A}$ and $\mathcal{S}_{\mathcal{D}}$ are simulated internally by $\mathcal{S}$. The same structure represents also the operation of the shell adversary in the definition of black-box simulation (see Section 4.3.2).

Environment $\mathcal{E}_{\mathcal{D}}$ internally runs an interaction between simulated instances of $\mathcal{E}$ and $\mathcal{A}$. That is, in the first activation on some initial input z, $\mathcal{E}_{\mathcal{D}}$ activates $\mathcal{E}$ on input z. Next:

(1) If $\mathcal{E}$ halts with some output, then $\mathcal{E}_{\mathcal{D}}$ halts with the same output. When $\mathcal{E}$ generates an input x with import n to some ITI (id, c), where id is the identity of the ITI and c is the code of the ITI, then:
   (a) If the target ITI is other than the adversary, then $\mathcal{E}_{\mathcal{D}}$ sends input x to ITI (id, c).
   (b) If the target ITI is the adversary, then $\mathcal{E}_{\mathcal{D}}$ activates $\mathcal{A}$ with input x and import n, and runs $\mathcal{A}$ until it completes its activation.

   Next $\mathcal{E}_{\mathcal{D}}$ provides an input to the external adversary, that mimics the input that $\mathcal{S}_{\mathcal{D}}$ receives when run within $\mathcal{S}$. That is, $\mathcal{E}_{\mathcal{D}}$ provides to the external adversary input $(i, (m, id, c, i'))$, where m is the outgoing message that $\mathcal{A}$ generated in this activation, id, c are the identity and code of the target ITI of this message, and i' is the import of this message. The import i of the input to the external adversary is set as $i = p_{\pi,\phi}(\hat{n}) - p_{\pi,\phi}(\hat{n} - n)$, where $p_{\pi,\phi}(\cdot)$ is the maximum between the polynomials bounding protocols π and ϕ, n is the import of the input received by $\mathcal{A}$ in this activation, and $\hat{n}$ is the overall import of the inputs received by $\mathcal{A}$ in the execution so far.

   It is stressed that $\mathcal{E}_{\mathcal{D}}$ activates the external adversary even if the target of this message is the environment. Also, if $\mathcal{A}$ did not generate a message in this activation then $\mathcal{E}_{\mathcal{D}}$ sets $(m, id, c, i') = (\perp, 0, 0, 0)$, i.e., $\mathcal{E}_{\mathcal{D}}$ instructs the external adversary to send an empty subroutine-output back to the environment.

   Next, if $id = 0$, i.e., the target identity of the message in the input to the external adversary is the environment, then $\mathcal{E}_{\mathcal{D}}$ enters a special bypass state.

(2) When $\mathcal{E}_{\mathcal{D}}$ obtains, on its subroutine-output tape, an output value v from the external adversary, it operates as follows:
   (a) If $\mathcal{E}_{\mathcal{D}}$ is in the bypass state, then it exits the bypass state, activates $\mathcal{E}$ with output v from $\mathcal{A}$, and proceeds as in Step 1. (Here $\mathcal{E}_{\mathcal{D}}$ bypasses $\mathcal{A}$ to deliver the output directly to $\mathcal{E}$, similarly to the behavior of $\mathcal{S}$.)
   (b) Else, $\mathcal{E}_{\mathcal{D}}$ parses $v = (m, id, c)$, activates $\mathcal{A}$ with incoming message m from ITI id with code c, and runs $\mathcal{A}$ till $\mathcal{A}$ completes its activation. If in this activation $\mathcal{A}$ generates an

outgoing message to its environment, then $\mathcal{E}_{\mathcal{D}}$ activates $\mathcal{E}$ with output v from $\mathcal{A}$. If $\mathcal{A}$ did not generate a message in this activation, then $\mathcal{E}_{\mathcal{D}}$ activates $\mathcal{E}$ with no output. Either way, $\mathcal{E}_{\mathcal{D}}$ then proceeds as in Step 1.

If the message generated by $\mathcal{A}$ is aimed at another ITI, then $\mathcal{E}_{\mathcal{D}}$ activates its external adversary with input $(0, (m, id, c, i'))$ where m is the message, i' is the import of the message and id, c are the identity and code of the recipient ITI. The import of this input is 0.

(3) When $\mathcal{E}_{\mathcal{D}}$ obtains an output value v from an ITI (id, c) other than the adversary, it activates $\mathcal{E}$ with output v from (id, c), and proceeds as in Step 1.

Clearly, if $\mathcal{E}$ is ξ-identity-bounded then so is $\mathcal{E}_{\mathcal{D}}$. Next, we argue that $\mathcal{E}_{\mathcal{D}}$ 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 import that $\mathcal{E}_{\mathcal{D}}$ gives to each ITI other than the adversary is at most the import that $\mathcal{E}$ gives this ITI. The overall import that $\mathcal{E}_{\mathcal{D}}$ gives its external adversary is at least $p_{\pi,\phi}(\hat{n})$, where $\hat{n}$ is the overall import that $\mathcal{E}$ has given its own external adversary (namely, $\mathcal{A}$) so far. Assuming that $p_{\pi,\phi}(\hat{n}) \geq \hat{n}$, it holds that $\mathcal{E}_{\mathcal{D}}$ is balanced.

It can also be verified that ensembles $\textsc{exec}_{\pi,\mathcal{D},\mathcal{E}_{\mathcal{D}}}$ and $\textsc{exec}_{\pi,\mathcal{A},\mathcal{E}}$ are identically distributed. (Here it is instructive to note that $\mathcal{D}$ never stops due to insufficient run-time.)

Similarly, $\textsc{exec}_{\phi,\mathcal{S}_{\mathcal{D}},\mathcal{E}_{\mathcal{D}}}$ and $\textsc{exec}_{\phi,\mathcal{S},\mathcal{E}}$ are identically distributed. In particular, the view of $\mathcal{S}_{\mathcal{D}}$ in the left experiment is distributed identically to the view of the $\mathcal{S}_{\mathcal{D}}$ module within $\mathcal{S}$ in the right experiment. □

4.3.2 Emulation with Respect to Black-box Simulation. Another alternative formulation of Definition 9 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., References [7, 64]. We show that in the UC framework security via black-box simulation is *equivalent* to the standard notion of security.

Black box emulation is formulated in a way that keeps the overall model of protocol execution unchanged, and only imposes restrictions on the operation of the simulator. Specifically, say that an adversary is composite if it consists of a main program or ITM and a subroutine, whose program is another ITM. Upon activation of a composite adversary, the main program is activated. The main program invokes and activates the subroutine at will and obtains the subroutine-outputs of the subroutine, but does not have access to the program or internal state of the subroutine. Furthermore the subroutine does not have access to the outgoing message tape of the overall composite ITM. A black-box simulator $\mathcal{S}$ is the main program of a composite adversary. Let $\mathcal{S}^{\mathcal{A}}$ denote the composite adversary that consists of the main program $\mathcal{S}$ with subroutine ITM $\mathcal{A}$. Black-box simulator $\mathcal{S}$ is PPT with bounding polynomial $p_s(\cdot)$ if, for any PPT $\mathcal{A}$, the number of computational steps taken by the main program of $\mathcal{S}^{\mathcal{A}}$ is bounded by $p_s(n - n')$, where n is the overall import received by $\mathcal{S}^{\mathcal{A}}$ on its input tape, and n' is the overall import that $\mathcal{S}$ provides to $\mathcal{A}$.

Definition 12. Let ξ be a predicate on extended identities. Say that protocol π ξ-UC-emulates protocol ϕ with black-box simulation if there exists a PPT black-box adversary $\mathcal{S}$ such that for any PPT adversary $\mathcal{A}$, and any balanced, PPT, ξ-identity-bounded environment $\mathcal{E}$, we have $\textsc{exec}_{\phi,\mathcal{S}^{\mathcal{A}},\mathcal{E}} \approx \textsc{exec}_{\pi,\mathcal{A},\mathcal{E}}$.

Observe that UC-emulation with black-box simulation is equivalent to plain UC-emulation:

Claim 13. *Let π, ϕ be PPT multiparty protocols, and let ξ be a predicate on extended identities. Then π ξ-UC-emulates ϕ as in Definition 9 if and only if π ξ-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 7. □

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 subroutine-outputs of complete executions with potentially hidden randomness. Still, the present definition is equivalent to the plain (non black-box) notion of security.

The present formulation of black-box simulation is reminiscent of the notions of strong black-box simulation in Reference [85] and in Reference [108]. However, in these works black-box simulation is not equivalent to the basic definition, due to different formalizations of probabilistic polynomial time.

4.3.3 Letting the Simulator Depend on the Environment. Consider a variant of Definition 9, 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 Reference [89], we call this variant emulation with respect to specialized simulators. As argued in Reference [89], emulation with respect to specialized simulators does not directly provide the guarantees promised by UC security. Indeed, our proof of the UC theorem crucially uses the fact that the same simulator works for all environments. However, it turns out that, within the present framework, emulation with respect to specialized simulators is actually equivalent to full-fledged UC security, hence UC with specialized simulators suffices for applying the UC theorem.

Claim 14. *A protocol π ξ-UC-emulates protocol ϕ as in Definition 9 if and only if π ξ-UC-emulates ϕ with respect to specialized simulators.*

Proof. Clearly, if π UC-emulates ϕ as in Definition 9 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.) It is thus guaranteed that there exists a simulator $\mathcal{S}$ such that $\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}_u} \approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}_u}$. It holds 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 6, 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 $d \in \mathbf{N}$, it holds that

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

In particular, as long as $|z|$ is polynomial in k, it holds 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 Reference [23]. 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 input. Indeed, there are several definitional frameworks in the literature (including Reference [23] and the basic model of Section 2) that restrict ITMs to run in time that is bound by a fixed polynomial in the security parameter. In these framework standard security and security with respect to specialized simulators end up being different notions (see, e.g., References [78, 89]).

4.4 Some Variants of UC-emulation

Next, some variants of the basic notion of UC-emulation, specifically statistical emulation, emulation with respect to closed environments, and two other, more quantitative notions of UC-emulation. We then make note additional observations.

On statistical and perfect emulation. Definition 9 can be extended to the standard notions of statistical and perfect emulation (as in, say, Reference [23]). 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 $\mathcal{A}$, 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 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, as discussed in Reference [23].

On security with respect to closed environments. Definition 9 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, environments that do not receive meaningful external input. Here the notion of security considers only environments whose external input contains no information other than import. Such environments would choose the inputs of the protocol parties using some internal stochastic process. Note that Claim 14 does *not* hold for closed environments. Indeed, jumping ahead, it can be seen that: (a) The UC theorem does not hold with respect to closed environments and specialized simulators. (b) As long as there is a single simulator that works for all environments, the UC theorem holds even with respect to closed environments.

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." This subsection formulates more quantitative variants of this definition.

Two parameters are quantified: 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 6), and that a functional is a function from functions to functions. Then:

Definition 15. Let π and ϕ be protocols, let ξ be a predicate on extended identities, and let ϵ, g be functionals. Say that π ξ-UC-emulates ϕ with emulation slack ϵ and simulation overhead g (or, in short, π (ϵ, g, ξ)-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, ξ-identity-bounded environment $\mathcal{E}$, any large enough value $k \in \mathbf{N}$ and any input $x \in \{0,1\}^{p_{\mathcal{E}}(k)}$, it holds that

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

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 $k \in \mathbf{N}$, let $g, \epsilon : \mathbf{N} \to \mathbf{N}$, and let ξ be a predicate on extended identities. Say that π (k, e, g, ξ)-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 ξ-identity-bounded environment $\mathcal{E}$ that runs in time $t_{\mathcal{E}}$, and any input $x \in \{0,1\}^{t_{\mathcal{E}}}$, it holds that

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

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 and let ξ be a predicate such that π (ϵ, g, ξ)-UC-emulates ϕ as in Definition 15. Then there exists a polynomial α such that π (ϵ, g', ξ)-UC-emulates ϕ, where $g'(p_{\mathcal{A}})(\cdot) = p_{\mathcal{A}}(\cdot) + \alpha(\cdot)$.*

Said otherwise, if π (ϵ, g, ξ)-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 or changing the predicate ξ. 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)). □

4.5 UC-emulation is Transitive

The following observation is straightforward:

Claim 18. *Let π_1, π_2, π_3 be protocols and let ξ_1, ξ_2 be predicates. Then, if π_1 ξ_1-UC-emulates π_2, and π_2 ξ_2-UC-emulates protocol π_3, then π_1 ξ_3-UC-emulates π_3, where $\xi_3 = \xi_1 \cup \xi_2$.*

Moreover, if π_1 (e_1, g_1, ξ_1)-UC-emulates π_2, and π_2 (e_2, g_2, ξ_2)-UC-emulates π_3, then π_1 $(e_1 + e_2, g_2 \circ g_1, \xi_1 \cup \xi_2)$-UC-emulates π_3. (Here $e_1 + e_2$ is the functional that outputs the sum of the outputs of e_1 and e_2, and $\circ$ denotes composition of functionals.) Transitivity for any number of protocols $\pi_1, \ldots, \pi_n$ 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.

We stress that transitivity of UC-emulation should not be confused with the case of UC-emulation for multiple *nested* protocols, which has to do with repeated applications of the UC theorem and is discussed in Section 6.3.

5 DEFINING SECURITY OF PROTOCOLS

We now turn to applying the general machinery of UC-emulation, developed in Section 4, towards one of the main goals of this work, namely, defining security of protocols via realizing ideal functionalities.

This section first formulate an additional, more structural class of protocols, called subroutine-respecting protocols. Essentially, these are protocols where only the main ITIs of each session take inputs from, and generate output to, ITIs that are not members of the extended session. To facilitate presenting this additional requirement, we formulate a general mechanism for making structural requirements from protocols. This mechanism, which we dub the *shell mechanism*, will be used extensively throughout the rest of this article: We will use it to capture a variety of real-life situations and security concerns within the minimalistic formal framework of Sections 3 and 4.

Section 5.1 presents the shell mechanism. Section 5.2 presents subroutine-respecting protocols. Section 5.3 defines ideal functionalities and ideal protocols for carrying out a given functionality, followed by the definition of securely realizing an ideal functionality.

5.1 Structured Protocols

This subsection presents a mechanism for fine-tuning the basic model of protocol execution to capture realistic attacks and security concerns in a more nimble way. The idea is simple: Instead of directly analyzing the security of the given protocol π, consider a protocol π' that "augments" π in some prescribed way: Essentially, π' will run π "encapsulated" within some "wrapper," or "shell" code that monitors and sometimes modifies the communication between π and the outside. Conceptually, the shell code will contain model-related instructions that are part of the "mental experiment" of security analysis; The advantage in using this mechanism is that, while different shells capture different actual settings, a structured protocol is always a legal protocol as per the modeling of Section 4, hence all the structural results of that section apply. Indeed, this mechanism will be used extensively: For instance, the code that sends messages on the backdoor tape and responds to messages coming on that tape will typically be shell code (indeed, actual real-life code does not typically include messages for communicating with the adversary). Also the shell mechanism is what will allow us to capture, in Section 7.1, communication and party-corruption within the basic model.

More specifically, say that a protocol (or ITM) is structured if it consists of two separate parts, or "sub-processes," called a body and a shell. It will be convenient to view the body and the shell of a structured ITM μ as two separate ITMs, where the shell ITM has read and write access to the tapes of the body, whereas the body does not have access to the tapes of the shell. The externally writable tapes of μ are the externally writable tapes of the shell, and the outgoing message tape of μ is the outgoing message tape of the shell. The externally writable and outgoing message tape of the body become internal tapes of μ and are writable and readable by the shell, respectively. The extended identity of μ represents the extended identity of the shell followed by that of the body. (A natural layout would be $(SID_{\text{B}}, SID_{\text{SH}}, PID_{\text{B}}, PID_{\text{SH}}, \pi_{\text{B}}, \pi_{\text{SH}})$, where SID, PID, π denote the SID, PID and program, and the subscripts B,SH denote shell and body, respectively. The shell can read the full identity tape of μ, whereas the body can read only the parts pertaining the the body. Neither the body nor the shell can modify any part of the identity tape.)

