GNUC: A New Universal Composability Framework

Dennis Hofheinz* Victor Shoup†

June 6, 2011

Abstract

We put forward a framework for the modular design and analysis of multi-party protocols. Our framework is called "GNUC" (with the recursive meaning "GNUC's Not UC"), already alluding to the similarity to Canetti's Universal Composability (UC) framework. In particular, like UC, we offer a universal composition theorem, as well as a theorem for composing protocols with joint state.

We deviate from UC in several important aspects. Specifically, we have a rather different view than UC on the structuring of protocols, on the notion of polynomial-time protocols and attacks, and on corruptions. We will motivate our definitional choices by explaining why the definitions in the UC framework are problematic, and how we overcome these problems.

Our goal is to make offer a framework that is largely compatible with UC, such that previous results formulated in UC carry over to GNUC with minimal changes. We exemplify this by giving explicit formulations for several important protocol tasks, including authenticated and secure communication, as well as commitment and secure function evaluation.

1 Introduction

Modular protocol design. The design and analysis of complex, secure multi-party protocols requires a high degree of modularity. By modularity, we mean that protocol components (i.e., subprotocols) can be analyzed separately; once all components are shown secure, the whole protocol should be.

Unfortunately, such a secure composition of components is not a given. For example, while one instance of textbook RSA encryption with exponent $e = 3$ may be secure in a weak sense, all security is lost if three participants encrypt the same message (under different moduli), see [Hås88]. Furthermore, zero-knowledge proof systems may lose their security when executed concurrently [GK96]. In both cases, multiple instances of the same subprotocol may interact in strange and unexpected ways.

The UC framework. However, if security of each component *in arbitrary contexts* is proven, then, by definition, secure composition and hence modular design is possible. A suitable security notion (dubbed "UC security") was put forward by Canetti [Can01], building on a series of earlier works [GMW86, Bea92, MR92, Can00].

Like earlier works, UC defines security through emulation; that is, a (sub)protocol is considered secure if it emulates an ideal abstraction of the respective protocol task. In this, one system Π emulates another system $\mathcal{F}$ if both systems look indistinguishable *in arbitrary environments*, and

*Karlsruhe Institute of Technology

†New York University; supported by NSF grant CNS-0716690

even in face of attacks. In particular, for every attack on Π, there should be a "simulated attack" on $\mathcal{F}$ that achieves the same results.

Universal composition. Unlike earlier works, UC features a *universal composition theorem* (hence the name UC): if a protocol is secure, then even many instances of this protocol do not lose their security in arbitrary contexts. Technically, if Π emulates an ideal functionality $\mathcal{F}$, then we can replace all $\mathcal{F}$-instances with Π-instances in arbitrary larger protocols that use $\mathcal{F}$ as a subprotocol.

This UC theorem has proven to be a key ingredient to modular analysis. Since the introduction of the UC framework in 2001, UC-compatible security definitions for most conceivable cryptographic tasks have been given (see, e.g., [Can05] for an overview). This way, highly nontrivial existing (e.g., [CLOS02, GMP+08]) and new (e.g., [Bar05, BCD+09]) multi-party protocols could be explained and analyzed in a structured fashion. In fact, a security proof in the UC framework has become a standard litmus test for complex multi-party computations. For instance, by proving that a protocol emulates a suitable ideal functionality, it usually becomes clear what exactly is achieved, and against which kinds of attacks.

The current UC framework. The technical formulation of the UC framework has changed over time, both correcting smaller technical bugs, and extending functionality. As an example of a technical bug, the notion of polynomial runtime in UC has changed several times, because it was found that earlier versions were not in fact compatible with common constructions or security proofs (see [HUMQ09] for an overview). As an example of an extension, the model of computation (and in particular message scheduling and corruption aspects) in UC has considerably evolved. For instance, earlier UC versions only allowed the atomic corruption of protocol parties; the most recent public UC version [Can05] allows a very fine-grained corruption of subprotocol parts, while leaving higher-layer protocols uncorrupted.

Issues in the UC framework. In the most recent public version UC05 [Can05], the UC framework has evolved to a rather complex set of rules, many of which have grown historically (e.g., the control function, a means to enforce global communication rules). As we will argue below, this has led to a number of undesirable effects.

Composition theorem. As a first example, we claim that, strictly speaking, the UC theorem itself does not hold in UC05. The reason has to do with the formalization of the composition operation, i.e., the process of replacing one subprotocol with another. In UC05, the composition operation replaces the code of an executed program party-internally, without changing even the interface to the adversary. Hence, an adversary may not even know which protocol a party is running.

However, during the proof of the composition theorem, we have to change exactly those parts of the adversary that relate to the replaced subprotocol instances. Because there is no way for the adversary to tell which program a party runs, it is hence not clear how to change the adversary.

We give a more detailed description of in §11.2, along with a counterexample, which we briefly sketch here (using traditional UC terminology).

We start with a one-party protocol Π' that works as follows. It expects an initialization message from the environment Z, which it forwards to the adversary A. After this, it awaits a bit b from A, which it forwards to Z.

We next define a protocol Π'_1, which works exactly the same as Π', except that upon receipt of the bit b from A, it sends $1 - b$ to Z.

We claim that in any reasonable UC framework, Π'_1 emulates Π'. The simulator A', which is attacking Π', uses an internal copy of an adversary A'_1 that is supposed to be attacking Π'_1. When A'_1 attempts to send a bit b to Π'_1, A' instead sends the bit $1 - b$ to Π'.

So now consider a one-party protocol Π that works as follows. Π expects an initial message from Z, specifying a bit c; if $c = 0$, it initializes a subroutine running Π', and if $c = 1$, it initializes a subroutine running Π'_1. However, the machine ID assigned to the subroutine is the same in either case. When Π receives a bit from its subroutine, it forwards that bit to Z.

The composition theorem says that we should be able to substitute Π'_1 for Π' in Π, obtaining a protocol Π_1 that emulates Π. Note that in Π_1, the subroutine called is Π'_1, regardless of the value of c. However, it is impossible to build such a simulator — intuitively, the simulator would have to decide whether to invert the bit, as in A', or not, and the simulator simply does not have enough information to do this correctly.

In any fix to this problem, the adversary essentially must be able to determine not only the program being run by a given machine, but also the entire protocol stack associated with it, in order to determine whether it belongs to the relevant subprotocol or not.

Trust hierarchy. Next, recall that UC05 allows very fine-grained corruptions. In particular, it is possible for an adversary to corrupt only a subroutine (say, for secure communication with another party) of a given party. In this, each corruption must be explicitly "greenlighted" by the protocol environment, to ensure that the set of corrupted parties does not change during composition. Specifically, this explicit authorization step prevents a trivial simulation in which the ideal adversary corrupts all ideal parties.

We claim that the UC05 corruption mechanism is problematic for two reasons. First, usually the set of (sub)machines in a real protocol and in an ideal abstraction differ. As an example, think of a protocol that implements a secure auction ideal functionality, and in the process uses a subprotocol for secure channels. The real protocol is comprised of at least two machines per party (one for the main protocol, one for the secure channels subprotocol). However, an ideal functionality usually has only one machine per party. Now consider a real adversary that corrupts only the machine that corresponds to a party's secure channels subroutine. Should the ideal adversary be allowed to corrupt the only ideal machine for this party? How should this be handled generally? In the current formulation of UC05, this is simply unclear. (We give more details and discussion about this in §11.3.)

Second, consider the secure auctions protocol again. In this example, an adversary can impersonate a party by *only* corrupting this party's secure channels subroutine. (All communication is then handled by the adversary in the name of the party.) Hence, for all intents and purposes, such a party should be treated as corrupted, although it formally is not. This can be pushed further: in UC05, the adversary can actually bring arbitrary machines (i.e., subroutines) into existence by sending them a message. There are no restrictions on the identity or program of such machines; only if a machine with that particular identity already exists is the message relayed to that machine. In particular, an adversary could create a machine that communicates with other parties in the name of an honest party before that party creates its own protocol stack and all its subroutines. This has the same effect as corrupting the whole party, but *without actually corrupting any machine.* (See §11.3 for more details.)

Polynomial runtime. Loosely speaking, UC05 considers a protocol poly-time when all machines run a number of steps that is polynomial in the difference between the length of their respective inputs minus the length of all inputs passed down to subroutines. This implies that the abstract interface of a protocol must contain enough input padding to propagate runtime through all necessary subprotocols of an implementation. In particular, this means that formally, the interface of an ideal protocol must depend on the complexity of the intended implementation. This

complicates the design of larger protocols, which, e.g., must adapt their own padding requirements to those of their subprotocols. Similar padding issues arise in the definition of poly-time adversaries. This situation is somewhat unsatisfying from an aesthetic point of view, in particular since such padding has no intuitive justification. We point out further objections against the UC05 notion of polynomial runtime in §11.7.

None of the objections we raise point to gaps in security proofs of existing protocols. Rather, they seem artifacts of the concrete technical formulation of the underlying framework.

GNUC. One could try to address all those issues directly in the UC framework. However, this would likely lead to an even more complex framework; furthermore, since significant changes to the underlying communication and machine model seem necessary, extreme care must be taken that all parts of the UC specification remain consistent. For these reasons, we have chosen to develop GNUC (meaning "GNUC's not UC", and pronounced g-NEW-see) from scratch as an alternative to the UC framework. In GNUC, we explicitly address all of the issues raised with UC, while trying to remain as compatible as possible with UC terminology and results.

Before going into details, let us point out our key design goals:

Compatibility with UC. Since the UC issues we have pointed out have nothing to do with concrete security proofs for protocols, we would like to make it very easy to formulate existing UC results in GNUC. In particular, our terminology is very similar to the UC terminology (although the technical underpinnings differ). Also, we establish a number of important UC results (namely, the UC theorem, completeness of the dummy adversary, and composition with joint state) for GNUC. We also give some example formulations of common protocol tasks. Anyone comfortable in using UC should also feel at home with GNUC.

Protocol hierarchy. We establish a strict hierarchy of protocols. Every machine has an identity that uniquely identifies its position in the tree of possible (sub)protocols. Concretely, any machine M' invoked by another machine M has an identity that is a direct descendant of that of M. The program of a machine is determined by its identity, via a *program map*, which maps the machine's identity to a program in a *library* of programs. Replacing a subprotocol then simply means changing the library accordingly. This makes composition very easy to formally analyze and implement. Furthermore, the protocol hierarchy allows to establish a reasonable corruption policy (see below).

Hierarchical corruptions. Motivated by the auction example above, we formulate a basic premise:

if any subroutine of a machine M is corrupt, then M itself should be viewed as corrupt.

In particular, if a subroutine is corrupted that manages all incoming and outgoing communication, then the whole party must be viewed as corrupt. This translates to the following corruption policy: in order to corrupt a machine, an adversary must corrupt all machines that are above that machine in the protocol hierarchy. Since our adversary cannot spontaneously create machines (as in UC), this completely avoids the "hijacking" issues in UC explained above. Furthermore, when we compare a real and an ideal protocol, real corruptions always have ideal counterparts. (If the real adversary corrupts a subroutine, it must have corrupted the corresponding root machine in the hierarchy; at least this root machine must have an ideal counterpart, which must now be corrupted as well.)

Polynomial runtime. Our definition of polynomial runtime should avoid the technical pitfalls that led to current UC runtime definition. At the same time, it should be convenient to use, without unnatural padding conventions (except, perhaps, in extreme cases). A detailed description can be

found in our technical roadmap in §2. However, at this point we would like to highlight a nice property that our runtime notion shares with that of UC05. Namely, our poly-time notion is closed under composition, in the following sense: if one instance of a protocol is poly-time, then many instances are as well. Furthermore, if we replace a poly-time subprotocol Π' of a poly-time protocol Π with a poly-time implementation Π'_1 of Π', then the resulting protocol is poly-time as well. (While this sounds natural, this is not generally the case for other notions of poly-time such as the one from [HUMQ09].)

Other related work. Several issues in the UC framework have been addressed before, sometimes in different protocol frameworks. In particular, the issues with the UC poly-time notion were already recognized in [HMQU05, Küs06, HUMQ09]. These works also propose solutions (in different protocol frameworks); we comment on the technical differences in §11.7, §11.8, and §11.9. The UC issues related to corruptions have already been recognized in [CCGS10].

Besides, there are other protocol frameworks; these include Reactive Simulatability [BPW07], the IITM framework [Küs06], and the recent Abstract Cryptography [MR11] framework. We comment on Reactive Simulatability and the IITM framework in §11.9 and §11.8. The Abstract Cryptography framework, however, focuses on abstract concepts behind cryptography and has not yet been fully specified on a concrete machine level.

2 Roadmap

In this section, we give a high-level description of each of the remaining sections.

Section 3: machine models

In this section, we present a simple, low-level model of *interactive machines (IMs)* and *systems of IMs*. Basically, a system of IMs is a network of machines, with communication based on *message passing*. Execution of such a system proceeds as a series of *activations*: a machine receives a message, processes it, updates its state, and sends a message to another machine.

Our model allows an unbounded number of machines. Machines are addressed by *machine IDs*. If a machine sends a message to a machine which does not yet exist, the latter is dynamically generated. A somewhat unique feature of our definition of a system of IMs is the mechanism by which the program of a newly created IM is determined. Basically, a system of IMs defines a *library of programs*, which is a finite map from *program names* to *programs* (i.e., *code*). In addition, the system defines a mapping from machine IDs to program names. Thus, for a fixed system of IMs, a machine's ID determines its program.

Section 4: structured systems of interactive machines

Our definition of a system of IMs is extremely general — too general, in fact, to effectively model the types of systems we wish to study in the context of universal composition of protocols. Indeed, the definitions in §3 simply provide a convenient layer of abstraction. In §4, we define in great detail the notion of a *structured* system of IMs, which is a special type of system of IMs with restrictions placed on machine IDs and program behaviors. We give here a brief outline of what these restrictions are meant to provide.

In a structured system of IMs, there are three classes of machines: *environment*, *adversary*, and *protocol*. There will only be one instance of an environment, and only one instance of an adversary, but there may be many instances of protocol machines, running a variety of different programs.

Protocol machines have IDs of the form $\langle\, pid, sid \,\rangle$. Here, *pid* is called a *party ID (PID)* and *sid* is called a *session ID (SID)*. Machines with the same SID run the same program, and are considered *peers*. The PID serves to distinguish these peers. Unfortunately, the term "party" carries a number of connotations, which we encourage the reader to ignore completely. The only meaning that should be applied to the term "party" is that implied by the rules regarding PIDs. We will go over these rules shortly.

We classify protocol machines as either *regular* or *ideal*. The only thing that distinguishes these two types syntactically is their PID: ideal machines have a special PID, which is distinct from the PIDs of all regular machines. Regular and real machines differ, essentially, in the communication patterns that they are allowed to follow. An ideal machine may communicate directly (with perfect secrecy and authentication) with any of its regular peers, as well as with the adversary.

A regular machine may interact with its ideal peer, and with the adversary; it may not interact directly with any of its other regular peers, although indirect interaction is possible via the ideal peer. A regular machine may also pass messages to *subroutines*, which are also regular machines. Subroutines of a machine M may also send messages to M, their *caller*.

Two critical features of our framework are that regular machines are only created by being called, as above, and that every regular machine has a unique caller. Usually, the caller of a regular machine M will be another regular machine; however, it may also be the environment, in which case, M is a "top level" machine. The environment may only communicate directly with such top-level regular machines, as well as with the adversary.

Another feature of our framework is that the SID of a regular machine specifies the name of the program run by that machine, and moreover, the SID completely describes the sequence of subroutine calls that gave rise to that machine. More specifically, an SID is structured as a "pathname", and when a machine with a given pathname calls a subroutine, the subroutine shares the same PID as the caller, and the subroutine's SID is a pathname that extends the pathname of the caller by one component, and this last component specifies (among other things) the program name of the subroutine.

One final important feature of our framework is that the programs of regular machines must "declare" the names of the programs that they are allowed to call as subroutines. These declarations are strictly enforced at run time.

Execution of such a system proceeds as follows. First, the environment is activated. After this, as indicated by the restrictions described above, control may pass between

- the environment and a top-level regular machine,
- a regular machine and its ideal peer,
- a regular machine and its caller,
- a regular machine and one of its subroutines,
- the adversary and the environment, or
- the adversary and protocol machine (regular or ideal).

We close this section with the definition of the *dummy adversary*. This is a specific adversary that essentially acts as a "wire" connecting the environment to protocol machines.

Section 5: protocols

This section defines what we mean by a *protocol*. Basically, a protocol Π is a structured system of IMs, minus the environment and adversary. In particular, Π defines a map from program names to programs. The subroutine declarations mentioned above define a *static call graph*, with program names as nodes, and with edges indicating that one program may call another. The requirement is that this graph must be acyclic with a unique node r of in-degree 0. We say that r is the *root of* Π, or, alternatively, that Π *is rooted at* r.

We then define what it means for one protocol to be a *subprotocol* of another. Basically, Π' is a subprotocol of Π if the map from program names to programs defined by Π' is a restriction of the map defined by Π.

We also define a *subprotocol substitution operator*. If Π' is a subprotocol of Π, and Π'_1 is another protocol, we define $\Pi_1 := \Pi[\Pi'/\Pi'_1]$ to be the protocol obtained by replacing Π' with Π'_1 in Π. There are some technical restrictions, namely, that Π' and Π'_1 have the same root, and that the substitution itself does not result in a situation where one program name has two different definitions.

Observe that protocol substitution is a *static* operation performed on *libraries*, rather than a run-time operation.

We also introduce some other terminology.

If Z is an environment that only calls regular protocol machines running programs named r, then we say that Z is *multi-rooted at* r. In general, we allow such a Z to call machines with various SIDs, but if Z only calls machines with a single, common SID, then we say that Z is *rooted at* r.

If Π is a protocol rooted at r, A is an arbitrary adversary, Z is an environment multi-rooted at r, then these programs define a structured system of IMs, denoted by $[\Pi, A, Z]$.