An activation of μ starts by activating the shell, which may or may not activate the body. (Activating the body is done by setting its activation bit to 1.) In case the body executes, it keeps executing until it completes its activation, at which point the shell resumes executing. It is stressed that only the shell sends outgoing messages or completes an activation of μ. (The shell may, but is not required to, copy outgoing messages from the body's outgoing message tape to the outgoing message tape of μ.) The run-time of μ includes the run-time of both the shell and the body.[16]

The definition of time bounded ITMs (Definition 6) applies also to structured ITMs. In particular, the import balance of a structured ITM is calculated with respect to the incoming and outgoing communiation of the shell. (The import values of the messages received and sent by the body are not counted toward the overall import balance of μ. In particular, the shell may set the import values in incoming messages to the body to be different than the import values on its own incoming messages.)

All protocols are henceforth assumed to be structured. Furthermore, the body of a structured protocol can be structured in and of itself. That is, protocols might have multiple shells, where each shell is unaware of the outer shells, and treats the inner shells as body. In fact, this will be the prevalent case later on, where the "innermost" body often correspond to actual, real-life code, and the shells represent "modeling code," namely, code that represents the security experiment.

5.2 Subroutine Respecting Protocols

The model of protocol execution of Section 4.1 is a highly simplified rendering of an execution of a single protocol session within a general execution environment where the protocol session is created by, and runs alongside, other ITIs. In particular, this model does not allow for ITIs other than $\mathcal{E}$, $\mathcal{A}$, and machines that are members of the extended test session of π. Furthermore, $\mathcal{E}$ is only allowed to provide inputs to the main parties of the test session of π. This, in particular, means that the sub-parties of the test session of π are never faced with a situation where they might obtain subroutine-output from an ITI that is not already a member of this extended session. Similarly, they are never faced with a situation where they might obtain input from an ITI that is not already a member of this extended session. Furthermore, when providing input to another ITI, members of the extended session of π do not face situations where the recipient ITI already exists in the system but is not a member of the extended test session.

In contrast, when executed as a component of a larger system where ITIs outside the extended test session of π may coexist, such situations might indeed happen. Furthermore, this gap between

[16] As in the definition of ITMs, we allow the shell to copy incoming messages from its own externally writable tapes to those of the body in unit time. As discussed in footnote 5, this added lenience is not necessary in the RAM/PRAM models where the shell can instead "redirect" the body to read the message from its own tape.

the formal model and the actual execution environment might lead to actual attacks that take advantage of the fact that the actual execution environment does not provide the assumed absolute separation between the extended session of the protocol and the rest of the system.

We present a mechanism for bridging this gap between the "*in vitro*" model of protocol execution and general execution environments. To allow capturing a variety of system designs, the gap is bridged in two steps: First, we define a set of "behavioral requirements" from protocols, formulated in terms of restrictions on the pattern of message receipt and generation, without making specific how these requirements might be implemented. Protocols that satisfy these behavioral requirements are called subroutine-respecting. These behavioral guarantees will arguably suffice for capturing our intuitive notion of security. In particular, it will suffice for the universal composition theorem.

Next, for sake of concreteness, we formulate a specific mechanism that implements these behavioral requirements. This mechanism models "complete separation," namely, a system where all the attempts at "extraneous" communication, either by the members of the extended protocol session or by the other ITIs in the system, are intercepted and blocked. We leave the construction of alternative mechanisms, that may provide other levels of separation, to future work.

Definition 19. Protocol π is subroutine-respecting if each session s of π, occurring within an execution of any protocol with any environment, satisfies the following four requirements, in any execution of any protocol ρ with any adversary and environment, as per the definition of protocol execution of Section 4.1. (It is stressed that these requirements must be satisfied even when session s of π is a subroutine of ρ and, in particular, when the execution involves ITIs that are not members of the extended session s.)

(1) The sub-parties of session s reject all inputs passed from an ITI that is not already a main party or subsidiary of session s. (Recall that rejecting a message means that the recipient ITI returns to its state prior to receiving the message and ends the activation without sending any message; see Section 3.1.2.)
(2) The main parties and sub-parties of session s reject all incoming subroutine-outputs passed from an ITI that is not already a main party or subsidiary of session s.
(3) No sub-party of session s passes subroutine-output to an existing ITI that is not already a main party or sub-party of session s.
(4) No main party or sub-party of session s passes input to an existing ITI that is not already a main party or sub-party of session s.

Writing subroutine-respecting protocols. First observe that in the basic model of Section 2 all protocols are automatically subroutine-respecting. Indeed, the above requirements are already "baked into" the static communication structure there (embodied in the construct of communication sets).

More generally, as discussed above, the methods for guaranteeing that protocols remain subroutine-respecting naturally depend on the level of control over inter-process communication, or more generally on the level of separation that is provided by the physical system under consideration. For sake of concreteness, we describe one such method. Here the formalism of structured protocols become useful: Define a shell that represents a very protective architecture, where the main ITIs of each session of the analyzed protocol operate within an isolated "sandbox" that enforces suboutine-respectfullness, no matter what the body does. That is, only the main ITIs can receive inputs from external ITIs, no input can be given to an existing external ITI, and no subroutine-output can be given by a sub-party of the session to an external ITI. (Less protective execution environments can be captured by appropriately relaxing the guarantees provided by this shell.)

More specifically, a complete sandbox shell for protocol π is a program σ_π that proceeds as follows:

(1) At first activation:
   (a) If the code of the body of the present (structured) ITI is π, and the SID of the shell is $\top$, then set flag `main-party`.
   (b) Else, if the shell code and shell SID of the caller ITI are the same as the shell code and shell SID of the present ITI, and in addition the incoming message is an input (as opposed to subroutine-output), then set flag `sub-party`.
   (c) Else, halt and remain inactive in all subsequent activations.

   (Case (1a) indicates that the present ITI is a main party of the session of π to be protected. Case (1b) indicates that the caller ITI is a member of the extended session to be protected.)

(2) When receiving a message on an externally writable tape: If the message is an input, and the flag `main-party` is set, then the input is transferred directly to the body. If the `sub-party` flag is set, then first verify that the shell of the sending ITI M' has the same code and SID as the present one. If so, then activate the body with the input message. Else, reject the input.

   Similarly, if the message is a subroutine-output from an ITI M', then first verify that the shell code and SID of the sending ITI M' are the same as the present one. If so, then the shell activates the body with the subroutine-output. Else, it rejects the output.

   Incoming backdoor messages are forwarded to the body without change. (Presumably, these messages will be handled by an inner shell within the body.)

(3) When the body completes an activation with an outgoing message v to an ITI M': If the outgoing message is an subroutine-output and the `main-party` flag is set, then send the message. Else, generate an instruction to provide output v to ITI M'', where M'' is the same as M' except that the code σ is added to the code of M' as a new shell, and the present shell SID is added to the SID of the recipient.

   Similarly, if v is an input message, then generate an instruction to provide input v to an ITI M'', where M'' is the same as M' except that a the code σ is added to the code of M' as a new shell, and the present shell SID is added to the SID of the recipient.

It can be verified that the complete sandbox shell σ_π makes sure that the overall structured protocol (π, σ_π) is subroutine-respecting, regardless of how the body behaves. Indeed, consider a session of (π, σ_π), with SID $(s, \top)$ (i.e., SID s for the body and $\top$ for the shell). Then all the ITIs in the extended session $((\pi, \sigma_\pi), (s, \top))$ have shell σ_π. Furthermore, all the sub-parties of this extended session have shell SID s. Observe that the shell σ_π prevents sub-parties of the session $((\pi, \sigma_\pi), (s, \top))$ from sending subroutine-outputs to new ITIs that are not already sub-parties of this session; it also prevents members of the session from sending inputs to existing ITIs that are not already members of session $((\pi, \sigma_\pi), (s, \top))$.[17]

5.3 Realizing Ideal Functionalities

As discussed in Section 2.2, security of protocols is defined by way of comparing the protocol execution to an *ideal process* for carrying out the task at hand, where the ideal process takes the

[17] Thanks to Christian Badertscher, Julia Hesse, Björn Tackmann, and Vassilis Zikas for pointing to inclarities and mistakes in previous formulations of subroutine-respecting protocols and for helping develop the current formalism.

form of running a special protocol called the ideal protocol for the task. Recall that the ideal protocol consists of an ideal functionality, which is a single machine that captures the desired functionality of the task by way of a set of instructions for a "trusted party," plus multiple "dummy parties" whose role is to make sure that the ideal protocol syntactically looks like a distributed protocol that consists of multiple separate machines, while forwarding all inputs to the ideal functionality and forwarding all subroutine-outputs from the ideal functionality to their specified destinations.

We extend the concepts of ideal functionalities and ideal protocols to the present model. An ideal functionality is now simply an ITI. (The PID of an ideal functionality may often be meaningless and set to $\perp$, but this is not formally necessary.) Extending the notion of dummy parties requires some care, given the dynamic character of the present formalism. Indeed, dummy parties can now be created during the course of the computation, with dynamically generated identities and programs. Furthermore, they and can be required to transmit an unbounded volume of inputs and subroutine-outputs.

The ideal protocol. The ideal protocol $\textsc{ideal}_{\mathcal{F}}$ for an ideal functionality $\mathcal{F}$ is defined as follows. Let (p, s) be the party and SID. Then:

(1) When activated with own SID s, own PID p, and with input v, coming from an ITI with extended identity eid_c, do: If the reveal-sender-identity flag is not set, then do nothing. Else, pass input (v, eid_c) to an instance $\mathcal{F}$ with identity $(s, \perp)$ (i.e., to the ITI with code $\mathcal{F}$, SID s and PID $\perp$), 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. Furthermore, $\mathcal{F}$ will obtain also p, the PID of the dummy party.)

(2) When activated with subroutine-output $(v, (s, p), eid_t)$ from the ITI with code $\mathcal{F}$ and identity $(s, \perp)$, where v is the actual subroutine-output value, and eid_t is the extended identity of the target ITI, pass subroutine-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 7.2.)

In terms of body and shell, both the dummy parties and the ideal functionality are shell-only protocols, in the sense that they are only part of the security analysis, and contain no "real life code." Note also that the ideal functionality $\mathcal{F}$ is, technically, a different protocol session than $\textsc{ideal}_{\mathcal{F}}$, since its code is different. In particular, it can be accessed by the environment only via the dummy parties.

To make sure that $\textsc{ideal}_{\mathcal{F}}$ is formally PPT without adding complexity to the task of protocol design, set the polynomial bounding the run-time of the dummy parties be large enough so that the dummy parties will need to "shave off" only a small fraction of the import passed to and from the ideal functionality. Specifically, let $p(\cdot)$ be the polynomial bounding the run-time of $\mathcal{F}$. The polynomial bounding the run-time of $\textsc{ideal}_{\mathcal{F}}$ will be $p'(\cdot) = p(q(\cdot))$, where $q(\cdot)$ is some fixed (potentially large) polynomial. Upon receiving input of length k and with import n, $\textsc{ideal}_{\mathcal{F}}$ will pass import $n' = n - q^{-1}(n)$ to $\mathcal{F}$. Similarly, upon receiving from $\mathcal{F}$ subroutine-output with import n, $\textsc{ideal}_{\mathcal{F}}$ will pass import $n' = n - q^{-1}(n)$ to the target ITI. (In both cases, if $p(n) < k$ then the activation ends without doing anything.) This mechanism guarantees that, when activated on input or subroutine-output with import n, $\textsc{ideal}_{\mathcal{F}}$ "keeps for itself" import $q^{-1}(n)$, which allows it to make $p(q(q^{-1}(n))) = p(n)$ steps - similarly to $\mathcal{F}$. Furthermore, the larger q is, the faster the ratio n'/n tends to 1 when n grows. 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.[18,19]

Realizing an ideal functionality. We are finally ready to define what it means for a protocol π to meet its specification, where the specification consists of interdependent correctness, termination and secrecy requirements, represented by way of an ideal functionality $\mathcal{F}$. The basic idea is to say that π UC-realizes $\mathcal{F}$ if π UC-emulates the ideal protocol for $\mathcal{F}$. However, as discussed in Section 5.2, this requirement does not suffice by itself for guaranteeing meaningful security, and one needs to additionally require that there is a clear parition between the ITIs that are "associated with a session of π" and the rest of the system, and that the only communication between the these ITIs and the rest of the system is done via inputs received by the main parties of this session, and subroutine-outputs generated by these parties. This is captured by requiring that the realizing protocol is subroutine-respecting. That is:

Definition 20. Let $\mathcal{F}$ be an ideal functionality, let π be a protocol, and let ξ be a predicate on extended identities. Protocol π ξ-UC-realizes $\mathcal{F}$ if π is subroutine-respecting, and in addition π ξ-UC-emulates $\textsc{ideal}_{\mathcal{F}}$, the ideal protocol for $\mathcal{F}$.

6 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. (A graphical depiction of the composition operation appears in Figure 4.) Section 6.1 defines the composition operation and states the composition theorem. Section 6.2 presents the proof. Section 6.3 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 to the present model of execution. The extensions are significant. In particular, while in the basic model of Section 2.3 the operation of substituting a session of protocol ϕ with a session of a protocol π (that preumably UC-realizes ϕ) is straightforward, in the present dynamic model this operation is a delicate "surgery" that requires careful specification. Furthermore, in keeping with the fact that our model of computation allows protocols to deynamically determine their subroutines, here the composition operation is formulated as an *operation on protocols,* namely, as a transformation applied to the code of the calling protocol, rather than as a "model operation" (as done in Section 2.3).[20] Finally, while the composition operation of Section 2.3 replaces a single session of ϕ with a session of π, here the composition operation replaces in a single step multiple sessions of ϕ with instnces of π, while making sure that the change is "seamless" from the point of view of both the calling protocol and the subroutine protocol.

[18] Alternatively, one can simply keep the definition of $\textsc{ideal}_{\mathcal{F}}$ as described in items 1–3, and allow $\textsc{ideal}_{\mathcal{F}}$ 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 7 would hold even if the system included ideal protocols where $\textsc{ideal}_{\mathcal{F}}$ is as proposed in this footnote.) Still, to keep the overall exposition simple, we make sure that $\textsc{ideal}_{\mathcal{F}}$ is polytime.

[19] 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.

[20] In particular, formalizing the composition operation as an operation on protocols allows us to treat the composed protocol as a self-contained object $\rho^{\phi\to\pi}$, without considering any specific model of protocol execution. It also allows keeping the model of execution simple and unchanged throughout.

6.1 The Universal Composition Operation and Theorem

The universal composition operator UC() takes as input protocols ρ, ϕ and π, such that ρ presumably makes subroutine calls to a protocol ϕ, and π presumably UC-emulates ϕ. The intention is to create a composed protocol $\rho^{\phi\rightarrow\pi} = \text{UC}(\rho, \pi, \phi)$ that behaves essentially the same as ρ, except that each call to ϕ is replaces by a call to π. However, while the intuitive intended meaning is clear, the details require some care.

To define the operator UC() more formally, we again make use of the shell and body mechanism. Specifically, shell code is added to ρ that changes the target code, in inputs given to the main ITIs of top-level sessions of ϕ, from ϕ to π. Similarly, in subroutine-outputs generated by these ITIs, it changes the source code from π back to ϕ. To function properly the shell also copies itself to all the sub-parties of ρ, while making sure that the bodies of all the ITIs in the extended session of the resulting structured protocol remain unaware of the existence of the shell. In particular, the "ϕ to π" replacement takes place not only at the main ITIs of ρ, but rather at any sub-party of a session of ρ, as long as the session of ϕ is a "top-level session"—namely, the calling ITI is not also a main or sub-party ITI of a session of ϕ.[21]

Specifically, given protocols ρ, ϕ and π, the composed protocol $\rho^{\phi\rightarrow\pi} = \text{UC}(\rho, \pi, \phi)$ is the structured protocol whose body is ρ, and whose shell, denoted $\sigma_{\pi,\phi}$, is the following:

(1) When activated with input (respectively, subroutine-output) v passed from an ITI with code ψ and identity (sid, pid), do:
   (a) If this is the first activation, then:
      (i) If v of the form (v',(`Shell:main:`ρ, s; `subr:`π, s')) for some ρ, then store (ρ, s) in variable *main*, and (π, s') in variable *subr*. (Variable *main* holds the SID and code of the protocol to which the UC operator is applied. If the present ITI is part of an extended instance of π or ϕ, then variable *subr* holds the SID and code of that instance; otherwise *subr* holds $\perp$.)
      (ii) Else, store the code and SID of the body of the present ITI in variable *main*, and set *subr* $\leftarrow \perp$. (This occurs when the present ITI is a main ITI of an instance of the calling protocol, and the sending ITI is outside that instance.)
   (b) Activate the body with input (respectively, subroutine-output) v' from an ITI with code ψ' and identity (sid, pid), where ψ' is determined as follows:
      (i) If v is an input, the code and SID of the body of the present ITI equal those in *main*, and are different than (ψ, sid), then $\psi' = \psi$. (This case may occur when the present ITI is a main ITI of ρ.)
      (ii) Else, ψ is interpreted as structured code with shell code $\sigma_{\rho,\pi,\phi}$ and with body code $\tilde{\psi}$. If v is an subroutine-output, and in addition $\tilde{\psi} = \pi$ and *subr* $= \perp$, then $\psi' = \phi$. Else $\psi' = \tilde{\psi}$.

(2) When the body instructs to pass input (respectively, subroutine-output) v' to an ITI running ψ' with identity (sid, pid), the shell passes input (respectively, subroutine-output) v to an ITI running code ψ with identity (sid, pid), where:
   (a) If $\psi' = \phi$ and *subr* $= \perp$, then $\psi = (\pi, \sigma_{\rho,\pi,\phi})$ and $v = $ (v',(`Shell:main:`*main*; `subr:`π, sid)).
   (b) If v' is subroutine-output, the code and SID of the body of the present ITI equal those in *subr* and are different than (ψ', sid), then $\psi = (\psi', \sigma_{\pi,\phi})$ and $v = $ (v', (`Shell:main:`*main*; `subr:`$\perp$)).

[21] This provision was not made clear in prior versions of this work, and we thank Björn Tackmann for calling out this omission.

   (c) If v' is subroutine-output, the code and SID of the body of the present ITI equal those in *main*, and are different than (ψ', sid), then $\psi = \psi'$ and $v = v'$.
   (d) Else $\psi = (\psi', \sigma_{\pi,\phi})$, and $v' = (v,$ `(Shell:main:`*main*`; subr:`*subr*`)`$)$.
(3) Backdoor messages from the adversary are forwarded to the body without change. Similarly, backdoor messages generated by the body are forwarded to the adversary without change.

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

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

Compliant and subroutine-exposing protocols. As in Section 2.3, the UC theorem will generally state that if protocol π UC-realizes ϕ then, for any protocol ρ, the protocol $\rho^{\phi\rightarrow\pi}$ UC-realized the original protocol ρ. However, in the present model we will need to impose a number of restrictions on ρ, π and ϕ: First, we will need to require that the identities of the ITIs that provide input to any instance of ϕ are compliant with the set ξ with respect to which π realizes ϕ. Second, we will need to require that π and ϕ are subroutine-respecting. Third, π and ϕ will need to be constructed so that their subroutine structure is exposed to the adversary, 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 member of a given session of π (or ϕ). In more detail, protocol ρ is called (π, ϕ, ξ)-compliant if:

- All external-writes made by main parties and sub-parties of any session of ρ, where the target tape is the input tape, use the forced-write mode. Similarly, all messages received on the subroutine-output tapes of these ITIs are expected to have reveal-sender-id flag on; other subroutine-outputs are rejected.
- No two external-write instructions, out of all the external-write instructions made by the members of an extended session of ρ, where one instruction has target code π, and the other instruction has target code ϕ, have the same target SID.
- The extended identities of all the ITIs in any extended session of ρ, that pass inputs to ITIs with code either π or ϕ, satisfy the predicate ξ.

As in the case of identity-bounded environments, ξ can be a polytime predicate that takes as input an entire configuration of the system at the moment where the ITI passes an input to a member of some protocol session, and determines whether to accept that input. (It is stressed, however, that the predicate ξ need not be evaluated locally by any ITI.)

We proceed to the final requirement, namely, that the subroutine structure of the relevant session of protocols π (respectively, ϕ) be exposed to the adversary. As in the case of subroutine-respecting protocols, we first present the requirement in a more general way, without specifying any implementation. Next, we present a concrete mechanism that implements the requirement.

Definition 21. Protocol π is subroutine-exposing if each session s of π, occurring within an execution of any protocol with any environment, provides an interface to the adversary where the adversary can specify an extended identity α and is notified in response whether α is a member of the extended session (π, s). The interface should take the form of a query on the backdoor tape of an ITI whose extended identity should be determined given (π, s) alone.

The general requirement is implemented in a straightforward way. In a nutshell, the mechanism requires each session s of π to include a dedicated "directory ITI," which will keep record of tall the ITIs that currently belong to the extended session (π, s). To find whether a given ITI is part of the extended session of a session of π (or ϕ), it will query the directory ITI of that session. To allow the directory ITI to keep the necessary record, each ITI in the extended session will inform the directory ITI in its first activation, and also before generating any input or subroutine-output to a new ITI.

In more detail, we specify the following mechanism, which again makes use of the body and shell structure. Protocol π is called subroutine-exposing for session s if it is structured, and its shell code, denoted χ_s, proceeds as follows:

(1) Let $\top$ be a special identifier, interpreted as "directory." In the first activation of an ITI with code π, SID s and PID $p \neq \top$, the shell of (π, s, p) 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.

(2) Let the local code be (μ, χ_s), and let the local identity be id. Then, when the body instructs to provide input (respectively, subroutine-output) v to ITI $M = (\mu', id')$, the shell of (μ, id) first sends input `(sending input` (respectively, `subroutine-output`)`(`μ', id'`))` to ITI $(\pi, s, \top)$. (See the instructions for ITI $(\pi, s, \top)$ in item 3 below.) Upon receiving subroutine-output `ok` from $(\pi, s, \top)$, the shell of (μ, id) external-writes v to the input tape of ITI $((\mu', \chi_s), id')$.

Note that the sub-parties of session s of π obtain the same shell χ_s as the main machines of session (π, s), and so they all use use the same directory ITI. That is, the directory serves all the ITIs in the extended session (π, s).

(3) If the local identity is $(\pi, s, \top)$, then the body is never activated. Instead, when activated with input `started` from an ITI M, where M is either a main ITI of session s of π, or else M is in the set of eligible ITIs, then the shell records M as a member of the extended session of s. Next $(\pi, s, \top)$ subroutine-outputs `ok` to M.

When activated with input `(sending input to` M'`)`, or `(sending subroutine to` M'`)`, coming from an ITI M, which is already recorded as a member of the extended session s, then: If M is a main party of session s of π, and M reports an subroutine-output, then the shell just outputs `ok` to M. Else, the shell adds M' to the set of eligible ITIs and outputs `ok` to M.

When activated with a backdoor message `(query` α`)` from the adversary, the shell informs the adversary whether ITI α is recorded as a member of extended session s.

To make sure that the directory ITI remains polytime, each `(invoking` M`)` input should carry enough import to cover the cost of registering M, plus the cost of answering a query.[22]

We are now ready to state the composition theorem. First, a general theorem is stated, to be followed by two corollaries. A more quantitative statement of the UC theorem is discussed in Section 6.3.

Theorem 22 (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, ϕ and π are both subroutine-respecting and subroutine-exposing, and π ξ-UC-emulates ϕ. Then protocol $\rho^{\phi \rightarrow \pi}$ UC-emulates protocol ρ.*

[22] An alternative way to turn subroutine protocols into subroutine-exposing ones would be to have the directory ITI invoke all the new ITIs as subroutines of itself, and then just have the shell of the newly invoked ITI contact the calling ITI to obtain the "actual message." This alternative method has the formal advantage that all the sub-parties of the session are, formally, subroutines of a single ITI. (We chose to present the method above, since it appears somewhat more natural.)

Another method for making sure that protocols are subroutine-exposing, used in Reference [77], mandates a hierarchical tree-like subroutine structure for protocol invocations, and furthermore requires that the hierarchical structure is represented in the SIDs. This convention is sometimes overly restrictive, and also does not always suffice.

It is stressed that $\rho^{\phi\rightarrow\pi}$ UC-emulates ρ with respect to environments that are not identity bounded (namely, environments that can assume any identity when providing inputs to the main ITIs of ρ).

As a special case, we have:

Corollary 23 (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 $(\pi, \textsc{ideal}_\mathcal{F}, \xi)$-compliant, both π and $\textsc{ideal}_\mathcal{F}$ are subroutine-respecting and subroutine-exposing, and π ξ-UC-realizes $\mathcal{F}$. Then protocol $\rho^{\mathcal{F}\rightarrow\pi}$ 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 $\rho^{\mathcal{F}\rightarrow\pi}$ securely realizes $\mathcal{G}$.

Corollary 24 (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, $(\pi, \textsc{ideal}_F, \xi)$-compliant protocol that ξ'-UC-realizes $\mathcal{G}$, and let π be a subroutine-exposing protocol that ξ-UC-realizes $\mathcal{F}$. Then the composed protocol $\rho^{\mathcal{F}\rightarrow\pi}$ ξ'-UC-realizes $\mathcal{G}$.*

Proof. Let $\mathcal{A}$ be an adversary that interacts with ITIs running $\rho^{\mathcal{F}\rightarrow\pi}$. Theorem 22 guarantees that there exists an adversary $\mathcal{A}'$ such that $\textsc{exec}_{\rho,\mathcal{A}',\mathcal{E}} \approx \textsc{exec}_{\rho^{\mathcal{F}\rightarrow\pi},\mathcal{A},\mathcal{E}}$ for any environment $\mathcal{E}$. Since ρ ξ'-UC-realizes $\mathcal{G}$ (i.e., ρ UC-realizes $\mathcal{G}$ with respect to ξ'-identity-bounded environments), there exists a simulator $\mathcal{S}$ such that $\textsc{exec}_{\text{IDEAL}_\mathcal{G},\mathcal{S},\mathcal{E}} \approx \textsc{exec}_{\rho,,\mathcal{A}_\mathcal{F},\mathcal{E}}$ for any ξ'-identity-bounded $\mathcal{E}$. Using the transitivity of indistinguishability of ensembles we obtain that $\textsc{exec}_{\text{IDEAL}_\mathcal{G},\rho^{\mathcal{F}\rightarrow\pi},\mathcal{S},\mathcal{E}} \approx \textsc{exec}_{\rho^{\mathcal{F}\rightarrow\pi},\mathcal{A},\mathcal{E}}$ for any ξ'-identity-bounded environment $\mathcal{E}$. □

6.2 Proof of Theorem 22

We start with an outline of the proof, in Section 6.2.1. The full proof appears in Section 6.2.2.

The proof here is significantly more complex than that of Section 2.3. One source of complication is the dynamic nature of the boundaries between protocol sessions. Another is the need to handle replacing multiple sessions of ϕ by sessions of π. In particular, the composite simulator $\mathcal{S}$ needs to be able to identify, for each incoming input and backdoor message, to which session of π (or ϕ) this message pertains. Reducing the environment for a single session of π (or ϕ) to an environment that distinguishes $\rho^{\phi\rightarrow\pi}$ from ρ faces similar complications. In contrast, in the model of Section 2.3 these steps are straightforward.

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

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

The general outline of the proof proceeds as follows. The fact that π ξ-UC-emulates ϕ guarantees that there exists an adversary (or, *simulator*) $\mathcal{S}_\pi$, such that for any ξ-identity-bounded environment $\mathcal{E}_\pi$:

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

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

Construction of $\mathcal{S}$. Simulator $\mathcal{S}$ operates as follows. Recall that $\mathcal{E}$ expects to interact with ITIs running ρ. The idea is to separate the interaction between $\mathcal{E}$ and the backdoor tapes of the protocol ITIs (via $\mathcal{S}$) into several parts. To mimic the sending of backdoor messages to the main parties and sub-parties of each session of π, and the receiving of backdoor messages from them, $\mathcal{S}$ runs a session 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 ITIs, mimicking the dummy adversary. (Recall that these ITIs are the members of the extended session of ρ, which are not members of any extended session of π. We call these ITIs side-players.)

More specifically, recall that $\mathcal{E}$ delivers, via the dummy adversary, backdoor messages to the members of ρ, and to the main parties and sub-parties of all sessions of π. 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 a session 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 ITI 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 instances 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 each $\mathcal{S}_\pi$ equals the overall import of the inputs to $\mathcal{S}$ so far.[23]

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 sessions 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 ITI 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 ITI 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 sessions of ϕ never occurs.

If sessions of ϕ recursively use other sessions of ϕ as subroutines, then only the "top-level" sessions of ϕ, namely, only sessions of ϕ that are not subsidiaries of other sessions of ϕ, will have

[23] 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 $p_{\mathcal{S}}(\cdot)$ bounding run-time of $\mathcal{S}$ should be roughly $p_{\mathcal{S}}(\cdot) = t \cdot p_{\mathcal{S}_\pi}(\cdot)$ where $p_{\mathcal{S}_\pi}(\cdot)$ is the polynomial bounding the run-time of $\mathcal{S}_\pi$, and t is the maximum number of sessions of ϕ generated by ρ. See more discussion in the detailed proof.

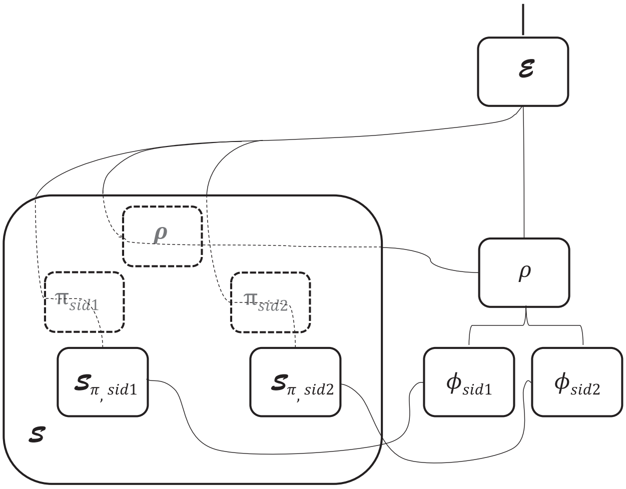

Fig. 8. The operation of $\mathcal{S}$ in the proof of the composition theorem. Inputs from $\mathcal{E}$ that represent backdoor messages directed at the ITIs that are not part of an extended session of π or ϕ are forwarded to the actual recipients. Inputs directed at a session of π are re-directed to the corresponding instance of $\mathcal{S}_\pi$. Backdoor messages from an instance of $\mathcal{S}_\pi$ are directed to the corresponding actual session of ϕ. For graphical clarity, a single box represents a session of a multi-party protocol.

an instance of $\mathcal{S}_\pi$ associated with them. Other sessions of ϕ, if they exist, will be "handled" by the instance of $\mathcal{S}_\pi$ associated with the corresponding top-level session of ϕ.

Figure 8 presents a graphical depiction of the operation of $\mathcal{S}$. A more complete description of the simulator is deferred to the detailed proof.

Analysis of $\mathcal{S}$. Assume that there exists an environment $\mathcal{E}$ that distinguishes with probability ϵ between an interaction with $\mathcal{S}$ and ρ, and an interaction with $\mathcal{D}$ and $\rho^{\phi\rightarrow\pi}$, and let t be an upper bound on the number of sessions of π that are invoked in an interaction. Construct an environment $\mathcal{E}_\pi$ that uses $\mathcal{E}$ to distinguish with probability ϵ/t between an interaction with $\mathcal{D}$ and ITIs running a single session of π, and an interaction with $\mathcal{S}_\pi$ and a single session of ϕ. The construction and analysis of $\mathcal{E}_\pi$ proceeds via a traditional hybrid argument. However, applying the argument in our setting requires some care. Let us explain.