If Z is rooted at r, then during an execution of $[\Pi, A, Z]$, a single *instance* of Π will be running; if Z is multi-rooted at r, then many instances of Π may be running. During the execution of such a system, it is helpful to visualize its *dynamic call graph*. The reader may wish to look at Fig. 2 on p. 24, which represents a dynamic call graph corresponding to two instances of a two-party protocol. In this figure, circles represent regular protocol machines. Rectangles represent ideal protocol machines, which are connected to their regular peers via a dotted line. In general, a dynamic call graph of regular machines will be a forest of trees, where the roots are the top-level machines that communicate directly with the environment.

Section 6: resource bounds

The main goal of this section is to define the notion of a polynomial time protocol.

When we define a polynomial time *algorithm*, we bound its running time as a function of the length of the input. For protocols that are reacting to multiple messages coming from an environment (either directly or via the dummy adversary), we shall bound the running time of *all* the machines comprising the protocols as a function of the *total* length of all these messages.

To this end, we only consider environments Z that are *well-behaved* in a technical sense defined in this section. Intuitively, and environment Z is well behaved if it runs in time polynomial in the length of all of its incoming messages, and the total length of messages emanating from it is bounded by a polynomial in security parameter.

The execution of a system $[\Pi, A, Z]$ will be driven by such a well-behaved environment Z. The running time of the protocol will be bounded by the length of various *flows*:

- f_{ep} is the flow from Z into Π;
- f_{ea} is the flow from Z into A;

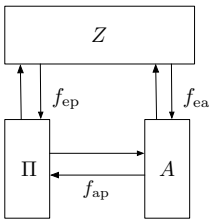


Figure 1: Flows

- f_{ap} is the flow from A into Π.

By flow, we mean the sum of the lengths of all relevant messages (including machine IDs and security parameter). To define the notion of a poly-time protocol, we do not need the flow f_{ap}, but this will come in handy later. See Fig. 1 for an illustration. We stress that in this figure, the box labeled Π represents all the running protocol machines.

Let us also define t_{p} to be the total running time of all protocol machines running in the execution of $[\Pi, A, Z]$, and t_{a} to be the total running time of A in this execution.

Our definition of a poly-time protocol runs like this: we say a protocol Π rooted at r is *(multi-)poly-time* if there exists a polynomial p such that for every well-behaved Z (multi-)rooted at r, in the execution of $[\Pi, A_{\text{d}}, Z]$, we have

$$t_{\text{p}} \leq p(f_{\text{ep}} + f_{\text{ea}})$$

with overwhelming probability. Recall that A_{d} denotes the dummy adversary. Since A_{d} is essentially a "wire", observe that f_{ap} is closely related to f_{ea}.

Two points in this definition deserve to be stressed: (i) the polynomial p depends on Π, but not on Z; (ii) the bound is required to hold only with overwhelming probability, rather than probability 1. We also stress that in bounding the running time of Π, we are bounding the total running time of all machines in Π in terms of the total flow out of Z — nothing is said about the running time of an individual machine in terms of the flow into that machine.

This definition is really two definitions in one: *poly-time* (for "singly" rooted environments) and *multi-poly-time* (for "multiply" rooted environments). The main theorem in this section, however, states that *poly-time implies multi-poly-time*.

Section 7: emulation

At the heart of any framework for universal composition is a notion of *emulation*. Very informally, one says that a protocol Π_1 emulates another protocol Π if for every attack on Π_1, there is a simulator that attacks Π, so that these two attacks are indistinguishable — the idea is that in this way, Π_1 acts as a "secure implementation" of Π. More formally, the definition says that for every adversary A_1, there exists an adversary A, such that $\text{Exec}[\Pi, A, Z] \approx \text{Exec}[\Pi_1, A_1, Z]$ for all environments Z. Here, $\text{Exec}[\Pi, A, Z]$ represents the output (which is generated by Z) of the execution of the system $[\Pi, A, Z]$, and "$\approx$" means "computationally indistinguishable". Also, we quantify over all well-behaved environments Z, rooted at the common root of Π and Π_1; if we allow multi-rooted environments, we say that Π_1 *multi-emulates* Π, and otherwise, we say that Π_1 *emulates* Π.

In the above definition, we still have to specify the types of adversaries, A and A_1, over which we quantify. Indeed, we have to restrict ourselves to adversaries that are appropriately resource bounded.

Recall the above definitions of flow and running time, namely, f_{ep}, f_{ea}, f_{ea}, t_{p}, and t_{a}. Suppose Π is a protocol rooted at r. We say that an adversary A is *(multi-)time-bounded for* Π if there exists a polynomial p, such that every well-behaved environment Z (multi-)rooted at r, in the execution of $[\Pi, A, Z]$, we have

$$t_{\mathrm{p}} + t_{\mathrm{a}} \leq p(f_{\mathrm{ep}} + f_{\mathrm{ea}})$$

with overwhelming probability. We also say that A is *(multi-)flow-bounded for* Π if there exists a polynomial p, such that every well-behaved environment Z (multi-)rooted at r, in the execution of $[\Pi, A, Z]$, we have

$$f_{\mathrm{ap}} \leq p(f_{\mathrm{ea}})$$

with overwhelming probability.

So in our definition of emulation, we restrict ourselves to adversaries that are both (multi-)time-bounded and (multi-)flow-bounded — we call such adversaries *(multi-)bounded*. The time-boundedness constraint should seem quite obvious and natural. However, the flow-boundedness constraint may seem somewhat non-obvious and unnatural; we shall momentarily discuss why it is needed, some difficulties it presents, and how these difficulties may be overcome.

It is easy to see that if Π is (multi-)poly-time, then the dummy adversary is (multi-)bounded for Π. So we always have at least one adversary to work with that satisfies our constraints — namely, the dummy adversary — and as we shall see, this is enough.

We state here the main theorems of this section.

Theorem 5 (completeness of the dummy adversary) Let Π and Π_1 be (multi-)poly-time protocols rooted at r. Suppose that there exists an adversary A that is (multi-)bounded for Π, such that for every well-behaved environment Z (multi-)rooted at r, we have $\mathrm{Exec}[\Pi, A, Z] \approx \mathrm{Exec}[\Pi_1, A_{\mathrm{d}}, Z]$. Then Π_1 (multi-)emulates Π.

Theorem 6 (emulates $\implies$ multi-emulates) Let Π and Π_1 be poly-time protocols. If Π_1 emulates Π, then Π_1 multi-emulates Π.

Recall that if a protocol is poly-time, then it is also multi-poly-time, so the statement of Theorem 6 makes sense. Because of these properties, we ignore multi-emulation in the remaining two theorems.

Theorem 7 (composition theorem) Suppose Π is a poly-time protocol rooted at r. Suppose Π' is a poly-time subprotocol of Π rooted at x. Finally, suppose Π'_1 is a poly-time protocol also rooted at x that emulates Π' and that is substitutable for Π' in Π. Then $\Pi_1 := \Pi[\Pi'/\Pi'_1]$ is poly-time and emulates Π.

Theorem 8 (reflexivity and transitivity of emulation) Let Π, Π_1, and Π_2 be poly-time protocols. Then Π emulates Π. In addition, if Π_2 emulates Π_1 and Π_1 emulates Π, then Π_2 emulates Π.

The composition theorem (Theorem 7) is arguably the "centerpiece" of any universal composition framework. It says that if Π'_1 emulates Π', then we can effectively substitute Π'_1 for Π' in any protocol Π that uses Π' as a subprotocol, without affecting the security of Π.

It is in the proof of the composition theorem that we make essential use of the flow-boundedness constraint. This constraint allows us to conclude from the hypotheses that the instantiated protocol Π_1 is itself poly-time. In order to use the composition theorem in a completely modular way, this is essential — typically, the protocol Π will be designed without regard to the implementation of Π', and the protocol Π'_1 is designed without regard to the usage of Π' in Π.

If we drop the flow-boundedness constraint, the composition theorem is no longer true, and in particular, Π_1 may not be poly-time. Perhaps there is another type of constraint that would allow us to prove the composition theorem, but our investigations so far have not yielded any viable alternatives.

The flow-boundedness constraint can, at times, be difficult to deal with. Indeed, it turns out that in some situations, it is difficult to design a simulator that satisfies it. To mitigate these difficulties, we found it necessary to use a somewhat more refined notion of poly-time than that discussed above. The idea is to introduce the notion of *invited messages*: a protocol machine may *invite* the adversary to send it a specific message, and the adversary may *invite* the environment to send it a specific message. We then stipulate that such invited messages are ignored for the purposes of satisfying the flow-boundedness constraint; however, they are also ignored for the purposes of bounding running times. Formally, this simply amounts to defining f_{ea} and f_{ap} to count only *uninvited messages*.

Using this invitation mechanism, we may allow the adversary to send certain "control messages" to the protocol, "free of charge", so to speak. Of course, the protocol designer must ensure that these invited messages do not upset the running-time bounds — this typically amounts to a very simple form of "amortized analysis". Our main use of the invitation mechanism is in the treatment of corruptions.

In our analysis of the application of our framework to many known "use cases", we have found that by using the invitation mechanism, along with a few simple design conventions, the flow-boundedness constraint does not present any insurmountable problems.

The flow-boundedness constraint may also seem to limit the adversary in a way that would rule out otherwise legitimate attacks — see Note 7.3 for a discussion on why this is not the case.

Section 8: Conventions regarding corruptions and ideal functionalities

Perhaps somewhat surprisingly, our fundamental theorems, such as the composition theorem, are completely independent of our mechanism for modeling corruptions, which is layered on top of the basic framework as a set of conventions. This section describes these conventions.

Basically, the environment may choose to corrupt a "top level" regular protocol machine by sending it a special corrupt message. When this happens, the chosen machine responds by notifying the adversary that it is corrupted. This notification may include some or all of the current internal state of the corrupted machine — this depends on whether one wants to model secure erasures or not.

After such a machine M is corrupted, it essentially acts as a slave to the adversary, in the following sense: any messages that M subsequently receives are forwarded to the adversary, and the adversary may instruct M to send a message to another machine. More precisely, the adversary may instruct M to send the special corrupt message to a subroutine of M, and it may instruct M to send an arbitrary message to its ideal peer. By sending such "subroutine corruption instructions", the adversary may recursively corrupt all the regular machines in the subtree of the dynamic call graph rooted at M. Moreover, for each subroutine of M extant at the time of M's corruption, the corresponding "subroutine corruption instructions" are invited messages, which means the adversary is free to carry out this recursive corruption procedure, without breaking the flow-boundedness

constraint. Similarly, the adversary is invited to send an instruction to M that will cause M to send the special corruption message to its ideal peer — the behavior of the ideal peer upon receiving this message is entirely protocol dependent.

Our model of corruptions enforces a strict hierarchical pattern of corruption, so that if Q is a subroutine of P, and Q is corrupted, then P must be corrupted as well.

This section also includes the definition of an *ideal protocol*, which is a particularly simple type of protocol, for which a single instance of the protocol consists of just an ideal machine, along with regular peers, so-called "dummy parties", that just act as "wires", each of which connects the ideal machine to the caller of the dummy party. The logic of the ideal machine is called an *ideal functionality*.

We also define the notion of *hybrid protocols*. If $\mathcal{F}_1, \ldots, \mathcal{F}_k$ are ideal functionalities, then we say that a protocol Π is an $(\mathcal{F}_1, \ldots, \mathcal{F}_k)$-hybrid protocol of the only non-trivial ideal machines used by Π are instances of the $\mathcal{F}_i$'s.

Section 9: protocols with joint state

Consider an $\mathcal{F}$-hybrid protocol Π. Note that this means that even a single instance of Π can use multiple instances of $\mathcal{F}$. These subprotocol instances will be independent, and thus their potential implementations will also be independent — in particular, they may not share joint state. Thus, if $\mathcal{F}$ represents, say, an authenticated channel, we will not be able to implement all these different $\mathcal{F}$-instances using, say, the same signing key.

This section provides a construction and a theorem — the so-called JUC theorem [CR03] — that allows us to sidestep this issue. The basic idea is this. First, we transform $\mathcal{F}$ into its multi-session version $\widehat{\mathcal{F}}$. Second, we transform Π into a "boxed" protocol $[\Pi]_{\mathcal{F}}$, in which all of the individual regular protocol machines that belong to one "party" will be run as virtual machines inside a single "container" machine, where calls to the various $\mathcal{F}$-instances are trapped by the container, and translated into calls to a single running instance of $\widehat{\mathcal{F}}$. Our JUC Theorem (Theorem 9) then states that $[\Pi]_{\mathcal{F}}$ emulates Π.

Of course, this is only interesting if we can design a protocol $\widehat{\Pi}$ that emulates $\widehat{\mathcal{F}}$. For our example above, where $\mathcal{F}$ models an authenticated channel, this can be done in a straightforward way, assuming some kind of ideal functionality $\mathcal{F}_{\text{ca}}$ which represents a "certificate authority". Then applying ordinary composition and transitivity, we get a $\mathcal{F}_{\text{ca}}$-hybrid protocol Π_1 that emulates Π.

This type of application is one of the main motivations for the JUC theorem — it allows us to replace a slow authenticated channel (i.e., between a user and a certificate authority) with a faster one (i.e., one based on signatures).

We note that unlike the composition theorem, the JUC theorem *does* depend on our conventions regarding corruptions, and its proof relies in an essential way on the invitation mechanism.

We also point out that our construction of $[\Pi]_{\mathcal{F}}$, which runs many virtual machines inside one container, is necessitated by the fact that our framework does not provide for "joint subroutines" — the single running instance of $\widehat{\mathcal{F}}$ may only be a subroutine of one machine, which is our $[\Pi]_{\mathcal{F}}$. This is actually quite natural, and in our application to authenticated channels, it closely resembles how protocols are actually used and deployed in practice. It also meshes well with our principle of hierarchical corruption. Moreover, we feel that any reasonable framework should support this type of virtualization, as this seems to be a fundamental abstraction.

Section 10: an extension: common functionalities

In this section, we present an extension to our framework, introducing the notion of *common functionalities*, which will involve some revisions to the definitions and theorems in the previous sections.

The goal is to build into the framework the ability for protocol machines that are not necessarily peers, and also (potentially) for the environment, to have access to a "shared" or "common" functionality.

For example, such a common functionality may represent a *system parameter* generated by a trusted party that is accessible to all machines in the system. By restricting the access of the environment, we will also be able to model a *common reference string (CRS)* — the difference being that a simulator is allowed to "program" a CRS, but not a system parameter. Using this mechanism, we can also model *random oracles*, both *programmable* and *non-programmable*.

We stress that our modeling of a CRS differs from the usual UC modeling, e.g., in [CF01]. Namely, [CF01] model a CRS as a hybrid ideal functionality, which means that every *instance* of a subprotocol gets its own CRS. Using the JUC theorem, it is then possible to (non-trivially) translate many instances of a protocol with individual CRSs into one instance of a multiple-use protocol that uses only one CRS [CR03]. We believe that our modeling of a CRS is more natural and direct, although one could of course formulate a CRS ideal functionality in GNUC.

Without some kind of *set up assumption*, such as a CRS, it is impossible to realize many interesting and important ideal functionalities [CF01]. Moreover, with a CRS, it is feasible to realize any "reasonable" ideal functionality [CLOS02] under reasonable computational assumptions. (While the results in [CF01] use a CRS that is formalized as a hybrid functionality, they also carry over easily to our CRS formalization.)

In contrast to CRSs, system parameters are not essential from a theoretical point of view, but often yield more practical protocols.

Section 11: comparison with UC05 and other frameworks

In this section, we compare our proposed UC framework with Canetti's UC05 framework [Can05], and (more briefly) with some other frameworks, including Küsters' IITM framework [Küs06], and the Reactive Simulatability framework of Backes, Pfitzmann and Waidner [BPW07].

Section 12: examples

In this section, we give some examples that illustrate the use of our framework. These examples include several fundamental ideal functionalities, carefully and completely specified in a way that is compatible with our framework. These examples also include a more detailed discussion of how the JUC theorem may be used together with a certificate authority to obtain protocols that use authenticated channels, and that are designed in a modular way within our framework, but yet are quite practical. Finally, the results in [BCL$^+$05], on secure computation without authentication, are considered, and we discuss how these may be translated into our framework.

3 Machine models

3.1 Some basic notation and terminology

Throughout, we assume a fixed, finite **alphabet Σ** of symbols, which shall include all the usual characters found on a standard American keyboard.

We assume a fixed **programming** language for writing programs; programs are written as strings over $\mathbf{\Sigma}$. We assume that our programs have instructions for generating a random bit. We also assume that a program takes as input a string over $\mathbf{\Sigma}$ and outputs a string over $\mathbf{\Sigma}$. For $\pi, \alpha \in \mathbf{\Sigma}^*$, if we view π as a program and α as an input, the notation $\beta \leftarrow \mathsf{Eval}(\pi, \alpha)$ means that the program π is run on input α, and if and when the program terminates with an output, that output is assigned to β. If $\omega \in \{0,1\}^\infty$ is an infinite bit string, we may also write $\beta \leftarrow \mathsf{Eval}(\pi, \alpha; \omega)$ to mean that π is run on input α, using ω as the source of random bits, and the output is assigned to β. When discussing the execution of such probabilistic programs, we are assuming a probability space on $\{0,1\}^\infty$ analogous to the Lebesgue measure on $[0,1]$.

Finally, we assume a fixed **list-encoding function** $\langle \cdot \rangle$: if $\alpha_1, \ldots, \alpha_n$ are strings over $\mathbf{\Sigma}$, then $\langle \alpha_1, \ldots, \alpha_n \rangle$ is a string over $\mathbf{\Sigma}$ that encodes the list $(\alpha_1, \ldots, \alpha_n)$ in some canonical way. It may be the case that a string does not encode a list, but if it does, then that list must be unique. We assume the encoding and decoding functions may be implemented in polynomial time. We also assume that the length of $\langle \alpha_1, \ldots, \alpha_n \rangle$ is at least n plus the sum of the lengths of $\alpha_1, \ldots, \alpha_n$. For example, one could define $\langle \alpha_1, \ldots, \alpha_n \rangle := \mathsf{|Quote}(\alpha_1)\mathsf{|} \cdots \mathsf{|Quote}(\alpha_n)$, where $\mathrm{Quote}(\alpha)$ replaces each occurrence of | in α with the "escape sequence" \|, and each occurrence of \ with \\.

3.2 Interactive machines

An **interactive machine (IM)** M consists of a triple $(id, \pi, state)$, where id is its **machine ID**, π is its **program**, and $state$ is its **current state**, all of which are strings over $\mathbf{\Sigma}$. Given $id_0, msg_0 \in \mathbf{\Sigma}^*$, the machine M computes $\gamma \leftarrow \mathsf{Eval}(\pi, \langle id, state, id_0, msg_0 \rangle) \in \mathbf{\Sigma}^*$. The output γ is expected to be of the form $\langle state', id_1, msg_1 \rangle$, for some $state', id_1, msg_1 \in \mathbf{\Sigma}^*$. Such a computation is called an **activation of** M.

At the end of such an activation , the current state of M is modified, so that now M consists of the triple $(id, \pi, state')$ — although the state of M changes, we still consider it to be the same machine (and the values of id and π will never change). Intuitively, msg_0 represents an incoming message that was sent from a machine M_0 with machine ID id_0, while msg_1 represents an outgoing message that is to be delivered to a machine M_1 with machine ID id_1.

We say that a program π is a **valid IM program** if the following holds: for all $id, state, id_0, msg_0 \in \mathbf{\Sigma}^*$, whenever the computation $\mathsf{Eval}(\pi, \langle id, state, id_0, msg_0 \rangle)$ halts, its output is the encoding of a triple, that is, a string of the form $\langle state', id_1, msg_1 \rangle$ for some $state', id_1, msg_1 \in \mathbf{\Sigma}^*$. Note that a valid IM program may be probabilistic, and may fail to halt on certain inputs and certain coin-toss sequences, but whenever it does halt, its output must be of this form.

Whenever we consider an IM, we shall always assume that its program is a valid IM program, even if this assumption is not made explicit. Note that while there is no algorithm that decides whether a given program π is a valid IM program, we can easily convert an arbitrary program π into a valid IM program $\bar{\pi}$, which runs as follows: $\bar{\pi}$ simply passes its input to π, and if π halts with an output that is a valid triple, then $\bar{\pi}$ outputs that triple, and if π halts with an output that is not a valid triple, then $\bar{\pi}$ outputs the triple $\langle \langle\,\rangle, \langle\,\rangle, \langle\,\rangle \rangle$. Thus, $\bar{\pi}$ acts like a **software sandbox** in which π is allowed to run. We will use this "sandboxing" idea extensively throughout the paper to make similar restrictions on programs.

To simplify notation, if π is a valid IM program, we express a single activation as

$$\langle state', id_1, msg_1 \rangle \leftarrow \mathsf{Eval}(\pi, \langle id, state, id_0, msg_0 \rangle).$$

3.3 Machine models and running time

To complete our definition of an interactive machine, we should fully specify the programming language, the execution model, and the notion running time. However, as long as we restrict ourselves to models that are polynomial-time equivalent to Turing machines, none of these details matter, except for the specific requirements discussed in the next two paragraphs.

We assume that there is a universal program π_{u} that can efficiently simulate any other program, and halt the simulation in a timely fashion if it runs for too long. To make this precise, define $B(\pi, \alpha, \mathtt{1}^t, \omega)$ to be $\langle\, \mathtt{beep} \,\rangle$ if the running time of the computation $\beta \leftarrow \mathsf{Eval}(\pi, \alpha; \omega)$ exceeds t, and $\langle\, \mathtt{normal}, \beta \,\rangle$ otherwise. The requirement is that $\mathsf{Eval}(\pi_{\mathrm{u}}, \langle\, \pi, \alpha, \mathtt{1}^t \,\rangle; \omega)$ computes $B(\pi, \alpha, \mathtt{1}^t, \omega)$ in time bounded by a fixed polynomial in the length of $\langle\, \pi, \alpha, \mathtt{1}^t \,\rangle$.

We also assume that the programming language allows us to place a "comment" in a program, so that the comment can be efficiently extracted from the program, but does not in any way affect its execution. While not strictly necessary, this assumption will be convenient.

Finally, we define a simple notion of polynomial time for IMs. Let π be an IM program. We say that π is **multi-activation polynomial-time** if the following holds: there exists a polynomial p such that for all

$$id, id_0, msg_0, \ldots, id_{k-1}, msg_{k-1} \in \mathbf{\Sigma}^*, \tag{1}$$

if

$$n := |\langle\, id, id_0, msg_0, \ldots, id_{k-1}, msg_{k-1} \,\rangle|,$$

then with probability 1, the following computation runs in time at most $p(n)$:

$$\begin{array}{l} state \leftarrow \langle\, \rangle \\ \text{for } i \leftarrow 0 \text{ to } k-1 \\ \qquad \langle\, state, id'_i, msg'_i \,\rangle \leftarrow \mathsf{Eval}(\pi, \langle\, id, state, id_i, msg_i \,\rangle) \end{array}$$

In addition, if there is a polynomial q such that for all strings as in (1), we have $|msg'_i| \leq q(|msg_i|)$ for $i = 0, \ldots, k-1$ with probability 1, we say that π is **I/O-bounded**.

3.4 Systems of interactive machines

Our next goal is to define a system of interactive machines. To do this, we first define two of the basic components of such a system.

The first basic component of such a system is a **run-time library**. This is a function $Lib : \mathbf{\Sigma}^* \to \mathbf{\Sigma}^*$ with a finite domain.

Intuitively, Lib maps **program names** to **programs** (i.e., code). The definition implies that a given run-time library can only define a finite number of program names. The idea is that all programs for machines executing in a system of interactive machines will come from the run-time library. We write $Lib(name) = \perp$ to mean that the given program name $name$ is not in the domain of Lib.

The second basic component of a system of IMs is a **name map**. This is a function $NameMap : \mathbf{\Sigma}^* \to \mathbf{\Sigma}^* \cup \{\perp\}$ that can be evaluated in deterministic polynomial time. In addition, we require that $NameMap(\langle\, \rangle) = \perp$.

Intuitively, $NameMap$ maps a machine ID to a program name, or to $\perp$ if there is no corresponding program name. The machine ID $\langle\, \rangle$ is special, and we insist that there is no program name associated with this machine ID; more specifically, the string $\langle\, \rangle$ will essentially be used as the name of a "master machine" that carries out the execution of a system of interactive machines, but is itself not a part of the system.

A given software library *Lib* and name map *NameMap* determine the **program map** *ProgMap* : $\mathbf{\Sigma}^* \to \mathbf{\Sigma}^* \cup \{\bot\}$ defined as follows: for $id \in \mathbf{\Sigma}^*$, if $\mathit{NameMap}(id) = \bot$, then $\mathit{ProgMap}(id) := \bot$, and otherwise, if $\mathit{NameMap}(id) = \mathit{name} \neq \bot$, then $\mathit{ProgMap}(id) := \mathit{Lib}(\mathit{name})$.

Intuitively, such a program map associates to each machine ID a corresponding program, or the symbol $\bot$ if there is no such program. By definition, a program map can be evaluated in deterministic polynomial time.

A **system of interactive machines (IMs)** is a triple $S = (id_{\text{init}}, \mathit{NameMap}, \mathit{Lib})$, where id_{init} is a machine ID, *NameMap* is a name map, and *Lib* is a run-time library. We require that if *ProgMap* is the corresponding program map, then $\pi_{\text{init}} := \mathit{ProgMap}(id_{\text{init}}) \neq \bot$.

We shall next define the execution of such a system S. Such an execution takes an **external input** $\alpha \in \mathbf{\Sigma}^*$, and produces an **external output** $\beta \in \mathbf{\Sigma}^*$, if the execution actually halts.

Intuitively, the execution of the system proceeds as a sequence of activations. The machine $M_{\text{init}} := (id_{\text{init}}, \pi_{\text{init}}, \langle\,\rangle)$ is the **initial machine**, and it performs the first activation: starting with the (rather arbitrary) initial state $\langle\,\rangle$, it is given the message $\langle\, \texttt{init}, \alpha \,\rangle$, apparently originating from the "master machine". In general, whenever a machine M performs an activation, it receives a message msg_0 from a machine M_0 with ID id_0, and then M's state is updated, and a message msg_1 is sent to a machine M_1 with ID id_1, after which, the next activation is performed by M_1. If there is no machine M_1 with the ID id_1, then a new machine is created afresh, using the program determined by the program map. Two special conditions may arise:

- $id_1 = \langle\,\rangle$ (i.e., id_1 is the ID of the "master machine"). In this case, the execution of the system terminates, and the output of the system is defined to be msg_1.
- $id_1 \neq \langle\,\rangle$ but $\mathit{ProgMap}(id_1) = \bot$ (i.e., the program corresponding to id is undefined). In this case, a special error message (apparently originating from the "master machine") is sent back to M.

We formalize of the above intuition by describing in detail the algorithm representing the execution of a system S as above. To streamline the presentation, we make use of an "associative array" or "lookup table" *Machine*, which maps machine IDs to program/state pairs. Initially, $\mathit{Machine}[id] = \bot$ (i.e., is undefined) for all $id \in \mathbf{\Sigma}^*$; as the algorithm proceeds, for various values id, π, state, we will set $\mathit{Machine}[id]$ to (π, state), so that the triple $(id, \pi, \mathit{state})$ represents an IM.

Here is the algorithm:

Input: $\alpha \in \mathbf{\Sigma}^*$
Output: $\beta \in \mathbf{\Sigma}^*$

$id \leftarrow id_{\text{init}}$; $id_0 \leftarrow \langle\,\rangle$; $msg_0 \leftarrow \langle\, \texttt{init}, \alpha \,\rangle$

while $id \neq \langle\,\rangle$ {
    /* *message* msg_0 *is being passed from* id_0 *to* id */

    if $Machine[id] = \perp$
        then { $\pi \leftarrow ProgMap(id)$; $state \leftarrow \langle\,\rangle$ } // *create new machine*
        else $(\pi, state) \leftarrow Machine[id]$ // *fetch description of existing machine*

    /* *perform one activation:* */
    $\langle\, state', id_1, msg_1 \,\rangle \leftarrow \mathsf{Eval}(\pi, \langle\, id, state, id_0, msg_0 \,\rangle)$
    $Machine[id] \leftarrow (\pi, state')$ // *update state*

    if $id_1 \neq \langle\,\rangle$ **and** $ProgMap(id_1) = \perp$
        then { $id_0 \leftarrow \langle\,\rangle$; $msg_0 \leftarrow \langle\, \texttt{error}, id_1, msg_1 \,\rangle$ } // *error – undefined program*
        else { $id_0 \leftarrow id$; $msg_0 \leftarrow msg_1$; $id \leftarrow id_1$ } // *pass message* msg_1 *to* id_1
}
$\beta \leftarrow msg_0$; **output** β

4 Structured systems of interactive machines

In the previous section, we defined the notion of a system of IMs. This is a very simple and general notion, but is far too general for our purposes. Our goal in this section is to define the notion of a *structured* system of IMs. It is with respect to such structured systems that our main theorems (such as the Composition Theorem) will be formulated.

In defining a structured system of IMs, we will define once and for all the *name map* (which maps machine IDs to program names) that will be used for *all* structured systems (although in §10 we will define the notion of an *extended* structured system, which will have some extra features). In addition, we will define various classes of machines that satisfy certain constraints — all of these constraints can be imposed "locally", by software sandboxing, and we will not require any form of "global controller", beyond the simple "master machine" that carries out the execution of a (general) system of IMs.

4.1 Some basic syntax

At a high level, in a structured system of IMs, there are three classes of machines: **environment**, **adversary**, and **protocol**. Moreover, in any execution of a structured system, there will be only one instance of an environment machine, and only one instance of an adversary machine, so there will be no confusion when we speak of "the environment" and "the adversary".

Syntactically, what distinguishes these three classes is the form of their machine IDs. For specificity, we will say that the environment has machine ID $\langle\, \texttt{env} \,\rangle$, the adversary has machine ID $\langle\, \texttt{adv} \,\rangle$. Protocol machines will have machine IDs of the form $\langle\, pid, sid \,\rangle$, where *pid* is a string representing the **party ID** (or **PID**) of the machine and *sid* is a string representing the **session ID** (or **SID**) of the machine. We say that two such IDs are **peers** if their SIDs match, and two

protocol machines whose machine IDs are peers are also called **peers**.

We next describe constraints on the format of PIDs and SIDs of protocol machines. We divide protocol machines into two subclasses: **regular** and **ideal**. These two subclasses of machines are distinguished by the form of their PIDs: regular machines have a PID of the form $\langle\, \texttt{reg}, \mathit{basePID} \,\rangle$, while ideal protocol machines have the PID $\langle\, \texttt{ideal} \,\rangle$.

As will be described in detail below, regular machines may invoke other regular machines as "subroutines". When a machine P invokes a machine Q as a subroutine, we say that P is the **caller** of Q and Q is a **subroutine** of P. This caller/subroutine relationship will play a crucial role. An essential constraint we shall impose is the following:

every regular machine has a unique caller.

This is an important constraint, and we shall have much more to say about it. However, for the time being, we indicate how this constraint relates to the syntactic structure of SIDs.

SIDs for all protocol machines (regular and ideal) are structured as "path names", which will be used to explicitly represent the caller/subroutine relationship of regular machines. That is, SIDs will be of the form

$$\langle\, \alpha_1, \ldots, \alpha_k \,\rangle. \tag{2}$$

Some terminology will be helpful. We say an SID sid' is an **extension** of an SID sid if for some $\alpha_1, \ldots, \alpha_\ell \in \mathbf{\Sigma}^*$, and for some $k \leq \ell$, we have $\mathit{sid} = \langle\, \alpha_1, \ldots, \alpha_k \,\rangle$ and $\mathit{sid}' = \langle\, \alpha_1, \ldots, \alpha_k, \alpha_{k+1}, \ldots, \alpha_\ell \,\rangle$. If $\ell > k$, we say sid' is a **proper extension** of sid, and if $\ell = k + 1$, we say sid' is a **one-step extension** of sid.

Next, we define a simple syntactic relation on *machine IDs*. Given two machine IDs $\mathit{id} = \langle\, \mathit{pid}, \mathit{sid} \,\rangle$ and $\mathit{id}' = \langle\, \mathit{pid}', \mathit{sid}' \,\rangle$, we say that the ID id is a **parent** of the ID id' if $\mathit{pid} = \mathit{pid}'$ and sid' is a one-step extension of sid. Evidently, the parent of any ID is uniquely determined. As usual, if id is a parent of id', then we also say that id' is a **child** of id. Moreover, we say that id is an **ancestor** of id' if $\mathit{pid} = \mathit{pid}'$ and sid' is a proper extension of sid.

With this definition in hand, we can state more precisely the constraint that regular machines will satisfy: for any regular machine with ID id, its caller is either the environment or the regular machine whose ID is the parent of id. This will be more precisely formulated below.

For an SID of the form (2), the last component, namely, α_k, is called the **basename** of the SID. This basename also must be of a particular form, namely, $\langle\, \mathit{protName}, \mathit{sp} \,\rangle$. Here, *protName* specifies the name of the program being executed by the machine. For protocol machines, we require that program names be of the form `prot-`*name*. We call a program name of this form a **protocol name**. Also, *sp* is an arbitrary string, which we call a **session parameter** — its contents is determined by the application.

That completes the description of the format of the machine IDs of protocol machines. Now that we have defined the format of all machine IDs, we can define the *name map* that will be used in every structured system of IMs. The name map will map the machine ID $\langle\, \texttt{env} \,\rangle$ to the program name `env`, the machine ID $\langle\, \texttt{adv} \,\rangle$ to the program name `adv`, and a machine ID of the form $\langle\, \mathit{pid}, \langle \ldots, \langle\, \mathit{protName}, \mathit{sp} \,\rangle \rangle \,\rangle$ to the protocol name *protName*.

Of course, certain aspects of this definition of the name map are somewhat arbitrary, but we have included them for specificity.

4.2 Overall execution pattern

We now describe the overall execution pattern of a structured system of IMs.

The environment is the initial machine. In any execution of the system, it will be given the external input and will perform the first activation.

In all executions of interest, the external input will be of the form $\mathtt{1}^\lambda$, where λ represents the security parameter. We will also insist on the following constraints:

C1: *The environment is the only machine that can produce an external output (and thereby halt the execution of a structured system).*

C2: *Every message sent from one machine to another is of the form* $\langle\, \mathtt{1}^\lambda, m, i_1, \ldots, i_k \,\rangle$*, where* λ *is the security parameter;* m *is called the* ***message body****, and* $i_1, \ldots, i_k$ *are called* ***invitations****; invitations may be included only in messages from a protocol machine to the adversary, and in messages from the adversary to the environment.*

Note that there is no constraint placed on the format of the external output generated by the environment.

These constraints are easily enforced by sandboxing. To implement constraint C1, the sandbox would check for an illegal attempt to generate an external output — if this happens, the message would simply be sent to the adversary (this will always be legal). Similarly, if a machine generates a message that does not satisfy constraint C2, the message is translated into one that does; the details of this translation are not so important, but we can assume that the message is simply replaced by $\langle\, \mathtt{1}^\lambda, \langle\, \rangle \,\rangle$.

Invitations will play a special role when we discuss running time and other resource bounds in §6 and §7. For now, we leave the reader with the following, admittedly vague, intuition: when a machine P sends an invitation i to a machine Q, this is a *hint* that Q should send the message i back to P.

Because the security parameter is always transmitted as part of a message, when describing protocols at high level, we will generally leave the security parameter out of the description. Whenever we say "send message m", the low-level message that is transmitted is actually $\langle\, \mathtt{1}^\lambda, m \,\rangle$. Whenever we say "send message m with invitations for $i_1, \ldots, i_k$", the low-level message that is transmitted is actually $\langle\, \mathtt{1}^\lambda, m, i_1, \ldots, i_k \,\rangle$.