Naively, 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. Therefore, we would have $\rho_0 = \rho^{\phi\rightarrow\pi}$ and $\rho_t = \rho$. This in turn would mean that, for a random $l \in \{1, \ldots, t\}$, $\mathcal{E}$ must distinguish with probability e/t between an interaction with $\mathcal{S}$ and ρ_{l-1}, and an interaction with $\mathcal{S}$ and ρ_l. We would exploit this by having $\mathcal{E}_\pi$ run $\mathcal{E}$, choose a random $l \in \{1, \ldots, t\}$, and making sure that if $\mathcal{E}_\pi$ interacts with $\mathcal{D}$ and $\rho^{\phi\rightarrow\pi}$ (respectively, with $\mathcal{S}_\pi$ and ϕ), then the internal instance of $\mathcal{E}$ sees an interaction with $\mathcal{S}$ and ρ_{l-1} (respectively, ρ_l).

However, while the overall plan is indeed solid, it is not clear how such a 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. Furthermore, the simulator $\mathcal{S}$ might need to behave differently for different top-level sessions of π and ϕ, and might not have the necessary global view either.

We get around this by defining the $t+1$ hybrid experiments differently: We leave the protocol $\rho^{\phi\rightarrow\pi}$ as is, and instead define $t+1$ control functions, where the lth control function replaces the $t-l+1$ last top-level sessions of π back to being sessions of ϕ. Similarly, we modify the simulator

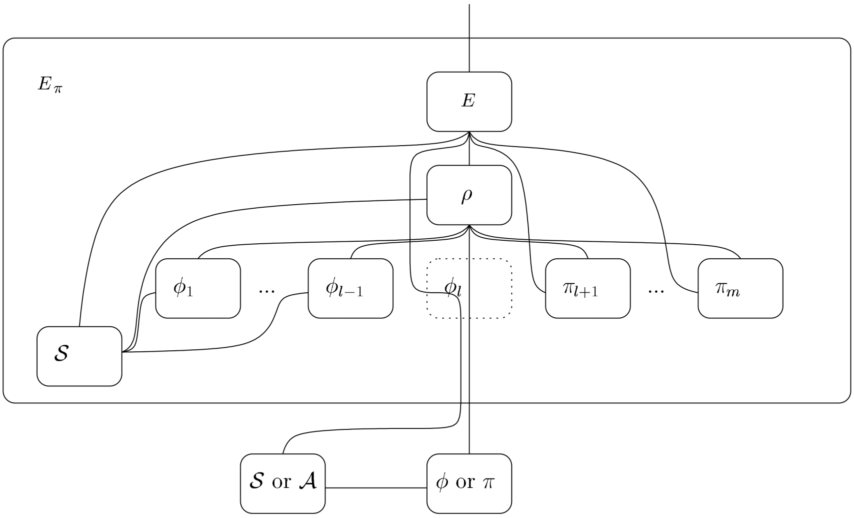

Fig. 9. The operation of $\mathcal{E}_\pi$. An interaction of $\mathcal{E}$ with π is simulated, so that the first $l-1$ sessions of ϕ remain unchanged, the lth session is mapped to the external execution, and the remaining sessions of ϕ are replaced by sessions of π. For graphical clarity, we use a single box to represent a session of a multi-party protocol.

$\mathcal{S}$ so that it asks its environment, for each new top-level session of π or ϕ, whether this session should be simulated or else the members of this session should be treated as side-parties. We let $\hat{\mathcal{S}}$ denote this modified simulator. We then construct $\mathcal{E}_\pi$ so that, for each $l \in \{1, \ldots, t\}$, if $\mathcal{E}_\pi$ interacts with $\mathcal{D}$ and $\rho^{\phi \rightarrow \pi}$ (respectively, with with $\mathcal{S}_\pi$ and ϕ), then the internal instance of $\mathcal{E}$ sees the $(l-1)$th (respectively, lth) hybrid experiment.

Specifically, $\mathcal{E}_\pi$ chooses a random $l \in \{1, \ldots, t\}$, and then runs a simulated execution of $\mathcal{E}$ with $\hat{\mathcal{S}}$ and the lth control function sketched above, with the following exceptions. First, whenever $\hat{\mathcal{S}}$ asks whether to simulate a top-level session of π or ϕ, $\mathcal{E}_\pi$ answers positively only if this is one of the first l top-level sessions of π or ϕ.

Next, $\mathcal{E}_\pi$ 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. That is, whenever some simulated side-player passes an input x to a main party or sub-party 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. subroutine-outputs generated by an actual ITI running π are treated like subroutine-outputs from ϕ_l to the corresponding simulated side-player.

Similarly, whenever the simulated adversary $\hat{\mathcal{S}}$ passes input value v to the session of $\mathcal{S}_\pi$ that corresponds to ϕ_l, $\mathcal{E}_\pi$ passes input v to the actual adversary it interacts with. Any subroutine-output obtained from the actual adversary is passed to $\hat{\mathcal{S}}$ as an subroutine-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 9 presents a graphical depiction of the operation of $\mathcal{E}_\pi$.

6.2.2 A Detailed Proof. We proceed with a detailed proof of Theorem 22, substantiating the above outline.

Simulator $\mathcal{S}$

Initially, no instances of $\mathcal{S}_\pi$ exist. At each activation, if the total import received so far from $\mathcal{E}$, minus the overall import given out to backdoor tapes, minus the total lengths of all inputs and all incoming messages on the backdoor tape, is smaller than the security parameter then $\mathcal{S}$ halts. Else:

1. When activated with input $(i, (m, M, i'))$ (coming from the environment $\mathcal{E}$), where i, i' are the incoming and outgoing import, m is a message, and $M = (id, c)$ is an ITI with identity $id = (sid, pid)$ and code c, do:
   - (a) Activate each running 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. Next, when an instance s of $\mathcal{S}_\pi$ sends a query to the directory ITI of session s of ϕ, $\mathcal{S}$ sends the query to that 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 $(2i'', (m, M, i'))$, and follow the instructions of this instance of $\mathcal{S}_\pi$ for the rest of this activation. The value i'' is set to the overall import of the inputs received by $\mathcal{S}$ so far, minus the overall import given to this instance of $\mathcal{S}_\pi$ so far. (Jumping ahead, we note that the reason for giving to this instance of $\mathcal{S}_\pi$ import $2i''$, rather than i'', is to make sure that the distinguishing environment $\mathcal{E}_\pi$ remains balanced.)
   - (c) If all instances of $\mathcal{S}_\pi$ answer negatively, then:
     - (i) If M is a main ITI of a session s' of π, then invoke a new instance of $\mathcal{S}_\pi$ with SID s', activate this instance, and follow its instructions as in Step 1b.
     - (ii) Else deliver the backdoor message m to ITI M, subject to the run-time 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 session s of ϕ whether M is in its directory. If the directory ITI of any session 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 then:
     - (i) If M is a main ITI of a session s' of ϕ, then invoke a new instance of $\mathcal{S}_\pi$, with SID s', activate this session of $\mathcal{S}_\pi$, and follow its instructions as in Step 2a.
     - (ii) Else forward (M, m) to $\mathcal{E}$, subject to the run-time restrictions of the dummy adversary.

Fig. 10. The simulator 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 $\rho^\pi = \rho^{\phi \rightarrow \pi} = \textsc{uc}(\rho, \pi, \phi)$ be the composed protocol. Let $\mathcal{S}_\pi$ be the simulator for a single instance of π, i.e., $\textsc{exec}_{\phi, \mathcal{S}_\pi, \mathcal{E}_\pi} \approx \textsc{exec}_{\mathcal{D}, \pi, \mathcal{E}_\pi}$ holds for any ξ-identity-bounded environment $\mathcal{E}_\pi$. Simulator $\mathcal{S}$ uses $\mathcal{S}_\pi$ and is presented in Figure 10.

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 $t \le n$, where n is the import of the input to $\mathcal{S}$. This is so since $\mathcal{E}$ is balanced.)[24]

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)). 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

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

We show that

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

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

Toward constructing $\mathcal{E}_\pi$: The hybrid experiments. In preparation to presenting $\mathcal{E}_\pi$, define the following hybrid experiments. Consider an execution of protocol $\rho^{\phi\longrightarrow\pi}$ 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 $\rho^{\phi\longrightarrow\pi}$ 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 ITIs running π need not be aware of t. Also, since $\mathcal{E}$ is PPT, t is polynomial in $k, |z|$.)

Recall that $C^{\rho^{\phi\longrightarrow\pi},\mathcal{S}}$ is the control function, defined in the model of protocol execution (Section 4.1), in the case where the protocol is $\rho^{\phi\longrightarrow\pi}$and the adversary is $\mathcal{S}$. For $0 \le l \le t$, let the l-hybrid control function be the function $C^{\rho^{\phi\longrightarrow\pi},\hat{\mathcal{S}},l}$ that is identical to the control function $C^{\rho^{\phi\longrightarrow\pi},\mathcal{S}}$, with the following exceptions:

(1) The external-write requests to input tapes of the main ITIs of the first l top-level sessions of π to be invoked within the test session of $\rho^{\phi\longrightarrow\pi}$ are redirected (back) to the corresponding sessions of ϕ. The external-write requests to the input tapes of the main ITIs 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 session of $\rho^{\phi\longrightarrow\pi}$; then, given an external-write request made by some ITI to the input tape of ITI (π, id) where $id = (sid_i, pid)$ for $i \le l$ and some pid, the control function writes the requested value to the input tape of ITI (ϕ, id). 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 members of extended sessions of ϕ and π.

[24] 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, in turn, is done to account for the following two facts: (a) Our model allows different sessions of ϕ to have very different imports, and $\mathcal{S}$ does not know the imports of the different sessions. (b) The view of each instance of $\mathcal{S}_\pi$ should be consistent with an execution where its environment is balanced. Indeed, setting the import of each instance of $\mathcal{S}_\pi$ to the maximal possible value, namely, n, will resolve this issue. 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 ITIs 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 session 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 sessions of ρ are known in advance, or alternatively where all sessions of ϕ have roughly equal imports, or alternatively where the run-time of $\mathcal{S}_\pi$ depends only on the import of the ITIs running ϕ, and where in addition the import of the sessions of ϕ is not larger than the import of ρ, the polynomial bounding the complexity of $\mathcal{S}$ becomes the maximum of the polynomials bounding the run-times 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.

(2) Similarly, whenever $(\phi, (sid_i, pid))$ requests to pass subroutine-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.

(3) The adversary invoked by $C^{\rho^{\phi\rightarrow\pi},\hat{\mathcal{S}},l}$, denoted $\hat{\mathcal{S}}$, is identical to $\mathcal{S}$ with the following two exceptions:

 (a) When $\hat{\mathcal{S}}$ receives an input that is aimed at a new session of π, it asks its environment whether this session should be simulated. If yes, then $\hat{\mathcal{S}}$ proceeds as $\mathcal{S}$ with respect to this session (see Step (1(c)i in Figure 10). If no, then from now on all the members of this session are treated like side-parties (Step 1(c)ii there).

 (b) Similarly, when $\hat{\mathcal{S}}$ receives a backdoor message coming from a member of a new session of ϕ, it asks its environment whether this session should be simulated (see Step 2(b)i there). If yes, then $\hat{\mathcal{S}}$ proceeds as $\mathcal{S}$ with respect to this session. If no, then from now on all the members of this session are treated like side-parties (Step 2(b)ii there).

The definition of the hybrid models, as well as that of the environment $\mathcal{E}_\pi$ below, make crucial use of the following facts: (a) It is possible to determine, at any moment during the execution, which ITIs are members of which extended top-level session of ϕ and π. (b) Furthermore, each ITI is a subsidiary of at most one session of either π of ϕ, and membership is determined at the moment of invocation. These facts hold, since π and ϕ are subroutine-respecting.

Construction and analysis of $\mathcal{E}_\pi$. Environment $\mathcal{E}_\pi$ is presented in Figure 11. 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 run-time of $\mathcal{E}_\pi$ can be bounded by the maximum among the polynomials bounding the running times of $\mathcal{E}$, ρ, $\rho^{\phi\rightarrow\pi}$, and $\mathcal{S}$.) Also, since ρ is (π, ϕ, ξ)-compliant, it holds that $\mathcal{E}_\pi$ is ξ-identity-bounded; this holds in spite of the fact that $\mathcal{E}$ need not be identity bounded.

Furthermore, $\mathcal{E}_\pi$ 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) in any execution of $\mathcal{E}_\pi$, the overall import, I_2, received by the main ITIs of any top-level session of π or ϕ, is at most the overall import I_3 that the main ITIs of the test session of ρ receive from $\mathcal{E}$, plus the import I_4 that the members 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, $I_4 \leq I_1$. Since $\mathcal{E}$ is balanced, we also have $I_3 \leq I_1$. Thus, $I_2 \leq I_3 + I_4 \leq 2I_1 \leq I_0$.

The rest of the proof analyzes the validity of $\mathcal{E}_\pi$, demonstrating (5). For $1 \leq l \leq t$, let $\textsc{exec}_{\phi,\hat{\mathcal{S}},\mathcal{E}_\pi^l}(k,z)$ (respectively, $\textsc{exec}_{\pi,\hat{\mathcal{S}},\mathcal{E}_\pi^l}(k,z)$) denote the distribution of $\textsc{exec}_{\phi,\hat{\mathcal{S}},\mathcal{E}_\pi}(k,z)$ (respectively, $\textsc{exec}_{\pi,\hat{\mathcal{S}},\mathcal{E}_\pi}(k,z)$) conditioned on the event that $\mathcal{E}_\pi$ chose value l.

Observe that $\textsc{exec}_{\phi,\hat{\mathcal{S}},\mathcal{E}_\pi^0}(k,z)$ is distributed identically to $\textsc{exec}_{\rho,\mathcal{S},\mathcal{E}}$; indeed, the view of $\mathcal{E}$ is distributed identically in the two executions. Similarly, $\textsc{exec}_{\pi,\hat{\mathcal{S}},\mathcal{E}_\pi^t}(k,z)$ is distributed identically to $\textsc{exec}_{\rho^{\phi\rightarrow\pi},\mathcal{D},\mathcal{E}}$; also here, the view of $\mathcal{E}$ is distributed identically in the two executions. Consequently, inequality Equation (4) can be rewritten as

$$\textsc{exec}_{\phi,\hat{\mathcal{S}},\mathcal{E}_\pi^0}(k,z) - \textsc{exec}_{\pi,\hat{\mathcal{S}},\mathcal{E}_\pi^t}(k,z) \geq \epsilon. \tag{6}$$

Furthermore, for all $l = 1, \ldots, t$, we have

$$\textsc{exec}_{\phi,\hat{\mathcal{S}},\mathcal{E}_\pi^{l-1}}(k,z) = \textsc{exec}_{\pi,\hat{\mathcal{S}},\mathcal{E}_\pi^l}(k,z). \tag{7}$$

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 environment $\mathcal{E}$ with the l-hybrid control function, $C^{\rho^{\phi\to\pi},\hat{\mathcal{S}},l}$. 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) The simulator $\hat{\mathcal{S}}$ asks its environment whether some session, SID, is to be simulated. If session SID is one of the l globally first top-level sessions of π or ϕ to be invoked, then respond positively to $\hat{\mathcal{S}}$. Else, respond negatively. Either way, continue running $\hat{\mathcal{S}}$ without passing this subroutine-output of $\hat{\mathcal{S}}$ to the simulated $\mathcal{E}$.
- (b) Some simulated ITI M passes input x to an ITI M' which is a member 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.
- (c) The simulated $\hat{\mathcal{S}}$ 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.
- (d) The simulated environment $\mathcal{E}$ halts. In this case, $\mathcal{E}_\pi$ outputs whatever $\mathcal{E}$ outputs and halts.

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

(1) When activated for the first time, with input z, choose $l \overset{R}{\leftarrow} \{1, ..., t\}$, and initialize a variable 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 $\hat{\mathcal{S}}$ 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 $\hat{\mathcal{S}}$ to include an subroutine-output v generated by the session of $\mathcal{S}_\pi$ that corresponds to session sid_l of ϕ.
  - (ii) If the new value x was written by another ITI, 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).

Fig. 11. The environment for a single session of π.

Equation (7) follows from inspection of $\mathcal{E}_\pi$ and $\hat{\mathcal{S}}$. Indeed, the view of the simulated $\mathcal{E}$ within $\mathcal{E}_\pi$ is distributed identically in the right and the in left experiments. (This view of $\mathcal{E}$ is also identical to the view of $\mathcal{E}$ when interacting with $\rho^{\phi\rightarrow\pi}$ and $\hat{\mathcal{S}}$ in the l-hybrid experiment, namely, with control function $C^{\rho^{\phi\rightarrow\pi},\hat{\mathcal{S}},l}$, as long as the questions of $\hat{\mathcal{S}}$ regarding which top-level sessions of π and ϕ to simulate are answered as $\mathcal{E}_\pi^l$ does.)

From Equations (6) and (7) it follows that

$$\begin{aligned}|\textsc{exec}_{\pi,\mathcal{D},\mathcal{E}_\pi}(k,z)-\textsc{exec}_{\phi,\mathcal{S}_\pi,\mathcal{E}_\pi}(k,z)| &= \left|\frac{1}{t}\sum_{l=1}^{t}(\textsc{exec}_{\phi,\hat{\mathcal{S}},\mathcal{E}_\pi^{l-1}}(k,z)-\textsc{exec}_{\pi,\hat{\mathcal{S}},\mathcal{E}_\pi^{l}}(k,z))\right| \\ &= \left|\textsc{exec}_{\phi,\hat{\mathcal{S}},\mathcal{E}_\pi^{0}}(k,z)-\textsc{exec}_{\pi,\hat{\mathcal{S}},\mathcal{E}_\pi^{t}}(k,z)\right| \\ &\geq \epsilon/t,\end{aligned}$$

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

6.3 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 9) considers only environments that take external input that contains no information other than its import. The UC theorem still holds even for this variant, with the same proof.

Composing multiple different protocols. The composition theorem (Theorem 22) 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 $\phi_1, \phi_2, \ldots$ are replaced by protocols $\pi_1, \pi_2, \ldots$, 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 $\rho^{\phi_2\rightarrow\pi_2^{\phi_1\rightarrow\pi_1}} = \textsc{uc}(\rho, \textsc{uc}(\pi_2, \pi_1, \phi_1), \phi_2)$ 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 Reference [17] 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 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 C, there exists an ITM μ in C such that, on any input and any random input, the subroutine-output of a single session of μ equals the subroutine-output of (I, C). (Stated in these terms, Proposition 7 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 C if Definition 9 holds when the class of PPT ITMs is replaced with class C, namely, when π, $\mathcal{A}$, $\mathcal{S}$, and $\mathcal{E}$ are taken to be ITMs in C. Then, we have:

Proposition 25. *Let C be a self-simulatable class of ITMs, and let ρ, π, ϕ be protocols in C such that π UC-emulates ϕ with respect to class C. Then protocol $\rho^{\phi \rightarrow \pi}$ UC-emulates protocol ρ with respect to class 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.[25] 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 $\rho^{\mathcal{F} \rightarrow \pi}$ does not UC-emulate ρ. In fact, for *any* PPT π', we have that $\rho^{\mathcal{F} \rightarrow \pi'}$ does not emulate ρ.

To define $\mathcal{F}$, first recall the definition of pseudorandom ensembles of evasive sets, defined in Reference [65] for a related purpose. An ensemble $\mathcal{S} = \{S_k\}_{k \in \mathrm{N}}$ 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 $k \in \mathrm{N}$ and for all $i \in \{0,1\}^k$, a random element $x \overset{\mathrm{R}}{\leftarrow} s_{k,i}$ is computationally indistinguishable from $x \overset{\mathrm{R}}{\leftarrow} \{0,1\}^k$. (b) $\mathcal{S}$ is evasive, that is for any non-uniform PPT algorithm A and for any $z \in \{0,1\}^*$, we have that $\mathrm{Prob}[i \overset{\mathrm{R}}{\leftarrow} \{0,1\}^k : A(z,i) \in s_{k,i}]$ is negligible in k, where $k = |z|$. It is shown in Reference [65], 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 ITI only. Given security parameter k, it first chooses $i \overset{\mathrm{R}}{\leftarrow} \{0,1\}^k$ 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 $r \overset{\mathrm{R}}{\leftarrow} s_{k,i'}$.

Protocol π for realizing $\mathcal{F}$ is simple: Given security parameter k it outputs $i \overset{\mathrm{R}}{\leftarrow} \{0,1\}^k$. Given an input $x \in \{0,1\}^k$, it outputs $r \overset{\mathrm{R}}{\leftarrow} \{0,1\}^k$. 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 $x_1 \overset{\mathrm{R}}{\leftarrow} \{0,1\}^k$ 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, $\rho^{\mathcal{F} \rightarrow \pi}$ never outputs success. In fact, the separation is stronger: F *any* PPT protocol π' that UC-realizes $\mathcal{F}$, protocol $\rho^{\mathcal{F} \rightarrow \pi'}$ outputs `success` only with negligible probability.

7 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

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

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, the bare model does not immediately provide natural ways to represent 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.

Section 7.1 presents a mechanism for expressing various forms of *party corruption,* namely, modeling situations where computational entities deviate from their prescribed protocol in a potentially adversarial way. Section 7.2 presents some useful conventions for writing ideal functionalities.

Section 7.3 then presents ideal functionalities that capture some commonplace abstract models of communication, specifically authenticated, secure, and synchronous communication. Finally, Section 7.4 presents an ideal functionality that captures non-concurrent protocol execution.

7.1 Modeling Party Corruption

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 the face of eavesdroppers, adversarial ("Byzantine") behavior, resilience to viruses and exploits, resilience to side-channel attacks, incoercibility, and so on.

The basic model of protocol execution and the definition of protocol emulation from Section 5 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. 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 identities of all corrupted ITIs, then the emulated and emulating protocol would need to be identical.

We provide the following mechanism for party corruption. This mechanism determines how the behavior of an ITI changes as a result of a particular type of corruption. Furthermore, it specifies how the information regarding which ITIs are currently corrupted is collected and made available to the environment upon request. The idea here is to allow the protocol analyst to determine the amount of information that the environment learns about the corruption operations, which in turn affects 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 denotes the parameters of the corruption. Let $f : \{0,1\}^* \rightarrow \{0,1\}^*$ be an "identity masking" function, which partially masks the ITI's identity. (See more discussion on this function below.)

Say that a protocol π is standard f-revealing corruption if the shell of π proceeds as follows, when running within an ITI with code π, SID s, and PID p: (a) Upon receipt of a backdoor message $(\texttt{corrupt}, cp)$, the shell first passes a $(\texttt{corrupt}, cp, f(\pi, s, p))$ input to a special corruption aggregation ITI $(\pi, s, \mathscr{A})$, where $\mathscr{A}$ is a special identifier. Once the control returns to the corrupted ITI, the behavior of the shell varies according to the specific corruption type (see some examples below). (b) When the body of π instructs to send input to a subroutine ITI M, the shell of π ITI augments the code of M with shell code so that the resulting code M' is a standard f-revealing corruption protocol. (The import needed for running $(\pi, s, \mathscr{A})$ is taken from the adversary. That is, the corruption instruction is required to be associated with some import, n, specified by the protocol. This import is forwarded to $(\pi, s, \mathscr{A})$.)

The corruption aggregation ITI $(\pi, s, \mathscr{A})$ proceeds as follows: When invoked with a corruption notification input $(\texttt{corrupt}, cp)$ from ITI M, ITI $(\pi, s, \mathscr{A})$ records (p, M) and returns control to M. When invoked with a $(\texttt{Report corruption status})$ input from some ITI (presumably, the environment), $(\pi, s, \mathscr{A})$ returns to that ITI the list of all notifications of corruption. (Note that the corruption aggregation machine plays a similar role to the directory machine for subroutine-exposing protocols, with the exception that here the aggregate information is given to the environment rather than to the adversary. This, in particular, means that the corruption aggregation machine needs to be a main ITI of protocol session π, s. Furthermore, when a corrupted ITI M is a member of multiple nested extended sessions of protocols, the protocol designer might want to have M register as corrupted with the aggregation machines of some or all of these protocol sessions.)

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 structure of an extended session of π can diverge from the structure of an extended session 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 ITIs. 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 session of π there exists an ITI in the extended session 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 7.2.

Another natural identity-masking function considers the case where the *sid* is constructed in a hierarchical way and includes the names of parent sessions 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, π can UC-emulate ϕ even when π and ϕ are not standard f-revealing corruption with respect to the same f. In particular, when π is a classic multi-party protocol and ϕ is the ideal protocol for some ideal functionality, the behavior of ϕ in case of party corruption will often be very different than that of π. See more details in Section 7.2.

7.1.1 *Some Corruption Models.*

We sketch how several prevalent party corruption models can be expressed within the current framework. (It should be kept in mind that if the body of the ITI is structured then the behavior of the inner shells might change as a result of the corruption operation. At the same time, the party corruption shell is unaware of, and cannot access, shells that are outer to it.)

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 requirements of being a standard f-revealing corruption protocol for some f. 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 ITIs 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 ITIs. Another standard corruption model only allows the adversary to *observe* the internal state of the corrupted ITI. We call such adversaries passive. Passive corruptions can be captured by setting the reaction of the shell 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 ITI is activated; it also allows the adversary to make the ITI halt. Alternative formulations are possible, where the adversary only learns the current state of a corrupted ITI, and is not allowed to make the ITI halt.

Second, the variant defined here does not allow the adversary to modify *input values* to the ITI. Alternative formulations, where the adversary is allowed to modify the inputs of the corrupted ITIs, 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") leakage 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 Reference [83], the "microwave attacks" of References [18, 21] and the "power analysis" attacks of Reference [49].) These attacks are often dubbed "side-channel" attacks in the literature. Some formalizations of security against such attacks appear in References [60, 95].

This type of attacks can be directly modeled via different reaction patterns of ITIs 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 ITI discloses only some function of its internal state, and furthermore does it only once (per leakage instruction).

One limitation of this modeling is that it only allows the adversary to 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 Reference [20].

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 ITI contributes to a computation, without physically controlling the attacked ITI 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 [37, 38]. We assume that the protocol description includes a "faking algorithm" F. Furthermore, each ITI M has a "caller ITI," which represents either an algorithm or a human user. 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 subroutine-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 ITI P, the ideal functionality forwards this request to the caller ITI of P. If in return the caller ITI inputs an instruction to cooperate, then 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.

Protocol π is incoercible if it is coercion compliant, and in addition it UC-realizes an ideal functionality $\mathcal{F}$ that guarantees ideal incoercibility.

7.2 Writing Ideal Functionalities

This section sets some writing conventions that may help making pseudo-code descriptions of ideal functionalities clearer and more readable. It also presents a number of methodologies for writing ideal functionalities in a way that captures some commonplace security requirements.

Determining the identities of ITIs that provide input and receive subroutine-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 subroutine-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, the code of the dummy parties (see Section 5.3) guarantees that $\mathcal{F}$ sees the extended identities of the ITIs that provide inputs to the dummy parties, and it determines the extended identities of the ITIs that obtain outputs from the fummy parties. 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 that received the input from an ITI with extended identity P." Similarly, we say "generate subroutine-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 $\textsc{ideal}_{\mathcal{F}}$, corruption of parties is modeled as messages written by the adversary on the backdoor tape of the ideal functionality $\mathcal{F}$. (Recall that backdoor messages delivered to the dummy parties are ignored.) 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 ITIs of $\textsc{ideal}_{\mathcal{F}}$. Specifically, 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 PID of a dummy party for the present session of $\textsc{ideal}_{\mathcal{F}}$, $\mathcal{F}$ marks p as corrupted and returns to the adversary all the values received from p and subroutine-output to p so far. In addition, from this point on, input values from the dummy party p are ignored. Instead, $\mathcal{F}$ now takes

from the adversary input instructions for p; that is, upon receipt of a backdoor message `(input, `p`, `v`)`, $\mathcal{F}$ behaves as if it received input v from p. Finally, all subroutine-output values intended for p are sent to the adversary instead.

(2) Upon receiving a `(Report corruption status)` 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., References [23, 63]. Note that here the "granularity" of corruption is at the level of PID for the main ITIs of the session. 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 ITIs. This is so even if the main ITIs 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 to represent more refined corruption mechanisms, such as corruption of specific subroutines or sub-sessions, forward secrecy, leakage, coercion, and so on. Furthermore, ideal functionalities may change their overall behavior depending on the identity or number of corrupted ITIs. We leave further discussion and examples out of scope.

Delayed output. Recall that a subroutine-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 subroutine-outputs generated by distributed protocols are inevitably delayed due to delays in message delivery. One natural way to relax an ideal functionality to allow this slack is to have the functionality "ask for the permission of the adversary" before generating subroutine-output. More precisely, say that an ideal functionality $\mathcal{F}$ sends a delayed output v to ITI 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 subroutine-output to M. If the subroutine-output is public, then the value v is included in the note to the adversary. If the subroutine-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), $\mathcal{F}$ subroutine-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 program; it will then run this program as long as some set of security or correctness properties are satisfied. If a required property is violated, then $\mathcal{F}$ will output an error message to the relevant ITIs. Examples of this use include the signature and encryption functionalities as formalized in Reference [40], or non-interactive zero knowledge as in Reference [72]. 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 7.4.

At first glance, this technique seems problematic in that $\mathcal{F}$ is expected to run algorithms of arbitrary polynomial run-time, whereas $\mathcal{F}$'s own run-time 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 run-time of γ is 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 $\mathcal{F}$. (To make sure that

γ is a new ITI rather than an existing one, $\mathcal{F}$ can e.g., use the same mechanism used in Section 5.2 to guarantee that protocols are subroutine-respecting.)

7.3 Some Communication Models

This subsection captures, 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 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 7.3.1, 7.3.2, and 7.3.3 present ideal functionalities for capturing authenticated, secure, and synchronous communication, respectively.

7.3.1 Authenticated Communication. Ideally authenticated message transmission is the primitive that allows an entity S to send a message m to an entity R, in a way that allows R to verify whether a received message m was indeed sent by S. That is, it is guaranteed that R accepts m as coming from 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.*

A bit more precisely (but still somewhat informally), assume that computational entities are associated with immutable identities, that the sender's identity is S, and that it knows the identity, R, of the intended recipient. We assume that the sender knows R, namely, the identity of the receiver, at the onset of the protocol. The receiver may not have any knowledge of S ahead of time, yet if it accepts a message then it also learns S.

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" that captures and formalizes the functionality and security requirements that are informally discussed in the above two paragraphs.

A first step toward formalization is casting the above informal notions of "entities" and "identities" within the present model. We do this in a natural way: We model the sender as an extended identity (namely, ITI) S that represents the computational process that initiates the transmission of the message. Similarly, the recipient (specified by S) is an extended identity (ITI) R that represents the computational process that should receive the message. It is stressed that S and R do not represent the programs that implement the sending and receiving of the message—rather, they are the "calling entities" that make use of the message transmission service. (Note that this formalism requires the sender to know the receiver's full code, and to fully disclose her own code. We discuss this point later on.)

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

1. Upon invocation, with input $(\texttt{Send}, sid, R, m)$ from ITI S, send backdoor message $(\texttt{Sent}, sid, S, R, m)$ to the adversary.
2. Upon receiving backdoor message $(\texttt{ok}, sid)$: If not yet generated output, then output $(\texttt{Sent}, sid, S, R, m)$ to R.
3. Upon receiving backdoor message $(\texttt{Corrupt}, sid, m', R')$, record being corrupted. Next, if not yet generated output then output $(\texttt{Sent}, sid, S, R', m')$ to R'.
4. On input $(\texttt{ReportCorrupted}, sid)$ from S: If corrupted, output $\texttt{yes}$ to S. Else output $\texttt{no}$.
5. Ignore all other inputs and backdoor messages.

Fig. 12. The Message Authentication functionality, $\mathcal{F}_{\text{AUTH}}$.