4.3 Constraints on the environment

In addition to the constraints imposed in §4.2, we shall impose additional constraints on the environment.

C3: *The environment may send messages to the adversary, and may send messages to* regular *protocol machines; however, it may* not *send messages to* ideal *protocol machines.*

C4 (peer/ancestor free constraint): *The set of regular protocol machines to which the environment sends messages must be* ***peer/ancestor free****, which means that no one machine in the set has an ID that is a peer of an ancestor of another.*

Constraint C4 can be rephrased as saying that among the regular protocol machines to which the environment sends messages, no one machine may have an SID that is a proper extension of the SID of any other machine. These constraints are easily enforced by sandboxing.

Note 4.1. Constraint C4 is not necessary to prove any of the theorems in this paper. However, this constraint can be justified, and including it helps to justify some other constraints — see Note 8.10. □

Note 4.2. The set of constraints taken together will imply that the only machines from which the environment receives messages are the adversary and the regular protocol machines to which it has already sent a message. □

Note 4.3. If the environment specifies an SID of the form $\langle \alpha_1, \ldots, \alpha_k \rangle$, we do insist that α_k is a well-formed basename; however, we do not make any other constraints: the program name need not be specified by the library, and $\alpha_1, \ldots, \alpha_{k-1}$ may be completely arbitrary strings. □

4.4 Constraints on the adversary

In addition to the constraints imposed in §4.2, we shall impose an additional constraint on the adversary:

C5: *The adversary may send messages to the environment. The adversary may send a message to a protocol machine (regular or ideal) only if it has previously received a message from that machine.*

Again, this constraint is easily enforced by sandboxing.

Note 4.4. This constraint means that the adversary never causes a protocol machine to be created. □

Note 4.5. The set of constraints taken together will imply that the adversary may receive messages from the environment and from any existing protocol machine (regular or ideal). □

Note 4.6. Constraint C5 may seem like an unrealistic restriction, since we normally think of the adversary as being generally unfettered. However, this constraint could alternatively be imposed by constraining protocol machines, so that messages received from the adversary too early would simply be ignored and bounced back to the adversary. Thus, this constraint may be viewed as a constraint on the protocol, rather than the adversary. □

4.5 Constraints on ideal protocol machines

In addition to the constraints imposed in §4.2, we shall impose an additional constraint on ideal protocol machines:

C6: *The only machines to which an ideal protocol machine may send messages are (i) the adversary and (ii) regular peers of the ideal machine from which it has previously received a message.*

Again, this constraint is easily enforced by sandboxing.

Note 4.7. The set of constraints taken together will imply that an ideal machine will receive messages only from the adversary and from its regular peers. □

4.6 Constraints on regular protocol machines

In addition to the constraints imposed in §4.2, we shall impose additional constraints on regular protocol machines. These constraints will require sandboxing — not only to enforce certain behavior, but to structure the computation in a particular way. Thus, we will insist that the program of the machine is structured as an "inner core", which is running inside a "sandbox". The inner core is quite arbitrary; however, the sandbox is fully specified here.

Let us call this regular machine M, and suppose *id* is its machine ID. Let *parentId* be the machine ID that is the parent of *id*. Because of all the constraints we are imposing across the system, the first message received by M (and the one that brings it into existence) will be either the environment or a regular protocol machine with ID *parentId*; moreover, whichever machine

sends this first message, the other machine will never send it a message. We call the machine sending this first message the **caller of** M, and let us denote it here by C. We will also say that C **invokes** M.

4.6.1 Caller ID translation

The first function of the sandbox is "caller ID translation", which works as follows:

C7 (caller ID translation): *When M receives a message from C, then if C is the environment, the sandbox changes the source address of the message to parentId before passing the message for further processing to the inner core. Similarly, when the inner core generates a message whose destination address is parentId, and C is the environment, then the sandbox sends the message instead to the environment.*

To implement caller ID translation, the sandbox will have to store the true identity of C; however, this information will never be directly accessible to the inner core. This can be achieved by structuring the state of M to be of the form $\langle\, \mathit{sandboxState}, \mathit{coreState} \,\rangle$, and only passing *coreState* to the inner core.

Note 4.8. Caller ID translation is useful in that it effectively hides the true identity of caller of M, which will be crucial in proving the main composition theorem. □

4.6.2 Communication constraints

The second function of the sandbox will be to enforce constraints on the machines to which M may send messages.

C8 (communication constraints): *M is only allowed to send messages to the following machines:*

- *the adversary;*
- *its ideal peer;*
- *machines whose machine IDs are children of id;*
- *the caller of M, via caller ID translation — the inner core is not allowed to address a message directly to the environment.*

We may assume that any attempt by the inner core to send a message to any other machine will simply be forwarded by the sandbox to the adversary.

4.6.3 Subroutine constraints

In §4.6.2, we allowed M to send messages to machines whose IDs are children of *id*. Such a machine S is called a **subroutine of** M. Because of all the constraints we have imposed, M must in fact be the caller of S, i.e., the first machine to send a message to S and to bring it into existence.

We shall require that M's program explicitly *declares* the program names of any subroutine it may invoke during any activation. As we are assuming that a "comment" can be placed in a program (see §3.3), this declaration can be placed in the program as a "comment", using any convenient format to encode the declaration. This could also be achieved by modifying the notion of a library, so that it allows us to associate such "comments" with the program, instead of embedding

them in the program itself. The details are really not important. Let us call this declaration the **subroutine declaration of** M.

We require, of course, that the sandbox actively enforces the subroutine constraints:

C9 (subroutine constraints): *M only invokes subroutines whose program names are explicitly declared in its subroutine declaration.*

Note 4.9. The set of all constraints taken together will imply that a regular protocol machine will receive messages only from its caller, its ideal peer (unique, if it exists at all), its subroutines (if any), and the adversary. □

Note 4.10. Regular and ideal protocol machines with a given program differ only in their PID. Thus, the various constraints for ideal and regular protocol machines will be enforced by a single sandbox — the behavior of this sandbox will be determined by the format of the PID. □

4.7 The dummy adversary

The dummy adversary is a special adversary that essentially acts as a router between the environment and protocol machines. It works as follows:

- if it receives a message of the form $\langle\, id, m \,\rangle$ from the environment, and it has previously received a message from a protocol machine with machine ID id, then it sends m to that protocol machine; otherwise, it sends the message $\langle\, \texttt{error} \,\rangle$ back to the environment;
- if it receives a message m from a protocol machine with machine ID id, it sends the message $\langle\, id, m \,\rangle$ to the environment;
- if it receives a message m along with invitations for $i_1, \ldots, i_k$ (see §4.2) from a protocol machine with machine ID id, it sends the message $\langle\, id, m \,\rangle$ along with invitations for $\langle\, id, i_1 \,\rangle, \ldots, \langle\, id, i_k \,\rangle$ to the environment.

That completely specifies the logic of the dummy adversary. We denote the dummy adversary by A_{d}.

Intuitively, a message $\langle\, id, m \,\rangle$ is an instruction to the dummy adversary to send the message m to the machine whose ID is id. Also, note that when the dummy adversary receives invitations from a protocol machine, it translates these invitations into invitations to the environment, which themselves are dummy adversary instructions to deliver the invited message to the machine that originally issued the invitation; thus, when the environment delivers this invited instruction to the dummy adversary, the latter will send the original invited message to the appropriate protocol machine.

5 Protocols

5.1 Definition of a protocol

We next define what we formally mean by a **protocol**. A protocol is a run-time library that satisfies the following four requirements.

P1: *The library should only define programs for* protocol names. *In particular, a protocol does not define the program for the environment or the adversary.*

P2: *The programs defined in the library should satisfy all the constraints discussed in §4 that apply to protocol machines.*

Before we state the next requirement, recall that the subroutine constraint (C9) in §4.6.3 says that the program for a regular protocol machine must explicitly declare the program names of the machines it is allowed to invoke as subroutines during any activation.

P3: *For each program name defined by the library, every program name declared by its corresponding program must also be defined by the library.*

Before we state the last requirement, we need to define the **static call graph** of the protocol. The set of nodes of this graph is the domain of the library, that is, the set of protocol names defined by the library; there is an edge from one program name to another if the subroutine declaration in the program associated with the first name allows a call to the second (by requirement P3, the second name is defined by the library, and so is a node in the graph).

P4: *The static call graph is acyclic and has a unique node of in-degree 0.*

The node r defined in P4 is called the **root** of the protocol — we may also say that the protocol is **rooted at** r. It follows from the definition that every node in the static call graph is reachable from r.

5.2 Subprotocols

We next define what it means for one protocol to be a subprotocol of another. The definition is quite natural, and is easy to define in a simple, mathematically rigorous way, as follows.

Let Π and Π' be protocols, as defined in §5.1 Recall that these are functions with finite domains, say, D and D'. We say that Π' **is a subprotocol of** Π if $D' \subset D$ and the restriction of Π to D' is equal to Π'.

Let Π be a protocol which defines a protocol name x. Let D' be the set of nodes reachable from x in the static call graph of Π. Let Π' be the restriction of Π to D'. Then it is easy to see that Π' is a subprotocol of Π with root x. We call Π' the **subprotocol of** Π **rooted at** x, and we denote it by $\Pi \mid x$.

It is clear that if Π' is an arbitrary subprotocol of a protocol Π, then $\Pi' = \Pi \mid x$ for some x.

We next wish to define a subprotocol substitution operator. Let Π be a protocol rooted at r, which defines a protocol name x. Let D'' be the set of nodes of the static call graph of Π that are reachable by paths from r that *do not* go through x. We write $\Pi \setminus x$ to denote the restriction of Π to D''.

Now, let $\Pi' := \Pi \mid x$, and let Π'_1 be any other protocol rooted at x. We say that Π'_1 **is substitutable for** Π' **in** Π if the following condition holds: for every y in the domain of $\Pi \setminus x$, if y is also defined by Π'_1, then the definition of y in Π'_1 agrees with that in Π. If this condition holds, we write $\Pi[\Pi'/\Pi'_1]$ to denote the protocol Π_1 rooted at r such that $\Pi \setminus x = \Pi_1 \setminus x$ and $\Pi_1 \mid x = \Pi'_1$. It is easy to see that Π_1 exists and is uniquely determined. In words, $\Pi[\Pi'/\Pi'_1]$ is the protocol obtained by substituting Π'_1 for Π' in Π.

The substitution operation is symmetric. Specifically, the following is easy to prove: if Π'_1 is substitutable for Π' in Π, and if $\Pi_1 := \Pi[\Pi'/\Pi'_1]$, then it holds that Π' is substitutable for Π'_1 in Π_1, and that $\Pi = \Pi_1[\Pi'_1/\Pi_1]$.

5.3 Protocol execution

Let Π be a protocol with root r. Let Z be a program defining an environment, meaning that it satisfies the constraints discussed in §4.2 and §4.3. that apply to the environment. If Z only invokes machines with the same SID with protocol name r, we say that Z is **rooted at** r — note that in this case, Z trivially satisfies Constraint C4 in §4.3. If Z invokes machines whose SIDs are not necessarily the same, but still all with protocol name r, we say that Z is **multi-rooted at** r.

For the remainder of this section, we assume that Π is a protocol rooted at r, and that Z is an environment multi-rooted at r. Also, we assume that A is a program defining an adversary, meaning that it satisfies the constraints discussed in §4 that apply to the adversary. The protocol Π, together with A and Z, define a structured system of IMs, which we denote $[\Pi, A, Z]$.

For a given value of the security parameter λ, we may consider the execution of the system $[\Pi, A, Z]$ on the external input 1^λ. This execution is a randomized process that may or may not terminate.

For such an execution of $[\Pi, A, Z]$, we may define the associated **dynamic call graph**, which evolves during the execution. At any point in time, the nodes of this graph consist of all *regular* protocol machines created during the execution up to that point. There is a an edge from one machine to a second machine if the second is a subroutine of the first.

By our constraints on structured systems, this graph is a forest of trees, where the roots of these trees are precisely the machines directly invoked by Z; all the nodes in any one tree have the same PID; moreover, for any two nodes in this graph, if these two nodes are peers, then the roots of their respective trees are also peers.

In this graph, we may group together all those nodes belonging to trees whose roots are peers; let us also add to such a group any ideal machines that are peers of these nodes. Let us call such a grouping of machines an **instance of** Π. By the observations in the previous paragraph, every protocol machine in the system (regular or ideal) belongs to a unique instance. In addition, we may associate with each instance the common SID of the root nodes belonging to that instance. We call this the **SID of the instance**. In any one instance, all machines will have SIDs that are extensions of the SID of the instance.

If Z is rooted at r, then there will only be one instance. If Z is multi-rooted at r, there may be many instances. Constraint C4 says that for every two distinct instances, the SID of one cannot be an extension of the SID of the other. The total number of instances will be bounded by the number of messages that Z sends to any protocol machine.

The overall pattern of execution can be broken up into **epochs**: an epoch begins with an activation of Z, and ends just before the next activation of Z. At the beginning of the epoch, control passes from Z to either a protocol machine or the adversary. From a protocol machine, control may pass to another protocol machine in the same instance (possibly a new one), or to A, or back to Z (which ends the epoch). From A, control may pass to an existing protocol machine (belonging to any instance), or back to Z (which, again, ends the epoch). The total number of epochs is bounded by the number of messages that Z sends to a protocol machine or to the adversary.

See Fig. 2 for a visual representation of a simple dynamic call graph corresponding to two instances of a two-party protocol. In this figure, circles represent regular protocol machines. Rectangles represent ideal protocol machines, which are connected to their regular peers via a dotted line.

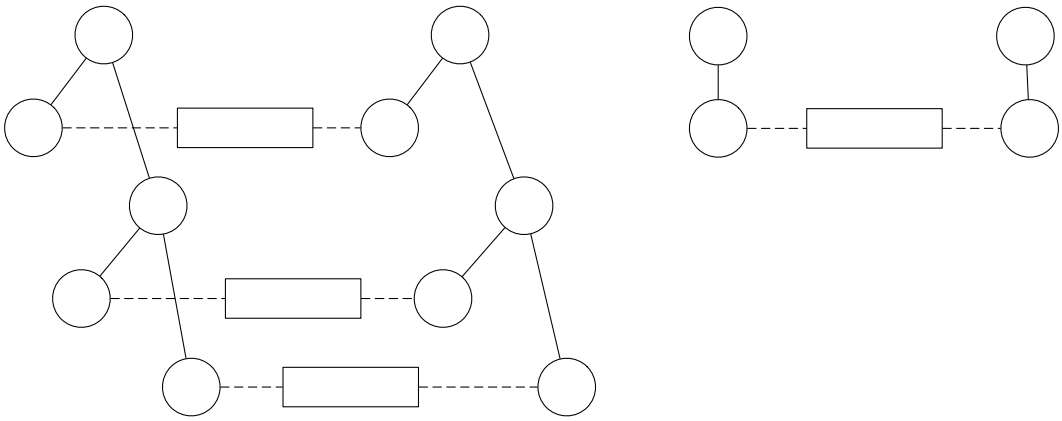


Figure 2: A dynamic call graph

6 Resource bounds

In this section, we discuss various notions of resource bounds, including a definition of a polynomial-time protocol.

Suppose Π is a protocol rooted at r, Z is an environment multi-rooted at r, and A is an arbitrary adversary. Suppose we fix the value of the security parameter λ, and consider the execution of the structured system $[\Pi, A, Z]$ on external input 1^λ

We wish to define several random variables which measure the running time of various machines during this execution. To this end, $\text{Time}_Z[\Pi, A, Z](\lambda)$ denotes the running time of Z, $\text{Time}_A[\Pi, A, Z](\lambda)$ denotes the running time of A, and $\text{Time}_\Pi[\Pi, A, Z](\lambda)$ denotes the sum of the running times of all protocol machines. For convenience, we define

$$\text{Time}_{\Pi,A}[\Pi, A, Z](\lambda) := \text{Time}_\Pi[\Pi, A, Z](\lambda) + \text{Time}_A[\Pi, A, Z](\lambda).$$

We also wish to define several random variables which measure the amount of flow of data between various machines during this execution.

For the purposes of this definition, we measure the length of a message from one machine to another as the length of the string $\langle\, id, id_0, msg_0 \,\rangle$, where id is the machine ID of the recipient, id_0 is the machine ID of the sender, and msg_0 is the low-level message, which is of the form $\langle\, 1^\lambda, \ldots \,\rangle$, as discussed in §4.2.

We also need to distinguish between *invited messages* and *uninvited messages*. Recall that messages from A to Z, and messages from a protocol machine to A, may include invitations (see §4.2). At any point in time, a multi-set of outstanding invitations from A to Z is defined; initially, this multi-set is empty; whenever A sends an invitation to Z, this invitation is added to the multi-set. A message sent from Z to A is called *invited* if it belongs to the multi-set of outstanding invitations, and when this message is sent, it is removed from the multi-set of outstanding invitations; otherwise, the message is called *uninvited*. Analogously, we may categorize each message from A to a protocol machine M as either *invited* or *uninvited*, depending on whether there is an outstanding invitation from M to A for that message.

$\text{Flow}_{Z\to A}[\Pi, A, Z](\lambda)$ denotes the sum of the lengths of all *uninvited* messages sent from Z to A, $\text{Flow}_{A\to\Pi}[\Pi, A, Z](\lambda)$ denotes the sum of the lengths of all *uninvited* messages sent from A to any protocol machine, and $\text{Flow}_{Z\to\Pi}[\Pi, A, Z](\lambda)$ denotes the sum of the lengths of all messages sent

from Z to any protocol machine. Note that there are no invitations sent from protocol machines to Z — in this sense, all messages from Z to protocol machines may be considered uninvited. For convenience, we define

$$\mathrm{Flow}_{Z\to\Pi,A}[\Pi, A, Z](\lambda) := \mathrm{Flow}_{Z\to\Pi}[\Pi, A, Z](\lambda) + \mathrm{Flow}_{Z\to A}[\Pi, A, Z](\lambda).$$

Finally, we define $\mathrm{Flow}^*_Z[\Pi, A, Z](\lambda)$ to be the *total* flow out of Z into Π and A, including both *invited and uninvited messages*.

Our definitions of a polynomial-time protocol, as well as other resource bound conditions to be presented in the next section, will be made in terms of environments which themselves are well behaved, in the following sense:

Definition 1 (well-behaved environment). *Suppose Z is an environment that is multi-rooted at r. We say that Z is a **well-behaved environment** if it is multi-activation polynomial-time (see §3.3), and there exists a polynomial p such that for every protocol Π rooted at r, for every adversary A, and for all λ, the following holds with probability 1:*

$$\mathrm{Flow}^*_Z[\Pi, A, Z](\lambda) \le p(\lambda).$$

Note 6.1. All messages sent to Z will contain 1^λ; therefore, Z will always have at least time polynomial in λ to perform its computations. □

Note 6.2. Our notion of a well-behaved environment essentially plays the same role as the notion of an a priori poly-time environment in [HUMQ09]. Roughly speaking, an environment Z is said to be *a priori poly-time* if $\mathrm{Time}_Z[\Pi, A, Z](\lambda)$ is bounded by a fixed polynomial in λ with probability 1, for all Π, A, and λ. The problem with this definition is that without making very specialized assumptions about encodings and machine execution models, it is impossible to even construct an a priori poly-time Z, since any Z could easily be overwhelmed with a very long message that it could not process quickly enough. We prefer to use the above (slightly more complicated) definition, rather than rely on such specialized assumptions. □

Before we present the definition of a poly-time protocol, recall that A_{d} denotes the dummy adversary (see §4.7).

Definition 2 (poly-time protocol). *A protocol Π rooted at r is called **(multi-)poly-time** if there exists a polynomial p such that for every well-behaved environment Z that is (multi-)rooted at r, we have*

$$\Pr\Big[\mathrm{Time}_\Pi[\Pi, A_{\mathrm{d}}, Z](\lambda) > p\big(\,\mathrm{Flow}_{Z\to\Pi,A_{\mathrm{d}}}[\Pi, A_{\mathrm{d}}, Z](\lambda)\,\big)\Big] = \mathrm{negl}(\lambda).$$

Here, $\mathrm{negl}(\lambda)$ denotes an anonymous negligible function, i.e., one that tends to zero faster than the inverse of any polynomial in λ.

In words, the last line of the definition says that when we consider the execution of the system $[\Pi, A_{\mathrm{d}}, Z]$, if f is the flow out of Z, the running time t of Π must satisfy $t \le p(f)$ with all but negligible probability. An essential feature of this definition is that the polynomial p does not depend on Z. Also note that unless specified otherwise, whenever we speak of flow, either from the environment to the adversary, or from the adversary to the protocol, we mean the flow of *uninvited* messages only.

Note that as we have defined it, the values t and f are measured at the *end* of the system execution, and the condition $t \le p(f)$ is required to hold at that time. We could consider a

seemingly stronger definition, where we measure the values t and f after each activation of a machine, and insist that $t \leq p(f)$ holds after each activation, with all but negligible probability. However, it turns out that this definition, which we could call **continuously (multi-)poly-time**, is actually no stronger:

Theorem 1 ((multi-)poly-time $\implies$ continuous (multi-)poly-time). *If Π is a (multi-)poly-time protocol (with polynomial bound p), then it is also continuously (multi-)poly-time (with the same polynomial bound p).*

Proof. The proof is a standard "guessing argument", where we guess the first epoch in which the time bound is violated.

Suppose Π is protocol rooted at r that is (multi-)poly-time with polynomial bound p, so that the relation $t_Z \leq p(f_Z)$ holds at the end of the execution of $[\Pi, A_{\mathrm{d}}, Z]$, for all Z and with all but negligible probability. Here, t_Z represents the execution time of Π and f_Z represents the flow out of Z in the execution of $[\Pi, A_{\mathrm{d}}, Z]$. More precisely, this means that for all well-behaved Z and all $c > 0$, there exists $\lambda_0 > 0$ such that the probability that $t_Z > p(f_Z)$ at the *end* of execution of $[\Pi, A_{\mathrm{d}}, Z]$ on external input $\mathtt{1}^\lambda$ is less than $1/\lambda^c$ for all $\lambda > \lambda_0$.

Now suppose, by way of contradiction, that there is a well-behaved Z' such that with non-negligible probability, $t_{Z'} > p(f_{Z'})$ at some point during the execution of $[\Pi, A_{\mathrm{d}}, Z']$. This means that for this Z' there is a $c > 0$ and an infinite set Λ such that for all $\lambda \in \Lambda$, the probability that $t_{Z'} > p(f_{Z'})$ at some point during the execution of $[\Pi, A_{\mathrm{d}}, Z']$ on external input $\mathtt{1}^\lambda$ is at least $1/\lambda^c$.

Using Z', we construct a new environment Z. To motivate the construction of Z, we recall the discussion of the dynamic call graph in §5.3. As discussed there, we may divide the execution of $[\Pi, A_{\mathrm{d}}, Z']$ into epochs. At the beginning of an epoch, as control departs from Z', the flow $f_{Z'}$ can only increase, but during the remainder of the epoch, $f_{Z'}$ remains constant while the time $t_{Z'}$ only increases. Therefore, if $t_{Z'} > p(f_{Z'})$ holds at any point in time during the execution of $[\Pi, A_{\mathrm{d}}, Z']$, it will hold at the end of some epoch. Because Z' is assumed well-behaved, the number of epochs is bounded by $q(\lambda)$ for some polynomial q — this follows from the fact that the number of epochs is bounded by the total flow out of Z (including invited messages), which is bounded by a polynomial in λ. We can number the epochs $1, \ldots, q(\lambda)$. Let j^* be the random variable that denotes the index of the first epoch in which $t_{Z'} > p(f_{Z'})$ in the execution of $[\Pi, A_{\mathrm{d}}, Z']$, defining $j_0 := 0$ if this event never occurs.

So Z works the same as Z', except that it makes a random guess j_0 at j^*, and halts at the end of epoch j_0. As a minor technical matter, since we want Z to be well-behaved, and our model of computation gives Z access to random bits, Z may not be able to sample j_0 from the uniform distribution on $\{1, \ldots, q(\lambda)\}$. So instead, in generating j_0, we round up the bound $q(\lambda)$ to the next power of 2.

The reader may easily verify that Z is well-behaved, and that for all $\lambda \in \Lambda$, the probability that $t_Z > p(f_Z)$ at the *end* of the execution of $[\Pi, A_{\mathrm{d}}, Z]$ on external input $\mathtt{1}^\lambda$ is at least $1/(2\lambda^c q(\lambda))$, which is the contradiction we sought. □

Clearly, if Π is multi-poly-time, then it is, in particular, poly-time. The following theorem establishes the converse:

Theorem 2 (poly-time $\implies$ multi-poly-time). *Every poly-time protocol is multi-poly-time.*

Proof. This is also a standard "guessing argument", where we guess which instance first violates a corresponding time bound.

Assume Π is a poly-time protocol rooted at r. We want to show that Π is multi-poly-time.

By assumption, there exists a polynomial p such that for all well-behaved Z rooted at r, with all but negligible probability, we have $t_Z \leq p(f_Z)$. Here, t_Z represents the running time of Π and f_Z represents the flow out of Z in the execution of $[\Pi, A_{\mathrm{d}}, Z]$.

Without loss of generality, we may assume that p takes only non-negative values and is *super-additive*, meaning that $p(a+b) \geq p(a) + p(b)$ for all non-negative integers a and b. This is because $t_Z = 0$ whenever $f_Z = 0$, which means that we may take p to be of the form $p(X) = c_0 X^{c_1}$ for positive constants c_0 and c_1. Such a p is non-negative and super-additive. Note that any non-negative and super-additive function is also non-decreasing.

By Theorem 1, we may assume that with all but negligible probability, the bound $t_Z \leq p(f_Z)$ holds continuously throughout the execution of $[\Pi, A_{\mathrm{d}}, Z]$. We claim that the same holds, even if Z is multi-rooted at r.

So suppose that Z' is a well-behaved environment, but is multi-rooted at r, and further suppose, by way of contradiction, that the claim does not hold for this Z'. This means that for this Z' there is an infinite set Λ and a $c > 0$ such that for all $\lambda \in \Lambda$, with probability at least $1/\lambda^c$, we have $t_{Z'} > p(f_{Z'})$ at some time during the execution of $[\Pi, A_{\mathrm{d}}, Z']$.

To finish the proof of the theorem, we will show how to use Z' to construct a well-behaved environment Z that is (singly) rooted at r, such that for some polynomial q, and for all $\lambda \in \Lambda$, with probability at least $1/q(\lambda)$, we have $t_Z > p(f_Z)$ at some time during the execution of $[\Pi, A_{\mathrm{d}}, Z]$. This will contradict the assumption that Π is poly-time.

To motivate the construction of Z, we recall the discussion in §5.3 on the dynamic call graph associated with the execution of $[\Pi, A_{\mathrm{d}}, Z']$. The number of instances of Π created during the execution of $[\Pi, A_{\mathrm{d}}, Z']$ is at most $q_{\mathrm{i}}(\lambda)$ for some polynomial q_{i} — this follows from the fact that the number of instances is bounded by the flow from Z into Π, which is in turn bounded by a fixed polynomial in λ. At the beginning of each epoch, control passes from Z to an instance of Π, either directly or via A_{d}; during the epoch, control stays within that instance; at the end of the epoch, control passes back to Z, either directly or via A_{d}.

Let us number the instances $1, 2, \ldots, q_{\mathrm{i}}(\lambda)$, in the order in which the first machine in the instance is created. Also, for $i = 1, \ldots, q_{\mathrm{i}}(\lambda)$, write $f^{(i)}_{Z'}$ for the flow out of Z that is bound for the ith instance (either directly or via A_{d}), and $t^{(i)}_{Z'}$ for the total running time of the machines in the ith instance. More precisely, to determine $f^{(i)}_{Z'}$, we count the messages sent from Z' directly to protocol machines in the ith instance, and the *uninvited* messages sent to A_{d} that are syntactically well-formed instructions to send a message to a machine that has previously sent a message to A_{d} and that belongs to the ith instance. By definition, we have $t_{Z'} = \sum_i t^{(i)}_{Z'}$. We also have $f_{Z'} \geq \sum_i f^{(i)}_{Z'}$ — note the inequality: some uninvited messages from Z' to A_{d} may not be forwarded to any instance of Π (in particular, a message that is not a syntactically well formed instruction, or an instruction to send a message to a machine that has not previously sent a message to A_{d}). By super-additivity,

$$p(f_{Z'}) \geq p\Big(\sum_i f^{(i)}_{Z'}\Big) \geq \sum_i p\big(f^{(i)}_{Z'}\big).$$

Thus, if $t_{Z'} > p(f_{Z'})$, we must have $t^{(i)}_{Z'} > p(f^{(i)}_{Z'})$ for some $i = 1, \ldots, q_{\mathrm{i}}(\lambda)$. Let i^* denote the index i of the instance for which $t^{(i)}_{Z'} > p(f^{(i)}_{Z'})$ for the *first* time during the execution of $[\Pi, A_{\mathrm{d}}, Z']$.

The machine Z will work as follows. It will make a guess i_0 at the index i^*. Z will internally simulate Z', but will only invoke machines in the i_0th instance invoked by Z'. The machines in other instances it will internally simulate, but with a "clamp" placed on their running times: for each instance $i \neq i_0$, Z will keep track of the values $t^{(i)}_{Z'}$ and $f^{(i)}_{Z'}$, and as soon as $t^{(i)}_{Z'} > p(f^{(i)}_{Z'})$ holds, Z will immediately terminate the execution of the system. When Z' attempts to send a

message directly to a protocol machine, Z forwards the message, either to an internally simulated machine or to an external one, as appropriate. In addition, when Z' attempts to send a message to a protocol machine via A_{d}, Z does the following: if such a machine (internally simulated or external) has previously sent a message to the adversary (which is something that Z can keep track of), then Z forwards the message to that machine (via a dummy adversary), and control eventually returns to Z (and then Z'); otherwise, Z returns the message $\langle\, \texttt{error} \,\rangle$ to Z'. Note that when sending a message to a protocol machine via A_{d}, Z can easily determine the instance to which the machine belongs by inspecting the SID associated with that machine: the SID can be truncated at the right-most component with protocol name r, and then compared to the SIDs of the existing instances. Finally, as in the proof of Theorem 1, in generating i_0, Z rounds up the bound $q_{\mathrm{i}}(\lambda)$ to the next power of 2.

By design, Z is a well-behaved environment rooted at r. Indeed, it is obvious that the flow out of Z is bounded by the flow out of Z', and hence bounded by a fixed polynomial in λ. One also has to verify that Z is itself multi-activation polynomial-time. This follows fairly easily from the fact that the internally simulated protocol machines of Z are clamped, and the fact that Z' is itself multi-activation polynomial-time; in particular, one observes that within a single activation of Z, the total length of the messages sent to Z' from simulated protocol machines of Z is bounded by a fixed polynomial in λ; moreover, the amount of time spent simulating clamped protocol machines is bounded by a fixed polynomial in λ.

The execution of $[\Pi, A_{\mathrm{d}}, Z]$ perfectly mimics that of $[\Pi, A_{\mathrm{d}}, Z']$, at least up until the point that Z forces the execution to a halt. Moreover, whenever $t_{Z'}$ would have exceeded $p(f_{Z'})$, with probability at least $1/(2q_{\mathrm{i}}(\lambda))$, Z will guess i^* correctly, which will lead to $t_Z > p(f_Z)$. Setting $q(\lambda) := 2\lambda^c q_{\mathrm{i}}(\lambda)$, this implies $t_Z > p(f_Z)$ with probability $1/q(\lambda)$ for all $\lambda \in \Lambda$. This is the contradiction we sought. □

7 Protocol emulation

In this section, we state the basic definitions and theorems related to protocol emulation. Intuitively, when we say that a protocol Π_1 emulates another protocol Π, we mean that Π_1 is "no less secure" than Π, in the sense that anything an adversary can achieve attacking Π_1 can be achieved by attacking Π.

To begin with, we introduce constraints on the types of adversaries we will consider in this context (as characterized in Definition 5 below).

Definition 3 (time-bounded adversary). *Let Π be a (multi-)poly-time protocol rooted at r. Let A be an adversary. We say that A is **(multi-)time-bounded for** Π if the following holds: there exists a polynomial p such that for every well-behaved environment Z that is (multi-)rooted at r, we have*

$$\Pr\Big[\mathrm{Time}_{\Pi,A}[\Pi, A, Z](\lambda) > p\big(\,\mathrm{Flow}_{Z\to\Pi,A}[\Pi, A, Z](\lambda)\,\big)\Big] = \mathrm{negl}(\lambda).$$

In words, the last line of the definition says that when we consider the execution of the system $[\Pi, A, Z]$, if f is the flow out of Z, the running time t of Π and A together must satisfy $t \le p(f)$ at the end of the execution with all but negligible probability. If this property holds, then just as in Theorem 1, we may assume that the bound $t \le p(f)$ holds continuously throughout the execution of $[\Pi, A, Z]$ with all but negligible probability. We omit the details: the statement and proof are nearly identical to that in Theorem 1. Again, unless specified otherwise, whenever we speak of flow, either from the environment to the adversary, or the adversary to the protocol, we mean the flow of *uninvited* messages only.

Definition 4 (flow-bounded adversary). *Let* Π *be a (multi-)poly-time protocol rooted at* r*. Let* A *be an adversary. We say that* A *is* ***(multi-)flow-bounded for*** Π *if the following holds: there exists a polynomial* p *such that for every well-behaved environment* Z *that is (multi-)rooted at* r*, we have*

$$\Pr\Big[\mathrm{Flow}_{A\to\Pi}[\Pi,A,Z](\lambda) > p\big(\,\mathrm{Flow}_{Z\to A}[\Pi,A,Z](\lambda)\,\big)\Big] = \mathrm{negl}(\lambda).$$

In words, the last line of the definition says that when we consider the execution of the system $[\Pi, A, Z]$, if f_{ea} is the flow from Z into A, then the flow f_{ap} from A into Π must satisfy $f_{\mathrm{ap}} \le p(f_{\mathrm{ea}})$ at the end of the execution with all but negligible probability. Again, if this property holds, we may assume that the bound $f_{\mathrm{ap}} \le p(f_{\mathrm{ea}})$ holds continuously throughout the execution of $[\Pi, A, Z]$ with all but negligible probability.

Definition 5 (bounded adversary). *Let* Π *be a (multi-)poly-time protocol. Let* A *be an adversary. We say that* A *is* ***(multi-)bounded for*** Π *if it is (multi-)time bounded for* Π *and (multi-)flow bounded for* Π*.*

A simple, but important, fact is the following:

Theorem 3 (dummy adversary is bounded). *Let* Π *be a (multi-)poly-time protocol. Then the dummy adversary* A_{d} *is (multi-)bounded for* Π*.*

Proof. This follows almost immediately from the definitions, as the reader may verify. □

Indeed, the converse of Theorem 3 is also easy to prove: if A_{d} is (multi-)bounded for Π, then Π is (multi-)poly-time.

We next state a simple theorem about the overall running times of structured systems, assuming all components satisfy their respective constraints.

Theorem 4 (overall poly-time). *Suppose* Π *is a (multi-)poly-time protocol rooted at* r*,* A *is an adversary that is (multi-)bounded for* Π*, and* Z *is a well-behaved environment that is (multi-)rooted at* r*. Then there exists a polynomial* p *such that for all* λ*, the total running time of all machines in the execution of the structured system* $[\Pi, A, Z]$ *on external input* $\mathtt{1}^\lambda$ *is bounded by* $p(\lambda)$ *with probability* $1 - \mathrm{negl}(\lambda)$*.*

Proof. By the well-behaved property for Z, the number of activations of Z and the flow out of Z is bounded by a fixed polynomial in λ with probability 1. Therefore, by the assumption that A is time-bounded for Π, the running time of all the machines in Π and A is bounded by a fixed polynomial in λ with all but negligible probability. Assuming the latter event holds, it follows that the flow into Z per activation is bounded by a fixed polynomial in λ, and the fact that Z is multi-activation polynomial-time implies the overall running time of Z is polynomial in λ. □

We are almost ready to present the central definition of our framework. However, we must first establish just a bit more (mostly standard) notation and terminology.