We proceed to describe the ideal functionality itself (see Figure 12). In its first activation, the ideal authenticated message transmission functionality, $\mathcal{F}_{\text{AUTH}}$, expects its input, coming from ITI S, to be of the form $(\texttt{Send}, sid, R, m)$. More precisely, the source ITI of this input will be a dummy party for $\mathcal{F}_{\text{AUTH}}$. S is the extended identity of the ITI that provided input to that dummy party. (Indeed, recall that the code of the dummy party included the extended identity of the ITI that invoked it.) The value sid is the SID for $\mathcal{F}_{\text{AUTH}}$, R is the extended identity of the intended recipient ITI, and m is the message. (Of course, the input to $\mathcal{F}_{\text{AUTH}}$ contains also the PID of $\mathcal{F}_{\text{AUTH}}$—which is $\perp$—as well as the value of the forced-write flag and the reveal-sender id flag—which are both 1. These values are ommitted for clarity.)

$\mathcal{F}_{\text{AUTH}}$ then generates a public delayed output $(\texttt{Send}, sid, S, m)$ to R. That is, $\mathcal{F}_{\text{AUTH}}$ first sends this value to the adversary on its backdoor tape. Upon receiving $\texttt{ok}$ on the backdoor tape, $\mathcal{F}_{\text{AUTH}}$, writes $(\texttt{Send}, sid, S, m)$ to the subroutine 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.)

$\mathcal{F}_{\text{AUTH}}$ also accepts an adversarial corruption request. Once corrupted, and as long as the message is not yet delivered to the receiver, $\mathcal{F}_{\text{AUTH}}$ lets the adversary determine the value of the message to be delivered, as well as the identity of the receiver. Finally, in response to a request, coming from S, $\mathcal{F}_{\text{AUTH}}$ reports whether it was corrupted. (As discussed in Section 7.2, the request to report corruption state is part of the "mental experiment" used to assert security, and does not correspond to any actual input generated by some "actual real-life code." Specifically, these requests can be thought of as coming from some shell of the sender S, where as this shell then reports the information onwards until eventually the environment leans the corruption information, potentially in some aggregate form.)

We highlight several points regarding the security guarantees provided by $\mathcal{F}_{\text{AUTH}}$. First, $\mathcal{F}_{\text{AUTH}}$ reveals the contents of the message, 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. Furthermore, the sender may not know the full code of the recipient in advance. However, these issues can be addressed by designing the protocol that uses $\mathcal{F}_{\text{AUTH}}$ so that the ITIs that directly call $\mathcal{F}_{\text{AUTH}}$ and receive subroutine-output from $\mathcal{F}_{\text{AUTH}}$ have code that is generic and contains no sensitive information.)

Second, $\mathcal{F}_{\text{AUTH}}$ 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 it 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. (In the present formalization, the formalism does not distinguish between the case where the sender is corrupted and the case where the receiver is corrupted. More fine-grained formalizations may provide a distinction between the two cases—for instance, if only the intended recipient is corrupted, then only the contents of the message can be adversarially controlled; the identity of the receiver remains R.)[26]

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., Reference [47]).

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 sessions of $\mathcal{F}_{\text{AUTH}}$, and relying on the universal composition theorem for security. This is an important property: It allows different sessions of protocols that use authenticated communication to use different sessions of $\mathcal{F}_{\text{AUTH}}$, thereby making sure that these protocols can be analyzed per session, independently of other sessions. 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 an SID in advance.[27]

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, 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:

Claim 26 ([26]). *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.

[26] Early formulations 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 References [3, 76].

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

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

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

(1) Upon invocation with input $(\mathtt{Send}, sid, R, m)$ from ITI S, send backdoor message $(\mathtt{Sent}, sid, S, R, l(m))$ to the adversary.
(2) Upon receiving backdoor message $(\mathtt{Corrupt}, sid)$, record being corrupted and send m to the adversary.
(3) Upon receiving backdoor message $(\mathtt{ok}, sid, m', R')$: if corrupted then output $(\mathtt{Sent}, sid, S, m')$ to R'. Else If not yet generated output then output $(\mathtt{Sent}, sid, S, m)$ to R.
(4) On input $(\mathtt{ReportCorrupted}, sid)$ from S: If corrupted, output yes to S. Else output no.
(5) Ignore all other inputs and backdoor messages.

Fig. 13. The Secure Message Transmission functionality parameterized by leakage function l.

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, which may come from different contexts and protocols. 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 some long-term-binding module (say, a digital signature algorithm along with public-key infrastructure) cannot be subroutine-respecting, unless each session of the protocol uses a new session of the long-term-binding module—which does not capture reality.

To allow for modular analysis of such situations, a number of mechanisms exist in the literature for "decomposing" a protocol where multiple sessions of $\mathcal{F}_{\text{AUTH}}$ (or of a protocol that realizes $\mathcal{F}_{\text{AUTH}}$) jointly use a single session of a long-term authentication module, into multiple smaller (and overlapping) components, where each component consists of a single session of $\mathcal{F}_{\text{AUTH}}$, along with a long-term authentication module. One can then use the UC theorem to assert the security of the desired system (which consist of multiple session of the protocol realizing $\mathcal{F}_{\text{AUTH}}$ where all sessions use the same session of the long-term authentication module). We leave these mechanisms out of scope; see details in References [46, 86].

7.3.2 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 13. 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 $l : \{0,1\}^* \rightarrow \{0,1\}^*$ 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.)

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

Another natural difference between $\mathcal{F}_{\text{SMT}}$ and $\mathcal{F}_{\text{AUTH}}$ is that $\mathcal{F}_{\text{SMT}}$ allows the adversary, upon corruption, to first learn the message m, and only then to determine m', R'. Also note that here, in contrast to the case of $\mathcal{F}_{\text{AUTH}}$, the behavior of $\mathcal{F}_{\text{SMT}}$ upon corruption is meaningful even if the corruption takes place after the execution of the protocol has completed.

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, 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 a session of $\textsc{ideal}_{\mathcal{F}_{\text{AUTH}}}$ 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 27. *If E is semantically secure for domain D as in References [61, 70], then π_E UC realizes $\mathcal{F}^{l_D}_{\text{SMT}}$ in the presence of non-adaptive corruptions.*

Furthermore, if E is non-committing (as in Reference [35]) then π_E UC-realizes $\mathcal{F}^{l_D}_{\text{SMT}}$ 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 sessions 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 Reference [43] 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 References [39, 103]. Using the UC with joint state mechanism [46], one can still design and analyze protocols that employ multiple independent sessions of $\mathcal{F}_{\text{SMT}}$, in spite of the fact that all these sessions are realized by a single (or few) sessions of an encryption protocol.

7.3.3 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 References [75, 82, 104]. This subsection provides an alternative way of capturing synchronous communication within the UC framework: It shows 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 abstricipants 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.

The first requirement is not essential, i.e., there exist meaningful notions of synchronous communication that do not imply common knowledge of the round number. Still, for simplicity, we choose to express the stronger variant.

Finally, it is stressed 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.

$\mathcal{F}_{\text{SYN}}$, presented in Figure 14, 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 session of $\mathcal{F}_{\text{SYN}}$ by other means. (Said otherwise, we separate out the task of letting the ITIs in $\mathcal{P}$ know about *sid*. Indeed, there can be a variety of mechanisms for this task, that make sense in different settings, and that provide different liveness guarantees.)

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 $(\mathsf{Send}, \mathit{sid}, M)$ from party $S \in \mathcal{P}$, $\mathcal{F}_{\text{SYN}}$ interprets M as a list of

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 $r \leftarrow 1$, a boolean $s_P^\rho \leftarrow 0$, and a set $N_P^\rho \leftarrow \emptyset$ for all $\rho \geq 0$ and all $P \in \mathcal{P}$. (s_P^ρ will hold whether P sent messages for round r, and N_P^ρ will hold the list of messages sent in round ρ to party P. That is, $N_P^\rho = \{(S, m)\}$ where $S \in \mathcal{P}$ is the sender and m is a message.)

(2) Upon receiving input $(\texttt{Send}, sid, M)$ from party $S \in \mathcal{P}$, where M is a set of pairs (R, m), do:

  (a) Set $s_S^r \leftarrow 1$.

  (b) For each pair (R, m) in M , add (S, m) to N_R^r.

  (c) If all uncorrupted parties in $\mathcal{P}$ have already provided their messages for round r (i.e., if $s_P^r = 1$ for all uncorrupted $P \in \mathcal{P}$), then increment the round number: $r \leftarrow r + 1$.

  (d) Send (sid, S, M, r) to the adversary.

(3) Upon input $(\texttt{Receive}, sid, r')$ from $R \in \mathcal{P}$, do: If $r' < r$ (i.e., r' is a completed round) then output $(\texttt{Received}, sid, N_R^{r'})$ to R. Else (i.e. $r' \geq r$) output $(\texttt{Round}$ r $\texttt{incomplete})$ to P.

(4) Upon receiving a backdoor message $(\texttt{corrupt}\ P)$ for some $P \in \mathcal{P}$, mark P as corrupted.

Fig. 14. The synchronous communication functionality, $\mathcal{F}_{\text{SYN}}$.

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.) At this point $\mathcal{F}_{\text{SYN}}$ also checks whether all uncorrupted parties have already sent their messages for this round. If so, then it marks the current round as complete and increments the round number.

Given an input $(\texttt{Receive}, sid, r')$ from a party $R \in \mathcal{P}$, where r' is a round number, $\mathcal{F}_{\text{SYN}}$ proceeds as follows: If round r' is completed, then $\mathcal{F}_{\text{SYN}}$ returns to R the messages sent to it in round r'. If round r' is not yet complete, then $\mathcal{F}_{\text{SYN}}$ reports this fact to R.

Upon receiving a $(\texttt{Corrupt}, P)$ from the adversary, for some $P \in \mathcal{P}$, $\mathcal{F}_{\text{SYN}}$ marks P as corrupted.[28]

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 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.

[28] 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 Reference [82], where a different fix to the one used here is proposed.

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 a single session of some protocol, π. Here all parties of a session of π use a single session of $\mathcal{F}_{\text{SYN}}$. This session can be invoked by any of the parties. Here, it is assumed that all parties of the session of π know the SID of the session of $\mathcal{F}_SYN$ in use. (Say, the session of F_{SYN} is derived from the SID of the session of π, along with the PIDs of the parties.)

Each party of a session 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)`, then this means that some parties have not completed this round yet. In this case, π does nothing (thus returning the control to the environment).

Discussion. We make the following additional observtions:

- 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 communication 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.
- Each session of $\mathcal{F}_{\text{SYN}}$ guarantees synchronous message delivery only within the context of the messages sent using that session. Delivery of messages sent in other ways (e.g., directly or via other sessions 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.
- Even when using $\mathcal{F}_{\text{SYN}}$, the inputs to the parties are received in an “asynchronous” 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.
- 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.
- To capture the case where the parties learn about the session of $\mathcal{F}_{\text{SYN}}$ from $\mathcal{F}_{\text{SYN}}$ itself, can add to $\mathcal{F}_{\text{SYN}}$ an initial stage where $\mathcal{F}_{\text{SYN}}$ notifies all (or some) of the parties in $\mathcal{P}$ that the execution started. Note that, with this initial stage in place, $\mathcal{F}_{\text{SYN}}$ may transfer control to the environment; still, as discussed above, delivery of all messages is still guaranteed as long as $\mathcal{F}_{\text{SYN}}$ does not wait for a response from the environment or the adversary.

On composing $\mathcal{F}_{\text{SYN}}$*-hybrid protocols.* Within the present framework, where protocols are bound to be subroutine-respecting, a session of $\mathcal{F}_{\text{SYN}}$ cannot be used as a subroutine by two different protocols sessions (π, sid) and (π', sid'), unless one session is a subroutine of the other. This means that if one wants to consider a (perhaps composite) protocol where the communication is synchronous across all parties of a session of the protocol, then one 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 [34, 46]. The JUC theorem provides a way (within the present framework) to analyze individual sessions of some protocol π, where each session uses its own session of $\mathcal{F}_{\text{SYN}}$, and then argue that the overall behavior does not change even if all sessions of π use the same session of $\mathcal{F}_{\text{SYN}}$. (Here care must be taken to account for protocols that take different number of rounds to complete.)

Relaxations. The reliability and authenticity guarantees provided within a single session 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 References [56, 62, 81]. 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., References [16, 45]).

7.4 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 properties of protocols within the present framework. That is, it presents 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, the argument below will 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 environment 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: (a) upon initial invocation, each party of π_{NC} calls $\mathcal{F}_{\text{NC}}$ with a $(\texttt{init}, s)$ input where s is the same SID as the local one, (b) any external-write, made by any ITI in the extended session of π_{NC} to the backdoor tape of the adversary, is replaced be an input to $\mathcal{F}_{\text{NC}}$; similarly, outputs coming from $\mathcal{F}_{\text{NC}}$ are treated like messages coming on the backdoor tape. Incoming messages from the actual backdoor tape are ignored. (Formally, these transformations can be implemented via appropriate shell code.)

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

(1) $\mathtt{init}$ inputs are ignored, except for the inital one, Upon initial invocation (due to an $(\mathtt{init}, s)$ input from some party, Report s to the adversary. (Subsequent $\mathtt{init}, s)$ input are ignored.)

(2) When receiving message $(\mathtt{Start}, \hat{\mathcal{A}})$ from the adversary, where $\hat{\mathcal{A}}$ is the code of an ITM (representing an adversary), invoke $\hat{\mathcal{A}}$ and change state to $\mathtt{running}$.

(3) When receiving input $(\mathtt{Backdoor}, m)$ from party (i.e., ITI) P, verify that the state is $\mathtt{running}$, 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 $(\mathtt{Backdoor}, m')$ 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.

Fig. 15. The non-concurrent communication functionality, $\mathcal{F}_{\text{NC}}$.

Functionality $\mathcal{F}_{\text{NC}}$ is presented in Figure 15. It expects to be first ativated by an input $(\mathtt{init}, s)$. It then notifies the adversary of the SID s 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 (these messages come from the parties of this extended session 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, 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 Reference [23]. Recall the security definition of Reference [23], 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 Reference [23] 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 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 one in Reference [23] 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 28. Let π and ϕ be PPT protocols and let ξ be a PPT predicate. Say that π ξ-NC-emulates ϕ if for any PPT adversary $\mathcal{A}$ there exists a PPT adversary $\mathcal{S}$ such that for any non-concurrent, balanced, ξ-identity-bounded PPT environment $\mathcal{E}$, it holds that $\textsc{exec}_{\phi,\mathcal{S},\mathcal{E}} \approx \textsc{exec}_{\pi,\mathcal{A},\mathcal{E}}$.

We argue (informally) that NC-emulation captures the essence of the notion of Reference [23]. In particular, we conjecture that the existing security analysis of known protocols (e.g., the protocol of Reference [67]; see Reference [63]) 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.

With this perspective in mind, we formalize the equivalence between the security notion of Reference [23] and UC-emulation in the presence of $\mathcal{F}_{\rm NC}$, via the following proposition:

Proposition 29. *Let π and ϕ be PPT protocols and let ξ be a PPT predicate. Then $\pi_{\rm NC}$ ξ-UC-emulates ϕ if and only if π ξ-NC-emulates ϕ.*

Notice that Proposition 29, together with the UC theorem, provide an alternative (albeit somewhat indirect) formulation of the non-concurrent composition theorem of Reference [23]. In fact, the present result is significantly more general, since it applies also to reactive protocols with multiple rounds of inputs and outputs.

Proof. First show that if $\pi_{\rm NC}$ ξ-UC-emulates ϕ then π ξ-NC-emulates ϕ. Let $\mathcal{A}$ be an adversary (geared toward interacting with π in a non-concurrent environment). We need to show a simulator $\mathcal{S}_{\mathcal{A}}$ such that $\textsc{exec}_{\pi,\mathcal{A},\mathcal{E}} \approx \textsc{exec}_{\phi,\mathcal{S}_{\mathcal{A}},\mathcal{E}}$ for any non-concurrent environment $\mathcal{E}$.