We write $\mathrm{Exec}[\Pi, A, Z](\lambda)$ to denote that random variable representing the output of the structured system $[\Pi, A, Z]$ on input $\mathtt{1}^\lambda$. If the process terminates, $\mathrm{Exec}[\Pi, A, Z](\lambda)$ is a string over $\mathbf{\Sigma}$; otherwise, we write $\mathrm{Exec}[\Pi, A, Z](\lambda) = \bot$.

Each value of λ defines a different random variable $\mathrm{Exec}[\Pi, A, Z](\lambda)$, and we may consider the family of random variables $\{\mathrm{Exec}[\Pi, A, Z](\lambda)\}_{\lambda=1}^{\infty}$. We write $\mathrm{Exec}[\Pi, A, Z]$ to denote this family.

We will need the standard notion of indistinguishability between two such families of random variables. To this end, let $\{X_\lambda\}_\lambda$ be a family of random variables, where each random variable

takes values in the set $\mathbf{\Sigma}^* \cup \{\bot\}$. Let D be a probabilistic polynomial-time program that takes as input a string over $\mathbf{\Sigma}$, and outputs 0 or 1. We call such a program a **distinguisher**. For each λ, we can define the random variable $D(X_\lambda)$ by the following randomized process: sample a value x according to the distribution of X_λ; if $x = \bot$, then $D(X_\lambda) := \bot$; otherwise, $D(X_\lambda)$ is defined to be the output of D on input $\langle\, 1^\lambda, x \,\rangle$.

Definition 6 (Computationally indistinguishable). *Let $X := \{X_\lambda\}_\lambda$ and $Y := \{Y_\lambda\}_\lambda$ be two families of random variables, where each random variable takes values in the set $\mathbf{\Sigma}^* \cup \{\bot\}$. We say X and Y are **computationally indistinguishable** if for every distinguisher D, we have $|\Pr[D(X_\lambda) = 1] - \Pr[D(Y_\lambda) = 1]| = \mathrm{negl}(\lambda)$. If this holds, we write $X \approx Y$.*

Here is the central definition of the framework.

Definition 7 (emulation). *Let Π and Π_1 be (multi-)poly-time protocols rooted at r. We say that Π_1 **(multi-)emulates** Π if the following holds: for every adversary A_1 that is (multi-)bounded for Π_1, there exists an adversary A that is (multi-)bounded for Π, such that for every well-behaved environment Z that is (multi-)rooted at r, we have*

$$\mathrm{Exec}[\Pi, A, Z] \approx \mathrm{Exec}[\Pi_1, A_1, Z].$$

We now state the *four central theorems* of the framework. The first two are somewhat technical theorems, which are used to prove the third theorem, which is the *main theorem* of the framework.

Theorem 5 (completeness of the dummy adversary). *Let Π and Π_1 be (multi-)poly-time protocols rooted at r. Suppose that there exists an adversary A that is (multi-)bounded for Π, such that for every well-behaved environment Z (multi-)rooted at r, we have $\mathrm{Exec}[\Pi, A, Z] \approx \mathrm{Exec}[\Pi_1, A_\mathrm{d}, Z]$. Then Π_1 (multi-)emulates Π.*

The proof is in §7.2.

Theorem 6 (emulates $\implies$ multi-emulates). *Let Π and Π_1 be poly-time protocols. If Π_1 emulates Π, then Π_1 multi-emulates Π.*

The proof is in §7.3 Recall that from Theorem 2, if a protocol is poly-time, then it is also multi-poly-time, so the statement of Theorem 6 makes sense. Because of these properties, we ignore multi-emulation in the remaining two theorems.

The next of the central theorems is the main theorem of the framework:

Theorem 7 (composition theorem). *Suppose Π is a poly-time protocol rooted at r. Suppose Π' is a poly-time subprotocol of Π rooted at x. Finally, suppose Π'_1 is a poly-time protocol also rooted at x that emulates Π' and that is substitutable for Π' in Π. Then $\Pi_1 := \Pi[\Pi'/\Pi'_1]$ is poly-time and emulates Π.*

The proof is in §7.4.

The final central theorem is simple but useful:

Theorem 8 (reflexivity and transitivity of emulation). *Let Π, Π_1, and Π_2 be poly-time protocols. Then Π emulates Π. In addition, if Π_2 emulates Π_1 and Π_1 emulates Π, then Π_2 emulates Π.*

Proof. The reflexivity property, that Π emulates Π, is obvious. For the transitivity property, suppose Π_2 emulates Π_1 and Π_1 emulates Π. We want to show that Π_2 emulates Π. Let A_2 be an adversary that is bounded for Π_2. Since Π_2 emulates Π_1, there exists an adversary A_1

that is bounded for Π_1 such that $\text{Exec}[\Pi_1, A_1, Z] \approx \text{Exec}[\Pi_2, A_2, Z]$ for all well-behaved Z. Since Π_1 emulates Π, there exists an adversary A that is bounded for Π such that $\text{Exec}[\Pi, A, Z] \approx \text{Exec}[\Pi_1, A_1, Z]$ for all well-behaved Z. The theorem then follows by the transitivity of the $\approx$ relation. □

7.1 Discussion

Note 7.1. Typically, in Definition 7, Π will be some kind of simple, idealized protocol which represents a security specification for some particular type of task, while Π_1 is a more complicated, but more concrete protocol that is meant to implement this specification. Indeed, Π will typically specify just a single program that implements an "ideal functionality", so that a single instance of Π will essentially consists of just a single ideal machine. See §8.2 for more details.

Also, the adversary A in the definition is often called a **simulator**. The task of any security proof is to design such a simulator. Because of Theorem 5, one can, if convenient, assume that A_1 is the dummy adversary. □

Note 7.2. A key feature of Theorem 7 is that we do not need to assume a priori that Π_1 is poly-time. This meshes well with the way we expect this theorem to be used in the practice of protocol design. Typically, Π' is an ideal functionality and Π'_1 is an implementation of this ideal functionality that may be quite complex. Naturally, Π'_1 was designed and analyzed in isolation, and proved to be poly-time and to emulate Π', without regard to any application for which Π' might be used. Similarly, Π was designed and analyzed in isolation, without regard to how Π' might be implemented. As a protocol designer, we do not have to additionally prove that Π_1 is poly-time — that is a conclusion of the theorem, and not an additional hypothesis. This leads to a high degree of modularity in protocol design and analysis.

The tradeoff in achieving this modularity is the introduction of the flow-boundedness constraint. Unfortunately, without this constraint, Theorem 7 is false, as demonstrated by the following example.

Let Π' be a one-party protocol that acts as a simple relay: it sends any message it receives from its environment to the adversary and sends any message it receives from the adversary to its environment. It is easy to see that Π' is poly-time.

Let Π'_1 be a protocol that works as follows: whenever it receives a message from its environment, it immediately "bounces" that message back to its environment. It is easy to see that Π'_1 is poly-time. Moreover, if we drop the flow-boundedness constraint, then Π'_1 emulates Π', via the following simulator A': whenever the simulator receives a message from Π', it immediately "bounces" that message to Π'. It is easy to see that A' is time-bounded for Π', and that $\text{Exec}[\Pi', A', Z] \approx \text{Exec}[\Pi'_1, A_\text{d}, Z]$ for all well-behaved Z. However, A' is *not* flow-bounded for Π' — A' sends messages to Π' without any provocation from Z.

We now define a one-party protocol Π that uses Π' as a subprotocol. After being activated by a message from its environment, Π sends that message to Π'; after this, whenever it receives any message from Π', it immediately "bounces" that message back to Π'. It is easy to see that Π is poly-time.

Now consider the corresponding protocol Π_1, which is the same as Π, but with Π'_1 substituted for Π'. It is easy to see that once activated by its environment, Π_1 runs forever, without any further activations of the environment. Indeed, when Π_1 is activated by a message m from its environment, Π_1 passes m down to Π'_1, which bounces it up to Π_1, which bounces it back down to Π'_1, which bounces it back up to Π_1, etc. In particular, Π_1 is not poly-time.

Thus, without the flow-boundedness constraint, poly-time is not preserved under composition.

It may be instructive to see where the proof of Theorem 7 fails in this case. The essence of the proof is to show that Π_1 emulates Π via a simulator A, so that $\text{Exec}[\Pi, A, Z] \approx \text{Exec}[\Pi_1, A_{\text{d}}, Z]$ for all well-behaved Z, and from this, argue that the poly-time property of Π_1 is inherited from the fact that A is time-bounded for Π. For this example, it suffices to consider environment Z which send an initial message to the protocol, but otherwise does not interact with the protocol or the adversary. In this case, the simulator A constructed in the proof of Theorem 7 is essentially just A'.

But it is easy to see that the execution of the system $[\Pi, A, Z]$ never halts: after Π is initialized with a message m from Z, that message passes through Π' to A', then bounced back through Π' to Π, then bounced back through Π' to A', then bounced back through Π' to Π, etc. In particular, A is not time-bounded for Π. □

Note 7.3. In Definition 7, we only consider adversaries that are bounded for the protocol. There are two senses in which this might appear to be too restrictive.

First, perhaps there are interesting attacks that cannot be modeled by such a restricted adversary. We argue that this is not the case. Ultimately, the only attack we really care about is an attack on a fully instantiated and concrete protocol as deployed in the real world. Such a real-world attack can be modeled using the dummy adversary: all the logic of the attack can be absorbed into the environment. Thus, a real-world attack on a concrete, poly-time protocol Π_1 is an environment Z interacting with Π_1 via the dummy adversary A_{d}. Theorem 3 says that A_{d} is bounded for Π_1. Now, when we show that Π_1 emulates some more idealized protocol Π, we show that there exists an adversary A that is bounded for Π such that $\text{Exec}[\Pi, A, Z] \approx \text{Exec}[\Pi_1, A_{\text{d}}, Z]$ for all well-behaved Z. In proving such a theorem, we may consider other, intermediate protocols Π', along with corresponding adversaries A' that are bounded for Π', and piece together these protocols and adversaries, using (among other things) Theorems 5–8. But the point is, this chain of reasoning is grounded by the boundedness of the dummy adversary — all of the other adversaries that arise are really just elements of various "thought experiments", which do not correspond to any real-world objects.

The second sense in which the boundedness constraint may appear to be too restrictive is this: perhaps the boundedness constraint makes it overly difficult to design simulators in security proofs. Indeed, while the time-boundedness constraint should not cause any problems, the flow-boundedness constraint may not always be trivial to maintain. For example, one consequence of the flow-boundedness constraint is that if the adversary has received no messages from the environment, then the adversary may not send any messages to the protocol — indeed, we saw above in Note 7.2 an example of what happens when this is not the case.

The restrictive nature of the flow-boundedness constraint is significantly mitigated by the message-invitation mechanism; indeed, this was the main motivation for introducing this mechanism. Our only essential use of invitations here is in connection with corruptions (see §8.1) and the JUC theorem (see §9). It may seem that the generality of the invitation mechanism is overkill for this application, but it turns out that this level of generality is needed to make all the components of our model work well together.

In addition to using the invitation mechanism, flow-boundedness problems can usually be avoided by simply following a reasonable set of conventions in the design of ideal functionalities. See §12.1 for examples of this (and in particular, §12.1.4). With such conventions, for typical protocols, the flow-boundedness constraints will naturally be satisfied by the simulators in their security analysis without much effort at all.

In some exceptional cases, it may be necessary to make a small, but usually benign, modification to a protocol so that flow-boundedness constraints may be satisfied in its security analysis. An

example of this is seen in §12.1.6. □

Note 7.4. The invitation mechanism itself may seem somewhat artificial. Indeed, it is only present to facilitate the analysis of protocols. In fact, in a fully instantiated and concrete protocol is deployed in the real-world, all invitations may be left out. To see this, suppose that Π_1 is such a concrete protocol with invitations, and that Π_2 is the same protocol, but with all invitations filtered out. It is not hard to see that Π_2 is poly-time and emulates Π_1 (the simulator just filters out the invitations). Thus, if we prove that Π_1 emulates some idealized protocol Π, then by transitivity, we have Π_2 emulates Π. Of course, it is only a good ideal to leave out the invitations in this final step — a protocol stripped of its invitations may no longer be useful as a building block in other protocols. □

Note 7.5. We should stress that our notions of poly-time and emulation have nothing at all to do with the notion of *liveness*. Intuitively, liveness means that the protocol actually gets something useful done. How this intuition is made rigorous is ultimately application dependent. Typically, for a particular class of protocols for some application, one defines the notions of "useful messages" and "useful results", and liveness means that to the extent that the adversary delivers useful messages, the protocol should deliver useful results.

For example, consider a trivial protocol Π_1 that essentially does nothing: whenever it receives a message from either its caller or the adversary, a machine running Π_1 sends an arbitrary (but fixed) message to the adversary. The protocol Π_1 satisfies our definition of poly-time, and emulates many interesting idealized protocols. However, it will certainly not satisfy any reasonable notion of liveness.

Because the notion of liveness is ultimately application dependent, we do not make any attempt to formalize it in this work. □

7.2 Proof of Theorem 5 (completeness of the dummy adversary)

We shall refer to Fig. 3 for several diagrams that will assist in the proof of the theorem.

Assume Π and Π_1 are (multi-)poly-time protocols rooted at r. Further, assume that there exists an adversary A that is (multi-)bounded for r, with the following property: for every well-behaved environment Z (multi-)rooted at r, we have $\text{Exec}[\Pi, A, Z] \approx \text{Exec}[\Pi_1, A_{\text{d}}, Z]$. Figs. 3(a) and (b) illustrate the execution of the systems $[\Pi_1, A_{\text{d}}, Z]$ and $[\Pi, A, Z]$. The arrows represent flows of messages among the various machines. Note that the boxes labeled Π_1 and Π represent all the protocol machines that exist in the system.

Our goal is to show that Π_1 (multi-)emulates Π. What we have to show is the following: for every adversary A_1^* that is (multi-)bounded for Π_1, there exists an adversary A^* that is (multi-)bounded for Π, such that for every for every well-behaved environment Z (multi-)rooted at r, we have $\text{Exec}[\Pi, A^*, Z] \approx \text{Exec}[\Pi_1, A_1^*, Z]$.

So let A_1^* be given. For convenience, we may assume that A_1^* is internally structured as illustrated in Fig. 3(c): it consists of a sub-machine A_1 that communicates directly with an arbitrary environment Z, but communicates with Π_1 via the dummy adversary A_{d}. It is easily seen that any adversary can be converted to an adversary of this type, without changing its behavior in any non-trivial way. Of course, in our formal model, A_1^* is a single machine, which internally simulates the "virtual" machines A_1 and A_{d}.

Fig. 3(d) illustrates how the corresponding adversary A^* is constructed: it is the same as A_1^*, except that the sub-machine A_{d} is replaced by A.

We now want to show that A^* is (multi-)bounded for Π and that for every for every well-behaved environment Z (multi-)rooted at r, we have $\text{Exec}[\Pi, A^*, Z] \approx \text{Exec}[\Pi_1, A_1^*, Z]$.

To this end, let Z be an arbitrary well-behaved environment that is (multi-)rooted at r. Using Z and A_1, we construct a new environment $Z^{(0)}$, corresponding to the box outlined with dotted lines in Fig. 3(e). In that diagram, we have also defined quantities f_{ep}, f_{ea}, and f_{ap}: here, f_{ep} represents the amount of flow from Z into Π_1, f_{ea} represents the amount of flow from Z into A_1, and f_{ap} represents the amount of flow from A_1 into A_d (which is essentially the same as the amount of flow from A_{d} into Π_1). Note that f_{ea} and f_{ap} measure the flow of *uninvited* messages. Let us define $f_{\mathrm{e}} := f_{\mathrm{ep}} + f_{\mathrm{ea}}$. In addition, we also stipulate the following: if the external output produced by Z is β, then the external output produced by $Z^{(0)}$ is $\langle\, \texttt{normal}, \beta \,\rangle$.

Note that $Z^{(0)}$ may not be well-behaved. So we define a new environment $Z^{(1)}$ that is. To define $Z^{(1)}$, we recall that since A_1^* is (multi-)bounded for Π_1, and since Z is well-behaved, we know that there are polynomials p and q, such that with all but negligible probability, the running time of A_1 (and indeed, A_1 and Π_1 together) is continuously bounded by $p(f_{\mathrm{e}})$ and f_{ap} is continuously bounded by $q(f_{\mathrm{ea}})$. The environment $Z^{(1)}$ is defined by placing a "clamp" on the execution of $Z^{(0)}$: if ever the running time of A_1 would exceed $p(f_{\mathrm{e}})$ or the flow f_{ap} would exceed $q(f_{\mathrm{ea}})$, the environment $Z^{(1)}$ immediately halts and emits a "beep" — more precisely, it outputs $\langle\, \texttt{beep} \,\rangle$, which is distinct from any "normal" output $\langle\, \texttt{normal}, \beta \,\rangle$ that might be produced by $Z^{(0)}$.