Construct $\mathcal{S}_{\mathcal{A}}$ in two steps. First, consider the adversary $\hat{\mathcal{A}}$, which is the version of $\mathcal{A}$ geared toward working with $\pi_{\rm NC}$. Specifically, upon initial activation, $\hat{\mathcal{A}}$ forwards the code $\mathcal{A}$ to $\mathcal{F}_{\rm NC}$, which is part of $\pi_n c$). From then on, $\hat{\mathcal{A}}$ behaves like the dummy adversary. Observe that, as long as $\mathcal{E}$ is non-concurrent, the ensembles $\textsc{exec}_{\pi,\mathcal{A},\mathcal{E}}$ and $\textsc{exec}_{\pi_{\rm NC},\hat{\mathcal{A}},\mathcal{E}}$ are identical. Now let $\mathcal{S}_{\mathcal{A}}$ be the simulator for $\pi_{\rm NC}$ and $\hat{\mathcal{A}}$, that is $\mathcal{S}_{\mathcal{A}}$ is such that $\textsc{exec}_{\pi_{\rm NC},\hat{\mathcal{A}},\mathcal{E}} \approx \textsc{exec}_{\phi,\mathcal{S}_{\mathcal{A}},\mathcal{E}}$ for all $\mathcal{E}$. This direction follows.

It remains to show that if π ξ-NC-emulates ϕ then $\pi_{\rm NC}$ ξ-UC-emulates ϕ. In fact, it suffices to demonstrate that $\pi_{\rm NC}$ UC-emulates ϕ with respect to the dummy adversary. Furthermore, using Claim 14, it suffices to show that $\pi_{\rm NC}$ 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 (that expects to interact with $\pi_{\rm NC}$ and the dummy adversary). Let $\hat{\mathcal{A}}$ denote the adversary code that is given by $\mathcal{E}$ in response to the first backdoor message from $\mathcal{F}_{\rm NC}$ (forwarded by the dummy adversary). Note that $\mathcal{E}$ generates this code before obtaining any output from any party.

Next, consider the following environment $\mathcal{E}_{\rm NC}$ $\mathcal{E}_{\rm NC}$ runs $\mathcal{E}$. When $\mathcal{E}$ generates an input to a main party of $\pi_{\rm NC}$, $\mathcal{E}_{\rm NC}$ forwards this inputs unchanged. Similarly, outputs from the main parties of $\pi_n c$ are forwarded to $\mathcal{E}$ unchanged. Inputs from $\mathcal{E}$ to the adversary are ignored, except for the first input that is directed at the backdoor tape of $\mathcal{F}_{\rm NC}$; this input (which contains the code $\hat{\mathcal{A}}$) is forwarded by $\mathcal{E}_{\rm NC}$ to its adversary. Once $\mathcal{E}_{\rm NC}$ receives an output value from its adversary, it hands this value to $\mathcal{E}$, outputs whatever $\mathcal{E}$ outputs, and halts.

Clearly, $\mathcal{E}_{\rm NC}$ is a non-concurrent environment. Therefore, since π ξ-NC-emulates ϕ, there exists a simulator $\mathcal{S}$ such that $\textsc{exec}_{\pi,\hat{\mathcal{A}},\mathcal{E}_{\rm NC}} \approx \textsc{exec}_{\phi,\mathcal{S},\mathcal{E}_{\rm NC}}$. However, $\textsc{exec}_{\pi,\hat{\mathcal{A}},\mathcal{E}_{\rm NC}}$ is identical to $\textsc{exec}_{\pi_{\rm NC},\mathcal{D},\mathcal{E}}$, 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. It then follows that $\textsc{exec}_{\phi,\mathcal{S},\mathcal{E}_{\rm NC}}$ is distributed identically to $\textsc{exec}_{\phi,\hat{\mathcal{S}},\mathcal{E}}$; indeed, the view of $\mathcal{E}$

is the same in the two executions. It follows that $\textsc{exec}_{\pi_{\mathrm{NC}},\mathcal{D},\mathcal{E}} = \textsc{exec}_{\pi,\hat{\mathcal{A}},\mathcal{E}_{\mathrm{NC}}} \approx \textsc{exec}_{\phi,\mathcal{S},\mathcal{E}_{\mathrm{NC}}} = \textsc{exec}_{\phi,\hat{\mathcal{S}},\mathcal{E}}$, namely, π ξ_{NC}-UC-emulates ϕ. □

Modeling partial concurrency. Finally, note 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 $\mathcal{F}_{\mathrm{NC}}$ 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 concurrent executions of other components (or of sub-components within a component) is allowed.

APPENDIX

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 References [25, 27, 28, 29, 30] 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 [120], 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 [67], 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 [55]. 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 Reference [55] are addressed is central to the present framework.

The first rigorous general definitional framework for secure protocols is due to Goldwasser and Levin [69] and was followed shortly by the frameworks of Micali and Rogaway [96] and Beaver [12]. In particular, the notion of "reducibility" in Reference [96] 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 References [12, 96] formalisms only address security in restricted settings; in particular, they do not deal with computational issues.) References [12, 69, 96] are surveyed in Reference [23] in more detail.

The frameworks of References [12, 69, 96] concentrate on *synchronous* communication. Also, although in Reference [67] 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 Reference [16]. A system model and notion of security for reactive functionalities is sketched in Pfitzmann and Waidner [110].

The first ideal-process based definition of computational security against resource bounded adversaries is given in Reference [31]. In Reference [23] the framework of Reference [31] is strengthened to handle secure composition. In particular, Reference [23] defines a general composition operation, called *modular composition,* which is a non-concurrent version of universal composition. That is, only a single protocol session can be active at any point in time. (See more details in Section 7.4.) In addition, security of protocols in that framework is shown to be preserved under modular composition. A closely related formulation appears in Reference [63, Section 7.7.2].

Reference [23] 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 [73] provides a rigorous treatment of reactive functionalities. Dodis and Micali [54] build on the definition of Micali and Rogaway [96] 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 sessions 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 [87, 88] develop a process calculus, based on the π-calculus of Milner [99, 100], that incorporates random choices and computational limitations on adversaries. (In Reference [101] 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 [50], Chor and Nelson [51], Bird et al. [19], and Canetti and Krawczyk [41].

Concurrent work. The framework of Pfitzmann, Schunter and Waidner [107, 108] is the first to rigorously address *concurrent* universal composition in a computational setting. (This work is based on the sketches in Reference [110]). They define security for reactive functionalities in a synchronous setting and prove that security is preserved when a *single* session of a subroutine protocol is composed concurrently with the calling protocol. An extension of the References [107, 108] framework to asynchronous networks appears in Reference [109].

At high level, the notion of security in References [107–109], 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 Reference [91], 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 sessions* 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 sessions 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 [5] extend the framework of Reference [109] to deal with the case where the number of parties and protocol sessions depends on the security parameter (still it is otherwise static as in Reference [108]). In that framework, they prove that reactive simulatability is preserved under universal composition. The Reference [6] formulation returns to the original approach where the number of entities and protocol sessions is fixed irrespective of the security parameter.

Mateus, Mitchell and Scedrov [93] and Datta, Küsters, Mitchell, and Ranamanathan [85] (see also Reference [53]) extend the frameork of References [87, 88] 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 Reference [25]) and reactive simulatability (as defined in Reference [6]). Tight correspondence between the Reference [93] notion of security and the one defined here is demonstrated in Almansa [4]. All of these frameworks postulate a static execution model that is most similar to the one in Section 2.

Canetti et al. [32] extend the probabilistic I/O automata of Lynch, Segala and Vaandrager [92, 115] 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 [97] provide an alternative, simplified formalism for composable simulation-based security of protocol. The formalism, which is a generalization of Kahn networks [80], allows for equational (rather than temporal) representation and analysis of protocols and their security.

Küsters, Tüngerthal, and Rausch [84, 86] formulate an ITM-based model of computation that allows for defining UC-style notions of security. The model uses a combination of traditional process calculus elements that allow for simple representation and argumentation about the composition of systems along fixed and pre-determined system boundaries, with more dynamic communication and addressing structure within each individual system. This model (called the IITM model) can be seen as a midpoint between the restricted model of Section 2 and the full-fledged model presented here. It is useful when one wants to express systems with dynamic internal structure, but at the same time one only needs to apply the universal composition theorems to protocols and systems with boundaries that are fixed in advance (as in process calculus, or alternatively as in the restricted model). The IITM model is, however, not useful in situations where one wants to apply the composition theorem to protocols whose structure is determined dynamically at run-time (such as, e.g., decentralized or peer-to-peer systems). Note that, while the IITM model takes a different approach to bounding runtime than done in this work, our notion of *import* of messages is influenced by the modeling of Reference [84].

Hofheinz, Müller-Quade, and Unruh [79] give an alternative definition of polynomial time ITMs (see discussion in Section 3.3.4). Hofheinz and Shoup [77] 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 Reference [79]. 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 26).

Nielsen [104], Hofheinz and Müller-Quade [75], and Katz et al. [82] formulate synchronous variants of the UC framework. Wikström [118, 119], as well as Canetti, Cohen, and Lindell [33] 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 Reference [24, Appendix B of Version of 2020].) The first versions of this work [24, 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 [24, 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 [24, 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 next version [24, Version of 2018] further improves the treatment of identities and subroutine-respecting protocols, introduces subroutine-exposing protocols as a tool to fix a flaw in the composition theorem, generalizes the notion of polynomial runtime (using import), simplifies the definition of systems of ITIs and the model of protocol execution, and introduces the simplified, "static" model of computation of Section 2.

The next version [24, Version of 2020] further spells out the model of Section 2, as well as the mechanism of bodies and shells and its use to express subroutine-respecting and subroutine-exposing protocols, as well as the composition operation and party corruption.

ACKNOWLEDGMENTS

Much of the motivation for undertaking this project, and many of the ideas that appear here, come from studying secure key-exchange protocols together with Hugo Krawczyk. I thank him for this long, fruitful, and enjoyable collaboration. I am also grateful to Oded Goldreich who, as usual, gave me both essential moral support and invaluable technical, presentational, and practical advice. In particular, Oded's dedicated advice greatly improved (and vastly reshaped) the presentation of this article. 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, Christian Badertscher, Mihir Bellare, Ivan Damgaard, Marc Fischlin, Shafi Goldwasser, Rosario Gennaro, Shai Halevi, Julia Hesse, 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, Alley Stoughton, Paul Syverson, Björn Tackmann, Rita Vald, Mayank Varia, Michael Waidner, Vassilis Zikas. In particular, it was Daniele Micciancio 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 the dedicated referees who have meticulously read the article over and over, pointing out scores of bugs—small and large.

REFERENCES

[1] Martín Abadi and Andrew Gordon. 1999. A calculus for cryptographic protocols: The Spi calculus. *Info. Comput.* 148, 1 (1999), 1–70.

[2] Martín Abadi and Phillip Rogaway. 2000. Reconciling two views of cryptography (The computational soundness of formal encryption). In *Proceedings of the IFIP International Conference on Theoretical Computer Science (IFIP-TCS'00) (Lecture Notes in Computer Science)*, Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses, and Takayasu Ito (Eds.), Vol. 1872. Springer-Verlag, 3–22.

[3] Masayuki Abe and Serge Fehr. 2004. Adaptively secure Feldman VSS and applications to universally composable threshold cryptography. In *Proceedings of the 24th Annual International Cryptology Conference (CRYPTO'04).* 317–334. DOI: https://doi.org/10.1007/978-3-540-28628-8_20

[4] Jesús F. Almansa. 2005. The full abstraction of the UC framework. *IACR Cryptol. ePrint Arch.* 2005 (2005), 19. Retrieved from http://eprint.iacr.org/2005/019.

[5] Michael Backes, Birgit Pfitzmann, and Michael Waidner. 2004. A general composition theorem for secure reactive systems. In *Theory of Cryptography (LNCS)*, Moni Naor (Ed.), Vol. 2951. Springer, 336–354.

[6] Michael Backes, Birgit Pfitzmann, and Michael Waidner. 2007. The reactive simulatability (RSIM) framework for asynchronous systems. *Info. Comput.* 205, 12 (2007), 1685–1720.

[7] Boaz Barak. 2001. How to go beyond the black-box simulation barrier. In *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science (FOCS'01).* 106–115. DOI: https://doi.org/10.1109/SFCS.2001.959885

[8] Boaz Barak, Ran Canetti, Yehuda Lindell, Rafael Pass, and Tal Rabin. 2011. Secure computation without authentication. *J. Cryptol.* 24, 4 (2011), 720–760.

[9] Boaz Barak, Oded Goldreich, Shafi Goldwasser, and Yehuda Lindell. 2001. Resettably sound zero-knowledge and its applications. In *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science (FOCS'01).* 116–125. DOI: https://doi.org/10.1109/SFCS.2001.959886

[10] Boaz Barak, Yehuda Lindell, and Tal Rabin. 2004. Protocol initialization for the framework of universal composability. *IACR Cryptol. ePrint Arch.* 2004 (2004), 6. Retrieved from http://eprint.iacr.org/2004/006.

[11] Boaz Barak and Amit Sahai. 2005. How to play almost any mental game over the net—Concurrent composition via super-polynomial simulation. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05).* 543–552. DOI: https://doi.org/10.1109/SFCS.2005.43

[12] Donald Beaver. 1991. Secure multiparty protocols and zero-knowledge proof systems tolerating a faulty minority. *J. Cryptol.* 4, 2 (Jan. 1991), 75–122.

[13] Donald Beaver. 1996. Adaptive zero knowledge and computational equivocation (extended abstract). In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing.* 629–638. DOI: https://doi.org/10.1145/237814.238014

[14] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. [n.d.]. Relations among notions of security for public-key encryption schemes. 26–45. Retrieved from http://www.cs.ucsd.edu/users/mihir/papers/relations.html.

[15] Mihir Bellare and Phillip Rogaway. [n.d.]. Entity authentication and key distribution. In *Proceedings of the Annual Cryptology Conference (CRYPTO'93).* 232–249. Retrieved from http://www-cse.ucsd.edu/users/mihir/.

[16] Michael Ben-Or, Ran Canetti, and Oded Goldreich. 1993. Asynchronous secure computation. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing.* 52–61. DOI: https://doi.org/10.1145/167088.167109

[17] Michael Ben-Or and Dominic Mayers. 2004. General security definition and composability for quantum & classical protocols. arXiv preprint quant-ph/0409062, 2004 - arxiv.org.

[18] Eli Biham and Adi Shamir. 1997. Differential fault analysis of secret key cryptosystems. In *Proceedings of the Annual Cryptology Conference (CRYPTO'97).*

[19] Ray Bird, Inder S. Gopal, Amir Herzberg, Philippe A. Janson, Shay Kutten, Refik Molva, and Moti Yung. 1991. Systematic design of two-party authentication protocols. In *Proceedings of the Annual Cryptology Conference (CRYPTO'91).*

[20] Nir Bitansky, Ran Canetti, and Shai Halevi. 2012. Leakage-tolerant interactive protocols. *IACR Cryptol. ePrint Arch.* 2011 (2012), 204.

[21] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. 1997. On the importance of checking cryptographic protocols for faults (extended abstract). In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'97).*

[22] Gilles Brassard, David Chaum, and Claude Crépeau. 1988. Minimum disclosure proofs of knowledge. *J. Comput. Syst. Sci.* 37 (1988), 156–189.

[23] Ran Canetti. 2000. Security and composition of multi-party cryptographic protocols. *J. Cryptol.* 13, 1 (Jan. 2000), 143–202.

[24] Ran Canetti. 2000. Universally composable security: A new paradigm for cryptographic protocols. *IACR Cryptol. ePrint Arch.* 2000 (2000), 67.

[25] Ran Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science (FOCS'01).* 136–145.

[26] Ran Canetti. 2004. Universally composable signature, certification, and authentication. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop.* 219–233.

[27] Ran Canetti. 2006. Security and composition of cryptographic protocols: A tutorial. *SIGACT News* 37 (2006), 67–92.

[28] Ran Canetti. 2007. Obtaining universally compoable security: Toward the bare bones of trust. *IACR Cryptol. ePrint Arch.* 2007 (2007), 475.

[29] Ran Canetti. 2008. Composable formal security analysis: Juggling soundness, simplicity and efficiency. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP'08).*

[30] Ran Canetti. 2013. Security and composition of cryptographic protocols: A tutorial. In *Secure Multi-Party Computation (Cryptology and Information Security Series 10)*, Manoj Prabhakaran and Amit Sahai (Eds.). IOS Press, 61–119.

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

[32] Ran Canetti, Ling Cheung, Dilsun Kirli Kaynar, Moses D. Liskov, Nancy A. Lynch, Olivier Pereira, and Roberto Segala. 2018. Task-structured probabilistic I/O automata. *J. Comput. Syst. Sci.* 94 (2018), 63–97. DOI: https://doi.org/10.1016/j.jcss.2017.09.007

[33] Ran Canetti, Asaf Cohen, and Yehuda Lindell. 2015. A simpler variant of universally composable security for standard multiparty computation. In *Proceedings of the 35th Annual Cryptology Conference (CRYPTO'15).* 3–22.

[34] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. 2007. Universally composable security with global setup. In *Theory of Cryptography*, Salil P. Vadhan (Ed.). Springer, Berlin, 61–85.

[35] Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. 1996. Adaptively secure multi-party computation. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing.* 639–648.

[36] Ran Canetti and Marc Fischlin. 2001. Universally composable commitments. In *Proceedings of the 21st Annual Cryptology Conference (CRYPTO'01).* 19–40.

[37] Ran Canetti and Rosario Gennaro. 1996. Incoercible multiparty computation. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS'96).* 504–513.

[38] Ran Canetti, Shafi Goldwasser, and Oxana Poburinnaya. 2015. Adaptively secure two-party computation from indistinguishability obfuscation. In *Proceedings of the 12th Theory of Cryptography Conference (TCC'15).* 557–585.

[39] Ran Canetti, Shai Halevi, and Jonathan Katz. 2005. Adaptively secure, non-interactive public-key encryption. In *Proceedings of the 2nd Theory of Cryptography Conference (TCC'05).* 150–168.

[40] Ran Canetti and Jonathan Herzog. 2011. Universally composable symbolic security analysis. *J. Cryptol.* 24, 1 (2011), 83–147. DOI: https://doi.org/10.1007/s00145-009-9055-0

[41] Ran Canetti and Hugo Krawczyk. [n.d.]. Analysis of key-exchange protocols and their use for building secure channels. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt'01).* 453–474.

[42] Ran Canetti and Hugo Krawczyk. [n.d.]. Universally composable notions of key exchange and secure channels. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt'02).* 337–351.

[43] Ran Canetti, Hugo Krawczyk, and Jesper Buus Nielsen. 2003. Relaxing chosen-ciphertext security. In *Proceedings of the 23rd Annual International Cryptology Conference (CRYPTO'03).* 565–582.

[44] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. 2002. Universally composable two-party and multi-party secure computation. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC'02).* ACM, 494–503.

[45] Ran Canetti and Tal Rabin. 1993. Fast asynchronous Byzantine agreement with optimal resilience. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC'93)*, S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal (Eds.). ACM, 42–51. DOI: https://doi.org/10.1145/167088.167105

[46] Ran Canetti and Tal Rabin. 2003. Universal composition with joint state. In *Proceedings of the Annual Cryptology Conference (CRYPTO'03) (LNCS)*, Dan Boneh (Ed.), Vol. 2729. Springer, 265–281.

[47] Ran Canetti, Daniel Shahaf, and Margarita Vald. 2016. Universally composable authentication and key-exchange with global PKI. In *Proceedings of the Conference on Public-Key Cryptography (PKC'16)*, Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang (Eds.). Springer, Berlin, 265–296.

[48] Ran Canetti and Margarita Vald. 2012. Universally composable security with local adversaries. In *Proceedings of the Conference on Simulation of Computer Networks (SCN'12) (Lecture Notes in Computer Science)*, Vol. 7485. Springer, 281–301.

[49] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. 1999. Toward sound approaches to counteract power-analysis attacks. In *Proceedings of the Annual Cryptology Conference (CRYPTO'99).*

[50] Benny Chor and Lior Moscovici. 1989. Solvability in asynchronous environments (extended abstract). In *Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS'89).*

[51] Benny Chor and Lee-Bath Nelson. 1999. Solvability in asynchronous environments II: Finite interactive tasks. *SIAM J. Comput.* 29 (1999), 351–377.

[52] Giovanni Di Crescenzo, Yuval Ishai, and Rafail Ostrovsky. 1998. Non-interactive and non-malleable commitment. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC'98).* 141–150.

[53] Anupam Datta. 2005. *Security Analysis of Network Protocols: Compositional Reasoning and Complexity-theoretic Foundations.* Ph.D. Thesis, Computer Science Department, Stanford University.

[54] Yevgeniy Dodis and Silvio Micali. 2000. Parallel reducibility for information-theoretically secure computation. In *Proceedings of the Annual Cryptology Conference (CRYPTO'00).*

[55] Danny Dolev, Cynthia Dwork, and Moni Naor. 2000. Nonmalleable cryptography. *SIAM J. Comput.* 30, 2 (2000), 391–437.

[56] Cynthia Dwork, Moni Naor, and Amit Sahai. 2004. Concurrent zero-knowledge. *J. ACM* 51, 6 (2004), 851–898.

[57] Shimon Even, Oded Goldreich, and Abraham Lempel. 1985. A randomized protocol for signing contracts. *Commun. ACM* 28, 6 (1985), 637–647.

[58] Marc Fischlin and Roger Fischlin. 2000. Efficient non-malleable commitment schemes. In *Proceedings of the 35th Annual Cryptology Conference (CRYPTO'00) (Lecture Notes in Computer Science)*, Vol. 1880. Springer, 413–431.

[59] Juan Garay and Philip MacKenzie. 2000. Concurrent oblivious transfer. In *Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS'00).* IEEE Computer Society, 314–324.

[60] Rosario Gennaro, Anna Lysyanskaya, Tal Malkin, Silvio Micali, and Tal Rabin. 2004. Algorithmic tamper-proof (ATP) security: Theoretical foundations for security against hardware tampering. In *Proceedings of the Theory of Cryptography Conference (TCC'04) (Lecture Notes in Computer Science)*, Vol. 2951. Springer, 258–277.

[61] Oded Goldreich. 2001. *Foundations of Cryptography—Basic Tools.* Cambridge University Press.

[62] Oded Goldreich. 2002. Concurrent zero-knowledge with timing, revisited. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC'02).* ACM, 332–340.

[63] Oded Goldreich. 2004. *The Foundations of Cryptography—Volume 2: Basic Applications.* Cambridge University Press.

[64] Oded Goldreich and Ariel Kahan. 1996. How to construct constant-round zero-knowledge proof systems for NP. *J. Cryptol.* 9, 3 (1996), 167–190.

[65] Oded Goldreich and Hugo Krawczyk. 1996. On the composition of zero-knowledge proof systems. *SIAM J. Comput.* 25, 1 (1996), 169–192.

[66] Oded Goldreich and Yehuda Lindell. 2006. Session-key generation using human passwords only. *J. Cryptol.* 19, 3 (2006), 241–340.

[67] Oded Goldreich, Silvio Micali, and Avi Wigderson. 2019. How to play any mental game, or a completeness theorem for protocols with honest majority. In *Providing Sound Foundations for Cryptography.* ACM, 307–328.

[68] Oded Goldreich and Yair Oren. 1994. Definitions and properties of zero-knowledge proof systems. *J. Cryptol.* 7, 1 (1994), 1–32.

[69] Shafi Goldwasser and Leonid A. Levin. 1990. Fair computation of general functions in presence of immoral majority. In *Proceedings of the Annual Cryptology Conference* (CRYPTO'90) (Lecture Notes in Computer Science), Vol. 537. Springer, 77–93.

[70] Shafi Goldwasser and Silvio Micali. 1984. Probabilistic encryption. *J. Comput. Syst. Sci.* 28, 2 (1984), 270–299.

[71] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1989. The knowledge complexity of interactive proof systems. *SIAM J. Comput.* 18, 1 (1989), 186–208.

[72] Jens Groth, Rafail Ostrovsky, and Amit Sahai. 2012. New techniques for noninteractive zero-knowledge. *J. ACM* 59, 3 (2012), 11:1–11:35.

[73] Martin Hirt and Ueli M. Maurer. 1997. Complete characterization of adversaries tolerable in secure multi-party computation (extended abstract). In *Proceedings of the Symposium on Principles of Distributed Computing (PODC'97).* ACM, 25–34.

[74] C. A. R. Hoare. 1985. *Communicating Sequential Processes.* Prentice-Hall.

[75] Dennis Hofheinz and Jörn Müller-Quade. 2004. A synchronous model for multi-party computation and the incompleteness of oblivious transfer. *IACR Cryptol. ePrint Arch.* 2004 (2004), 16.

[76] Dennis Hofheinz, Jörn Müller-Quade, and Rainer Steinwandt. 2003. Initiator-resilient universally composable key exchange. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS'03) (Lecture Notes in Computer Science)*, Vol. 2808. Springer, 61–84.

[77] Dennis Hofheinz and Victor Shoup. 2015. GNUC: A new universal composability framework. *J. Cryptol.* 28, 3 (2015), 423–508.

[78] Dennis Hofheinz and Dominique Unruh. 2005. Comparing two notions of simulatability. In *Proceedings of the Theory of Cryptography Conference (TCC'05) (Lecture Notes in Computer Science)*, Vol. 3378. Springer, 86–103.

[79] Dennis Hofheinz, Dominique Unruh, and Jörn Müller-Quade. 2013. Polynomial runtime and composability. *J. Cryptol.* 26, 3 (2013), 375–441.

[80] Gilles Kahn. 1974. The semantics of a simple language for parallel programming. In *Proceedings of the International Federation for Information Processing Congress (IFIP'74)*. North-Holland, 471–475.

[81] Yael Tauman Kalai, Yehuda Lindell, and Manoj Prabhakaran. 2005. Concurrent general composition of secure protocols in the timing model. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC'05)*.

[82] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. 2013. Universally composable synchronous computation. In *Proceedings of the Theory of Cryptography Conference (TCC'13) (Lecture Notes in Computer Science)*, Vol. 7785. Springer, 477–498.

[83] Paul C. Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of the Annual Cryptology Conference (CRYPTO'96)*.

[84] Ralf Küsters. 2006. Simulation-based security with inexhaustible interactive Turing machines. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW'06)* (2006), 12 pp.–320.

[85] Ralf Küsters, Anupam Datta, John C. Mitchell, and Ajith Ramanathan. 2008. On the relationships between notions of simulation-based security. *J. Cryptol.* 21 (2008), 492–546.

[86] Ralf Küsters, Max Tuengerthal, and Daniel Rausch. 2020. The IITM model: A simple and expressive model for universal composability. *J. Cryptol.* (2020). DOI: https://doi.org/10.1007/s00145-020-09352-1

[87] Patrick Lincoln, John C. Mitchell, Mark Mitchell, and Andre Scedrov. 1998. A probabilistic poly-time framework for protocol analysis. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'98)*.

[88] Patrick Lincoln, John C. Mitchell, Mark Mitchell, and Andre Scedrov. 1999. Probabilistic polynomial-time equivalence and security analysis. In *Proceedings of the World Congress on Formal Methods*.

[89] Yehuda Lindell. 2003. General composition and universal composability in secure multi-party computation. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*. 394–403.

[90] Yehuda Lindell, Anna Lysyanskaya, and Tal Rabin. 2002. On the composition of authenticated byzantine agreement. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC'02)*.

[91] Nancy A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann.

[92] Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. 2003. Compositionality for probabilistic automata. In *Proceedings of the International Conference on Concurrency Theory (CONCUR'03)*.

[93] Paulo Mateus, John C. Mitchell, and Andre Scedrov. 2003. Composition of cryptographic protocols in a probabilistic polynomial-time process calculus. In *Proceedings of the International Conference on Concurrency Theory (CONCUR'03)*.

[94] Ueli Maurer and Renato Renner. 2011. Abstract cryptography. In *Proceedings of the Innovations in Computer Science (ICS'11)*, Bernard Chazelle (Ed.). Tsinghua University Press, 1–21.

[95] Silvio Micali and Leonid Reyzin. 2004. Physically observable cryptography. In *Proceedings of the Theory of Cryptography Conference (TCC'04)*.

[96] Silvio Micali and Phillip Rogaway. 1991. Secure computation. In *Unpublished Manuscript. Preliminary version in Advances in Cryptology—CRYPTO (LNCS)*, Joan Feigenbaum (Ed.), Vol. 576. Springer, 392–404.

[97] Daniele Micciancio and Stefano Tessaro. 2013. An equational approach to secure multi-party computation. In *Proceedings of the Conference on Innovations in Theoretical Computer Science (ITCS'13)*. ACM, 355–372.

[98] Daniele Micciancio and Bogdan Warinschi. 2004. Soundness of formal encryption in the presence of active adversaries. In *Proceedings of the Theory of Cryptography Conference (TCC'04) (Lecture Notes in Computer Science)*. Springer, 133–151.

[99] Robin Milner. 1989. *Communication and Concurrency*. Prentice Hall.

[100] Robin Milner. 1999. *Communicating and Mobile Systems—The Pi-calculus*. Cambridge University Press.

[101] John C. Mitchell, Mark Mitchell, and Andre Scedrov. 1998. A linguistic characterization of bounded oracle computation and probabilistic polynomial time. In *Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS'98)*. IEEE Computer Society, 725–733.

[102] Moni Naor and Moti Yung. 1990. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC'90)*. ACM, 427–437.

[103] Jesper Buus Nielsen. 2002. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In *Proceedings of the Annual Cryptology Conference (CRYPTO'02) (Lecture Notes in Computer Science)*, Vol. 2442. Springer, 111–126.

[104] Jesper Buus Nielsen. 2003. On protocol security in the cryptographic model. Ph.D. Thesis, Arhus University.

[105] Rafael Pass. 2004. Bounded-concurrent secure multi-party computation with a dishonest majority. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC'04)*.

[106] Rafael Pass. 2006. A precise computational approach to knowledge. Ph.D. Thesis, MIT.

[107] Birgit Pfitzmann, Matthias Schunter, and Michael Waidner. 2000. Cryptographic security of reactive systems. *Electron. Notes Theor. Comput. Sci.* 32 (2000), 59–77. DOI: https://doi.org/10.1016/S1571-0661(04)00095-7

[108] Birgit Pfitzmann and Michael Waidner. 2000. Composition and integrity preservation of secure reactive systems. In *Proceedings of the ACM Conference on Computer and Communications Security*.

[109] Birgit Pfitzmann and Michael Waidner. 2001. A model for asynchronous reactive systems and its application to secure message transmission. *Proceedings of the IEEE Symposium on Security and Privacy (S&P'01)*. 184–200.

[110] Birgit Pfitzmann and Michael Waidner. 1994. Hildesheimer Informatik-Berichte 11/94, Universitat Hildesheim. Retrieved from http://www.semper.org/sirene/lit.

[111] Manoj Prabhakaran and Amit Sahai. 2004. New notions of security: Achieving universal composability without trusted setup. *IACR Cryptol. ePrint Arch.* 2004 (2004), 139.

[112] Michael O. Rabin. 1981. How to exchange secrets with oblivious transfer. *Technical report TR-81, Aiken Computation Laboratory, Harvard University* (1981).

[113] Charles Rackoff and Daniel R. Simon. 1991. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Proceedings of the Annual Cryptology Conference (CRYPTO'91)*.

[114] Ransom Richardson and Joe Kilian. 1999. On the concurrent composition of zero-knowledge proofs. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'99)*.

[115] Roberto Segala and Nancy A. Lynch. 1994. Probabilistic simulations for probabilistic processes. *Nord. J. Comput.* 2 (1994), 250–273.

[116] Victor Shoup. 1999. On formal models for secure key exchange. *IACR Cryptol. ePrint Arch.* 1999 (1999), 12.

[117] Michael Sipser. 2013. *Introduction to the Theory of Computation*, 3rd ed. Cengage Learning Publishing Company.

[118] Douglas Wikström. 2016. Simplified universal composability framework. In *Proceedings of the Theory of Cryptography Conference (TCC'16)*.

[119] Douglas Wikström. 2005. On the security of mix-nets and hierarchical group signatures. Ph.D. Thesis, KTH.

[120] Andrew Chi-Chih Yao. 1982. Protocols for secure computations. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science (FOCS'82)*.

[121] Andrew Chi-Chih Yao. 1982. Theory and applications of trapdoor functions (extended abstract). In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science (FOCS'82)*. 80–91.

Received December 2005; accepted May 2020