By construction, and the assumption that Z is well-behaved, it is clear that $Z^{(1)}$ is also well-behaved. Because A_1^* is (multi-)bounded for Π_1, it is clear that $[\Pi_1, A_{\mathrm{d}}, Z^{(1)}]$ emits a beep with negligible probability. Moreover, since the systems $[\Pi_1, A_{\mathrm{d}}, Z^{(0)}]$ and $[\Pi_1, A_{\mathrm{d}}, Z^{(1)}]$ behave identically unless the latter emits a beep, we have

$$\mathrm{Exec}[\Pi_1, A_{\mathrm{d}}, Z^{(0)}] \approx \mathrm{Exec}[\Pi_1, A_{\mathrm{d}}, Z^{(1)}]. \tag{3}$$

We next consider the system $[\Pi, A, Z^{(1)}]$, which is illustrated in Fig. 3(f). Since $Z^{(1)}$ is well-behaved, by hypothesis, we have

$$\mathrm{Exec}[\Pi_1, A_{\mathrm{d}}, Z^{(1)}] \approx \mathrm{Exec}[\Pi, A, Z^{(1)}]. \tag{4}$$

In particular, the probability that $[\Pi, A, Z^{(1)}]$ emits a beep is negligibly close to the probability that $[\Pi_1, A_{\mathrm{d}}, Z^{(1)}]$ emits a beep, and so itself is negligible.

In the system $[\Pi, A, Z^{(1)}]$, we write f'_{ap} to denote the flow from A into Π. Again, this counts the flow of *uninvited* messages. By the assumption that A is (multi-)bounded for Π, and since $Z^{(1)}$ is well-behaved, it follows that there exist polynomials p' and q', such that with all but negligible probability, the running time of Π and A together is continuously bounded by $p'(f_{\mathrm{ep}} + f_{\mathrm{ap}})$ and f'_{ap} is continuously bounded by $q'(f_{\mathrm{ap}})$. Moreover, in this system, we always have $f_{\mathrm{ap}} \leq q(f_{\mathrm{ea}})$. It follows that there exist polynomials p'' and q'', such that with all but negligible probability, the running time of Π and A together is continuously bounded by $p''(f_{\mathrm{e}})$ and the flow f'_{ap} is continuously bounded by $q''(f_{\mathrm{ea}})$.

Finally, we consider the system $[\Pi, A, Z^{(0)}]$, effectively removing the clamp. Since the two systems $[\Pi, A, Z^{(0)}]$ and $[\Pi, A, Z^{(1)}]$ proceed identically unless the clamp in the latter triggers the beep, and since the latter happens with negligible probability, we have

$$\mathrm{Exec}[\Pi, A, Z^{(1)}] \approx \mathrm{Exec}[\Pi, A, Z^{(0)}]. \tag{5}$$

Moreover, in the system $[\Pi, A, Z^{(0)}]$, with all but negligible probability, the running time of Π and A together is bounded by $p''(f_{\mathrm{e}})$, the running time of A_1 is bounded by $p(f_{\mathrm{e}})$, and the flow f'_{ap} is bounded by $q''(f_{\mathrm{ea}})$ — this follows from the fact that the corresponding events occur with all but negligible probability in $[\Pi, A, Z^{(1)}]$, and the two systems $[\Pi, A, Z^{(0)}]$ and $[\Pi, A, Z^{(1)}]$ proceed identically unless a negligible event occurs. This establishes that A^* is (multi-)bounded for Π.

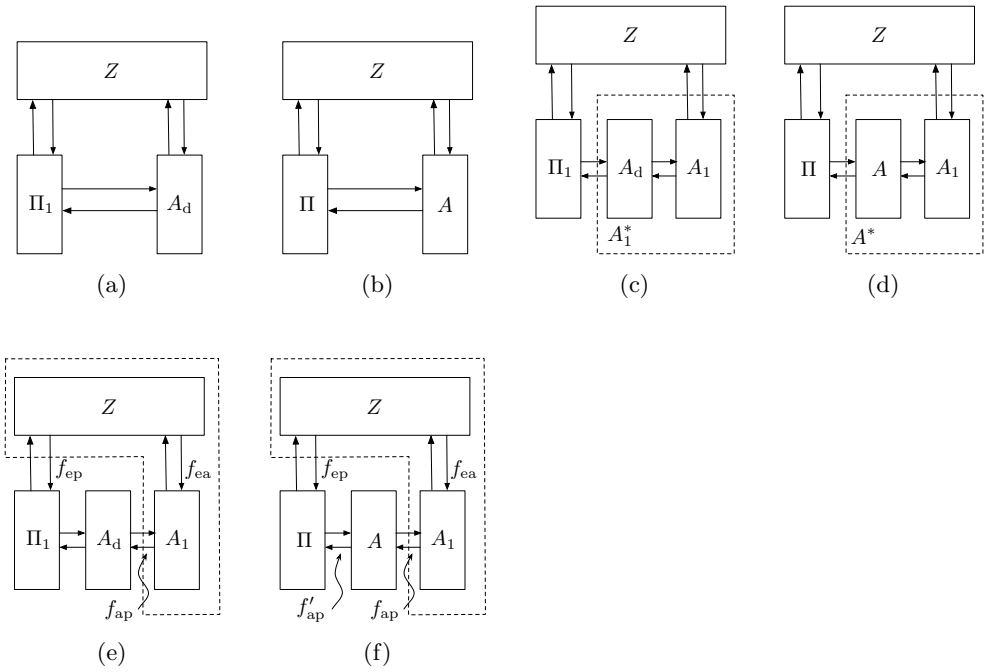


Figure 3: Diagrams for proof of Theorem 5

Finally, equations (3)–(5) imply that

$$\mathrm{Exec}[\Pi_1, A_\mathrm{d}, Z^{(0)}] \approx \mathrm{Exec}[\Pi, A, Z^{(0)}],$$

and this clearly implies that

$$\mathrm{Exec}[\Pi_1, A_1^*, Z] \approx \mathrm{Exec}[\Pi, A^*, Z],$$

which proves the theorem.

7.3 Proof of Theorem 6 (emulates $\implies$ multi-emulates)

Assume Π and Π_1 are poly-time protocols rooted at r. By Theorem 2, both Π and Π_1 are multi-poly-time protocols. Assume Π_1 emulates Π. In particular, there exists an adversary A that is bounded for Π, such that for every well-behaved environment Z that is rooted at r, we have

$$\mathrm{Exec}[\Pi, A, Z] \approx \mathrm{Exec}[\Pi_1, A_\mathrm{d}, Z].$$

We want to show that Π_1 *multi*-emulates Π. By the completeness of the dummy adversary (Theorem 5), it will suffice to construct an adversary A^* that is *multi*-bounded for Π, such that for every well-behaved environment Z that is *multi*-rooted at r, we have

$$\mathrm{Exec}[\Pi, A^*, Z] \approx \mathrm{Exec}[\Pi_1, A_\mathrm{d}, Z].$$

Before describing A^*, let us first make some preliminary observations. Consider an execution of the system $[\Pi, A, Z]$, where Z is (singly) rooted at r. We define f_ep to be the flow from Z into Π, f_ea to be the flow from Z into A, $f_\mathrm{e} := f_\mathrm{ep} + f_\mathrm{ea}$, and f_ap to be the flow from A into Π. We define t to be the running time of Π and A together. As usual, f_ea and f_ap measure the flow of *uninvited* messages.

By the assumption that A is bounded for Π, we know that there exist polynomials p and q, such that $t \leq p(f_{\mathrm{e}})$ and $f_{\mathrm{ap}} \leq q(f_{\mathrm{ea}})$ with all but negligible probability. Moreover, we may assume that these bounds hold continuously throughout the execution of $[\Pi, A, Z]$.

As in the proof of Theorem 2, we may assume that p is non-negative and super-additive. The essential fact used to establish this is that $t = 0$ whenever $f_{\mathrm{e}} = 0$. We may also assume that q is non-negative and super-additive. This requires a bit more of an argument, because we may have $f_{\mathrm{ap}} > 0$ even though $f_{\mathrm{ea}} = 0$. However, all messages sent between machines include the unary encoding of the security parameter λ. As λ tends to infinity, the length of any message sent from A to Π will eventually exceed $q(0)$. Therefore, with all but negligible probability, we have $f_{\mathrm{ap}} = 0$ whenever $f_{\mathrm{ea}} = 0$. Because of this, we may assume q is non-negative and super-additive.

Here is how A^* works. In the design of A^*, one should bear in mind that we expect the environment to be multi-rooted at r, and the messages the environment sends to A^* are expected to be instructions to a dummy adversary interacting with Π_1, even though A^* is actually interacting with Π. During the execution of the system, A^* will keep track of the instances it knows about and their SIDs. Let us number the instances $1, 2, \ldots$, in order of their discovery by A^*, and denote by sid_i the SID of the ith instance. In fact, the information that A^* has available to it will be incomplete, and because of this, some instances that actually exist in the system may not be known to A^*; because of this, we will also be forced to allow A^* to "discover" instances that do not exist. For each known instance, A^* will run an internally simulated copy of A.

When A^* receives a message from a protocol machine, it examines the SID of that machine. By scanning the components of this SID from right to left, and finding the first component from the right that names program r, A^* can determine the SID of the instance of Π to which this machine belongs. If necessary, this instance is added to its list of known instances. If the message received originates from the ith known instance, we say that the SID **resolves** to sid_i in this case, and A^* forwards this message to its ith internal copy of A. This copy of A will generate a message addressed to either a protocol machine in instance i or to the environment (recall that an adversary will only send messages to protocol machines from which it has previously heard, and moreover, this copy of A will only receive messages from protocol machines in instance i). In any case, A^* forwards the message generated by this copy of A to its intended recipient. However, if the message is to be sent to the environment, A^* first applies a special *invitation filter*: if the message to the environment includes any invitations, A^* checks that these invitations are instructions to send messages to machines whose SIDs also resolve to sid_i; if this check fails, A^* delivers the message $\langle\, \texttt{error} \,\rangle$ to the environment; otherwise, it forwards the message, as usual.

When A^* receives a message from its environment, it parses the message as an instruction to the dummy adversary, which is of the form $\langle\, id, m \,\rangle$. If the message it receives is not of this form, A^* sends $\langle\, \texttt{error} \,\rangle$ back to its environment. Otherwise, A^* processes id as in the previous paragraph, resolving its SID to the SID of a known instance. This process may fail if id is sufficiently ill-formed, in which case $\langle\, \texttt{error} \,\rangle$ is passed back to the environment. This process may also lead to the discovery of a new known instance; however, it may or may not correspond to an actual, existing instance; indeed, the information directly available to A^* is insufficient to determine this. Assuming the SID resolves to sid_i, A^* forwards $\langle\, id, m \,\rangle$ to its ith internal copy of A. This copy of A will respond with a message addressed to a protocol machine in instance i from which it has previously heard or to the environment. In any case, A^* forwards the message to its intended recipient. However, as in the previous paragraph, the same *invitation filter* is applied to messages directed towards the environment.

That completes the description of A^*. Before going further, we give some intuition behind the invitation filter. Essentially, this prevents one copy of A from inviting the environment to deliver a message to a different copy of A. Of course, we do not expect this to happen with any more than

negligible probability, but it will simplify the argument to simply explicitly enforce it. Doing this allows us to measure the "global" uninvited flow from the environment to A^* as the sum of the "local" uninvited flows from the environment to each copy of A.

To analyze A^*, we fix a well-behaved environment Z that is multi-rooted at r. Z will remain fixed for the rest of the proof.

We now describe a machine Z_0. Z_0 will not interact with any other machines: it will receive an input $\mathtt{1}^\lambda$, and will produce an output in $\mathbf{\Sigma}^*$. We will write $\mathrm{Exec}[Z_0](\lambda)$ for the random variable representing the output of Z_0 on input $\mathtt{1}^\lambda$, and define $\mathrm{Exec}[Z_0] := \{\mathrm{Exec}[Z_0](\lambda)\}_\lambda$. Here is how Z_0 works. Z_0 will internally run a copy of Z, keeping track of various protocol instances and their associated SIDs. The instances are numbered $1, 2, \ldots$, and the SID of the ith instance is denoted sid_i. Whenever Z attempts to deliver a message to a protocol machine, that machine's SID determines a protocol instance. New protocol instances may be discovered in this way. In addition, when Z attempts to send a message to the adversary, Z_0 processes the message as an instruction to the dummy adversary, and attempts to resolve the embedded SID to the SID of a known instance, using the same resolution procedure as in the design of A^*. This may result in an error, in which case the message $\langle\, \mathtt{error} \,\rangle$ is returned to Z. It may also result in the discovery of a new instance — although such an instance will not be "real", this is done to make Z_0 behave as much as possible like $[\Pi, A^*, Z]$. For each instance i, Z_0 maintains a cluster of virtual machines, consisting of one copy of the adversary A and all the protocol machines of one instance of Π. When Z attempts to pass a message to a machine in the ith cluster, Z_0 forwards the message to the appropriate virtual machine in that cluster (either a virtual protocol machine or the copy of A). Control will pass among machines in the ith cluster, eventually returning to the environment, either via a virtual protocol machine or via a copy of A. In the first case, Z_0 forwards the message to Z, while in the second case, Z_0 applies the same invitation filter used in the design of A^* before forwarding the message to Z.

As discussed above, the invitation filter ensures that the copy of A in one cluster will not invite Z to send a message to the copy of A in a different cluster.

Observe that the number of protocol instances discovered by Z_0 is bounded by $q_{\mathrm{i}}(\lambda)$ for some polynomial q_{i}. This follows from the fact that the number of discovered instances is bounded by the flow from Z into Π, which is in turn bounded by a fixed polynomial in λ. Moreover, the polynomial q_{i} may be chosen in a manner that is independent of the protocol Π and the adversary A.

The execution of Z_0 proceeds in a manner that is essentially the same as that of $[\Pi, A^*, Z]$. The only difference is that Z_0 may discover some protocol instances earlier than does A^*, but this will have no substantive effect. In particular, $\mathrm{Exec}[Z_0] = \mathrm{Exec}[\Pi, A^*, Z]$. The key feature of Z_0, however, is its modular internal structure: it consists of an internal copy of Z, a number of independent clusters of virtual machines built with Π and A, and a simple "dispatch module" connecting Z to the clusters.

Now consider the execution of Z_0 on input $\mathtt{1}^\lambda$. For $i = 1, \ldots, q_{\mathrm{i}}(\lambda)$, we define $f_{\mathrm{ep}}^{(i)}$ to be the flow from Z into the ith instance of Π, $f_{\mathrm{ea}}^{(i)}$ to be the flow from Z into the ith copy of A, $f_{\mathrm{e}}^{(i)} := f_{\mathrm{ep}}^{(i)} + f_{\mathrm{ea}}^{(i)}$, and $f_{\mathrm{ap}}^{(i)}$ to be the flow from the ith copy of A into the ith instance of Π; we also define $t^{(i)}$ to be the combined running time of Π and A in the ith cluster — this is the running time that the actual machines would take, not the amount of time required to simulate them. Note that, as usual, the flows $f_{\mathrm{ea}}^{(i)}$ and $f_{\mathrm{ap}}^{(i)}$ count uninvited messages only.

Next, we define a "clamped" version of Z_0, which we call Z_0^{c}. Each cluster will locally keep track of its internal flows and running times: the ith cluster will keep track of $f_{\mathrm{ep}}^{(i)}$, $f_{\mathrm{ea}}^{(i)}$, $f_{\mathrm{e}}^{(i)}$, $f_{\mathrm{ap}}^{(i)}$, and $t^{(i)}$. If at any time during the execution of the ith cluster it should happen that $t^{(i)} > p(f_{\mathrm{e}}^{(i)})$ or $f_{\mathrm{ap}}^{(i)} > q(f_{\mathrm{ea}}^{(i)})$, the ith cluster immediately halts the entire execution of Z_0.

Because of the clamps, and because Z is well-behaved, we see that Z_0^{c} runs in time polynomial in λ. Using a standard "guessing argument", almost identical to the proof of Theorem 2, one can show that the clamps set in Z_0^{c} will actually halt the execution prematurely with negligible probability. From this, it follows that

$$\mathrm{Exec}[\Pi, A^*, Z] = \mathrm{Exec}[Z_0] \approx \mathrm{Exec}[Z_0^{\mathrm{c}}]. \tag{6}$$

It also follows that with all but negligible probability, in the execution of Z_0, we have

$$\sum_i t^{(i)} \leq \sum_i p(f_{\mathrm{e}}^{(i)}) \leq p\Bigg(\sum_i f_{\mathrm{e}}^{(i)}\Bigg)$$

and

$$\sum_i f_{\mathrm{ap}}^{(i)} \leq \sum_i q(f_{\mathrm{ea}}^{(i)}) \leq q\Bigg(\sum_i f_{\mathrm{ea}}^{(i)}\Bigg).$$

Here, we have used the assumed super-additivity of p and q. Because the execution of $[\Pi, A^*, Z]$ proceeds in essentially the same as that of Z_0, we conclude that A^* is multi-bounded for Π.

It remains to show that $\mathrm{Exec}[\Pi, A^*, Z] \approx \mathrm{Exec}[\Pi_1, A_{\mathrm{d}}, Z]$. To this end, we define machines Z_1 and Z_1^{c}: these machines work just like Z_0 and Z_0^{c}, except that we use Π_1 and A_{d} in place of Π and A, and in defining Z_1^{c}, we use a time clamp appropriate to Π_1 (recalling that Π_1 is poly-time). As before, because of the clamps, Z_1^{c} runs in time polynomial in λ. Observe that in the execution of Z_1, the invitation filter will *never* trigger an error, and so execution of Z_1 proceeds in essentially the same way as the execution of $\mathrm{Exec}[\Pi_1, A_{\mathrm{d}}, Z]$. Thus, analogous to the observations above, we have

$$\mathrm{Exec}[\Pi_1, A_{\mathrm{d}}, Z] = \mathrm{Exec}[Z_1] \approx \mathrm{Exec}[Z_1^{\mathrm{c}}]. \tag{7}$$

So it will suffice to show that $\mathrm{Exec}[Z_0^{\mathrm{c}}] \approx \mathrm{Exec}[Z_1^{\mathrm{c}}]$.

To this end, we define two machines, Z_0^* and Z_1^*. For $b = 0, 1$, machine Z_b^* works as follows: it chooses a random index $i_0 \in \{1, \ldots, q_{\mathrm{i}}(\lambda)\}$, and then works like machine Z_0^{c}, except that in clusters $i = i_0 + 1 - b, \ldots, q_{\mathrm{i}}(\lambda)$ it uses the clamped versions of Π_1 and A_{d}, in place of the clamped versions of Π and A. Note that in generating i_0, we assume that its distribution is statistically close to the uniform distribution on $\{1, \ldots, q_{\mathrm{i}}(\lambda)\}$. Because execution in all clusters is clamped, both Z_0^* and Z_1^* run in time polynomial in λ.

Both Z_0^* and Z_1^* use clamped versions of Π and A in clusters $1, \ldots, i_0 - 1$, and use clamped versions of Π_1 and A_{d} in clusters $i_0 + 1, \ldots, q_{\mathrm{i}}(\lambda)$. The only difference between Z_0^* and Z_1^* is that in instance i_0, Z_0^* uses a clamped version of Π and A, and Z_1^* uses a clamped version of Π_1 and A_{d}. Using a standard hybrid argument, we have

$$\mathrm{Exec}[Z_0^*] \approx \mathrm{Exec}[Z_1^*] \implies \mathrm{Exec}[Z_0^{\mathrm{c}}] \approx \mathrm{Exec}[Z_1^{\mathrm{c}}]. \tag{8}$$

Finally, we define an environment Z^*, that is (singly) rooted at r, as follows: it works like machine Z_0^* (and, indeed, Z_1^*), except that instead of using cluster i_0 to process instance i_0, it uses whatever protocol and adversary it happens to be interacting with. In other words, after choosing i_0 at random, Z^* follows the logic of Z, but processes instances $1, \ldots, i_0 - 1$ using clusters of virtual machines, each running clamped versions of Π and A, processes instances $i_0 + 1, \ldots, q_{\mathrm{i}}(\lambda)$ using clusters of virtual machines, each running clamped versions of Π_1 and A_{d}, and processes instance i_0 using its external protocol and adversary. The structure of Z^* is illustrated in Fig. 4, where it is shown interacting with an arbitrary external protocol $\widetilde{\Pi}$ and an arbitrary external adversary $\widetilde{A}$.

Because Z is well-behaved, and because all of its clusters run with clamps, Z^* is well-behaved. The execution of $[\Pi, A, Z^*]$ (resp., $[\Pi_1, A_{\mathrm{d}}, Z^*]$) proceeds essentially identically to that of Z_0^* (resp.,

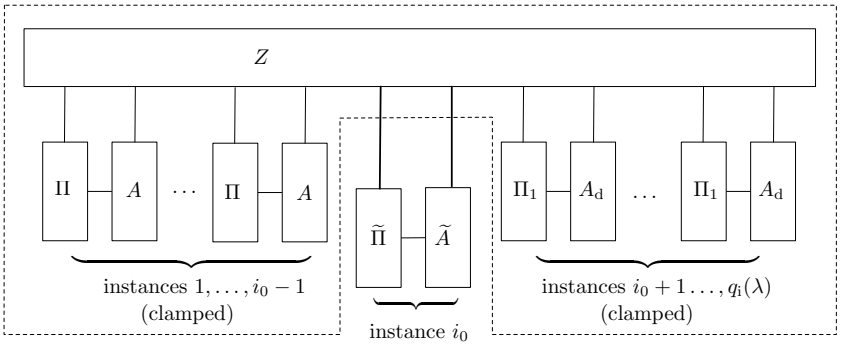


Figure 4: Structure of the environment Z^* in the proof of Theorem 6

Z_1^*), unless the constraints defining the boundedness of A for Π (resp., A_{d} for Π_1) are violated, which happens with negligible probability. It follows that

$$\mathrm{Exec}[\Pi, A, Z^*] \approx \mathrm{Exec}[Z_0^*] \quad \text{and} \quad \mathrm{Exec}[\Pi_1, A_{\mathrm{d}}, Z^*] \approx \mathrm{Exec}[Z_1^*]. \tag{9}$$

Now, using the defining property of A, and the fact that Z^* is a well-behaved environment rooted at r, we have

$$\mathrm{Exec}[\Pi, A, Z^*] \approx \mathrm{Exec}[\Pi_1, A_{\mathrm{d}}, Z^*]. \tag{10}$$

Combining (9) and (10), we obtain

$$\mathrm{Exec}[Z_0^*] \approx \mathrm{Exec}[Z_1^*].$$

Combining this with (8), we obtain

$$\mathrm{Exec}[Z_0^{\mathrm{c}}] \approx \mathrm{Exec}[Z_1^{\mathrm{c}}].$$

And combining this with (6) and (7), we obtain

$$\mathrm{Exec}[\Pi, A^*, Z] \approx \mathrm{Exec}[\Pi_1, A_{\mathrm{d}}, Z],$$

which proves the theorem.

7.4 Proof of Theorem 7 (composition theorem)

We are assuming that Π_1' emulates Π'. By Theorem 6, this implies that Π_1' multi-emulates Π'. This means that there exists an adversary A' that is multi-bounded for Π' such that for every well-behaved environment Z that is multi-rooted at x, we have $\mathrm{Exec}[\Pi', A', Z] \approx \mathrm{Exec}[\Pi_1', A_{\mathrm{d}}, Z]$.

We want to show that Π_1 emulates Π. We will in fact prove the stronger result that Π_1 multi-emulates Π. Strictly speaking, this is not necessary, in light of Theorem 6. However, we choose to do this for several reasons. First, the proof is really no more difficult. Second, by giving a direct proof of this stronger result, the resulting security reduction is more efficient than that obtained via Theorem 6. Third, we will later wish to prove a variant of this theorem in a slightly different setting where Theorem 6 will not be available to us, and so it will prove convenient to prove this stronger result now.

Because of Theorem 5, it suffices to exhibit an adversary A that is multi-bounded for Π, such that for every well-behaved Z that is multi-rooted at r, we have $\mathrm{Exec}[\Pi, A, Z] \approx \mathrm{Exec}[\Pi_1, A_{\mathrm{d}}, Z]$.

It will fall out as a natural by-product of the proof that Π_1 is itself poly-time. In fact, the proof will show directly that Π_1 is multi-poly-time, bypassing Theorem 2. However, we will use Theorem 2 to deduce from the hypotheses that Π and Π'_1 are multi-poly-time, and hence (by Theorem 3) that A_{d} is multi-bounded for Π'_1.

The first part of the proof is to describe our adversary A, which will interact with an environment Z and protocol machines belonging to instances of Π. One should think of Z as providing instructions to the dummy adversary to send messages to machines belonging to instances of the protocol Π_1 (and not Π). Our adversary A will internally maintain a copy of A'.

As a first step, let us introduce some helpful terminology. Given an SID *sid*, we shall classify it either *sub* or *super*. To do this, we scan the components of *sid* from right to left, until we reach one that specifies either r or x as a protocol name. If such a component exists and specifies x, we classify *sid* as *sub*; otherwise, we classify *sid* as *super*. Intuitively, a classification of *sub* means the SID apparently belongs to a machine in an instance of the subprotocol.

So here is how A works. Suppose A receives a message from a protocol machine. A first calculates the classification of this machine's SID. If the classification is *sub*, A forwards the message to its internal copy of A'. If the classification is *super*, A records the ID of the machine and forwards the message to Z, after performing the translations usually done by the dummy adversary.

Suppose A receives a message from Z. This message should be in the form of a dummy machine instruction (otherwise A just sends the usual error message $\langle\, \texttt{error} \,\rangle$ back to Z). So assume the message is $\langle\, id, m \,\rangle$, where $id = \langle\, pid, sid \,\rangle$. A calculates the classification of the SID *sid*. If the classification is *sub*, A forwards $\langle\, id, m \,\rangle$ to its internal copy of A'. If the classification is *super* and A has previously recorded *id* (as in the previous paragraph), A sends the message m to the machine with ID *id*. Otherwise, A sends the message $\langle\, \texttt{error} \,\rangle$ to Z.

Suppose A' generates a message, which is either addressed to a protocol machine or to the environment. If it is addressed to a protocol machine, A forwards the message to that machine — by Constraint C5 (see §4.4) and by the construction of A, this machine exists, belongs to an instance of the subprotocol, and has previously sent a message to A. If it is addressed to the environment, A will forward the message to Z, but subject to the following *invitation filter*: if the message to the environment includes any invitations, A checks that these invitations are instructions to send messages to machines whose SIDs are classified as *sub*; if this check fails, A delivers the message $\langle\, \texttt{error} \,\rangle$ to Z; otherwise, A forwards the message to Z.

That completes the description of A. The purpose of the invitation filter is essentially the same as in the proof of Theorem 6; in this case, we want to prevent A' from inviting Z to deliver a message to the superprotocol, rather than the subprotocol; this ensures that invitations to deliver a message to the superprotocol come from superprotocol, and invitations to deliver a message to the subprotocol come from A', keeping the measure of the flow of uninvited messages locally consistent. We also note that if the adversary A' is the one constructed in Theorem 6, the invitation filter here becomes redundant.

That completes the description of our adversary A. See Fig. 5(a) for a schematic view of how A interacts with protocol machines belonging to instances of Π and an environment Z. Here, all the protocol machines that belong to instances of the subprotocol Π' are grouped together in the box labeled Π', while all other protocol machines are grouped together in the box labeled $\Pi \setminus x$. Our adversary A is represented by the two boxes labeled "router" and A'.

- Messages from Z to A that represent instructions to send messages to machines in $\Pi_1 \setminus x$ (which is the same as $\Pi \setminus x$) are passed through the router to these machines, with just a bit of reformatting (translating dummy machine instructions to actual messages).

- Messages from Z to A that represent instructions to send messages to machines that appear to belong to instances of Π'_1 are passed directly through the router to the internal copy of A'.
- Messages from protocol machines in $\Pi \setminus x$ to A are passed through the router to Z, applying the usual transformations performed by the dummy adversary.
- Messages from the internal copy of A' to Z are passed through the router directly, subject to the invitation filter described above.
- Messages from machines in Π' to A are passed directly to the internal copy of A'.
- Messages from the internal copy of A' to machines in Π' are passed by A directly to those machines.

The arrows represent the flows of messages between machines grouped in the various boxes.

For the remainder of the proof, we let Z be an arbitrary, but fixed, well-behaved environment that is multi-rooted at r. Consider the diagram in Fig. 5(b). Here, we have labeled the diagram with various bounds that we anticipate to hold by virtue of the assumptions we are making. We will presently give a rigorous argument that these bounds hold continuously throughout the execution of the system, with all but negligible probability, but let us first describe the relevant notation in some detail.

- f_{ep} is the flow from Z into Π.
- f_{ea} is the flow from Z into A. Note that the flow from the router into A' is bounded by f_{ea} as well, while the flow from the router into $\Pi \setminus x$ is bounded by $h^*(f_{\mathrm{ea}})$ for some implementation-dependent polynomial h^* that bounds any flow expansion in translating dummy instructions to messages. As usual, this flow counts only uninvited messages.
- We anticipate that the flow from A' into Π' is bounded by $q_1(f_{\mathrm{ea}})$, where q_1 is the polynomial that is guaranteed to exist by virtue of the assumption that A' is flow-bounded for Π'. Again, this is the flow of uninvited messages.
- We anticipate that the total running time of protocol machines in $\Pi \setminus x$ and in Π' will be bounded by $t := p(f_{\mathrm{ep}} + f_{\mathrm{ea}} + h(q_1(f_{\mathrm{ea}})))$. Here, p is the polynomial that is guaranteed to exist by virtue of the assumption that Π is poly-time, and h is an implementation-dependent polynomial that bounds any flow expansion in translating from messages to dummy instructions. The reader should recall that in defining poly-time, the relevant quantity is the flow from Z into the dummy adversary, which we anticipate to be at most $f_{\mathrm{ea}} + h(q_1(f_{\mathrm{ea}}))$, rather than the flow from the dummy adversary into Π.
- We anticipate that the flow from $\Pi \setminus x$ into Π' will be bounded by $h'(t)$, where h' is an implementation-dependent polynomial. Indeed, if the running time of machines in $\Pi \setminus x$ is bounded by t, then the flow that they can generate and pass into the subprotocol Π' is bounded by some polynomial in t.
- We anticipate that the running time of A' will be bounded by $p_1(f_{\mathrm{ea}} + h'(t))$, where p_1 is the polynomial that is guaranteed to exist by virtue of the assumption that A' is time-bounded for Π'.

We assume that the polynomials h^*, h, h', p_1, q_1, and p are all non-negative and non-decreasing.

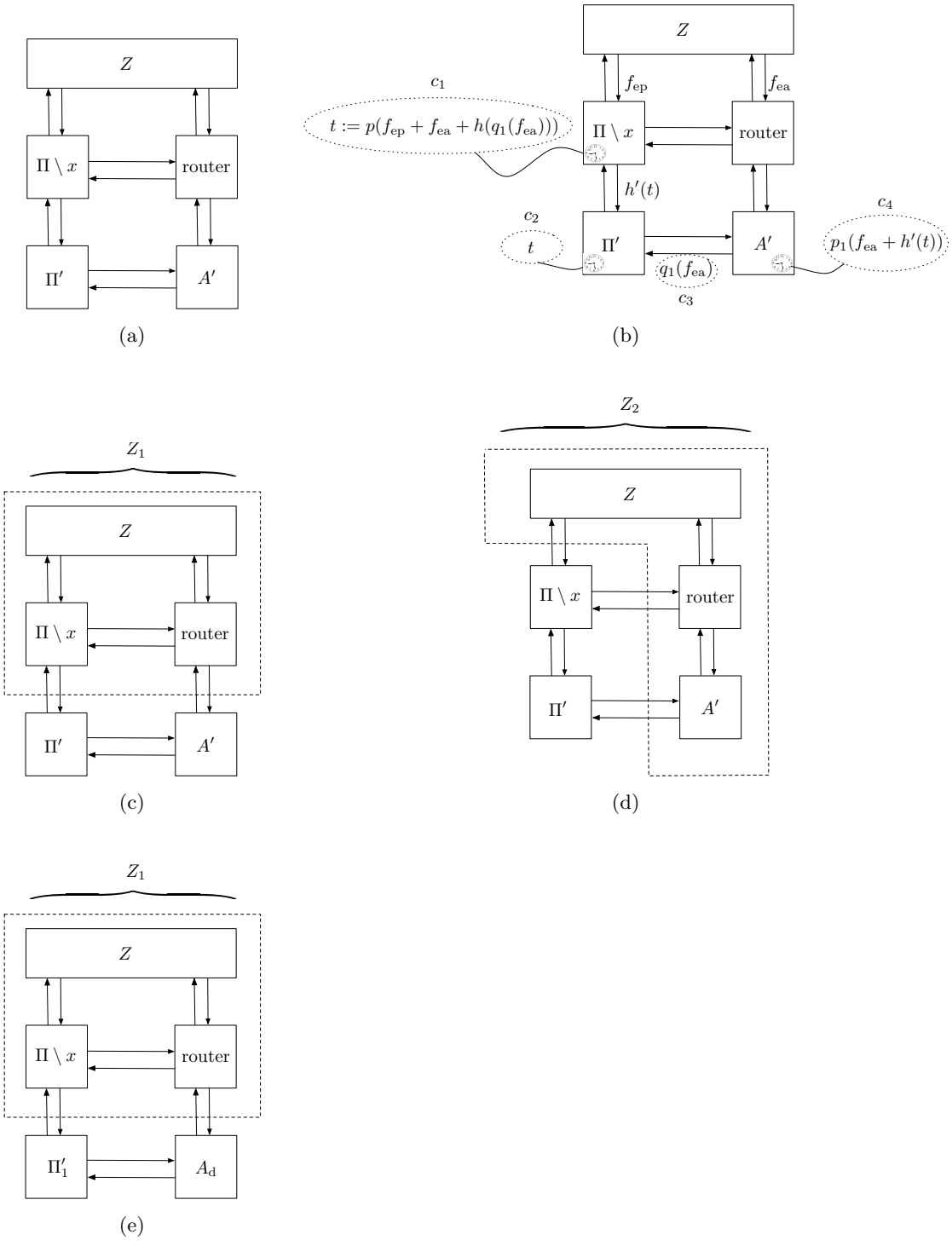


Figure 5: Diagrams for proof of Theorem 7

Our first goal is to show that the bounds in the dotted ovals in the diagram, labeled c_1, c_2, c_3, and c_4, hold continuously throughout the execution, with all but negligible probability. Here, c_1 says the running time of $\Pi \setminus x$ is continuously bounded by t, c_2 says the running time of Π' is continuously bounded by t, c_3 says that the flow from A' into Π' is continuously bounded by $q_1(f_{\text{ea}})$, and c_4 says the running time of A' is continuously bounded by $p_1(f_{\text{ea}} + h'(t))$.

To this end, consider a *single* machine M_0 that simulates the execution of the system, but with "clamps" placed that enforce the bounds c_1, c_2, c_3, and c_4. If any of these clamps are "triggered", meaning that the corresponding bound, which is continuously monitored by M_0, is violated, then M_0 immediately halts with an arbitrary output. It will suffice to show that during the execution of M_0, each of these clamps is triggered with only negligible probability.

We first argue that c_3 and c_4 are triggered with negligible probability. Consider the interaction of Π' and A' with an environment Z_1 that comprises Z, $\Pi \setminus x$, and the router, but with the clamp c_1 in place. This system is illustrated in Fig. 5(c), where Z_1 is outlined by a dashed box. The environment Z_1 simulates the execution of all the machines in $\Pi \setminus x$, monitoring the clamp c_1. It is straightforward to verify that the system $[\Pi', A', Z_1]$ satisfies all the constraints defined in §4 that any structured system must satisfy — this is entirely trivial, except perhaps Constraint C4 in §4.3 (the peer/ancestor free constraint), but even this is easily verified using the corresponding constraint for Z, and the fact that the static call graph of Π is acyclic. It is also worth noting here that because of the caller ID translation mechanism (see Constraint C8 in §4.6) the machines in Π' behave exactly the same, regardless of whether they are invoked by Z_1 or by machines in $\Pi \setminus x$. It is also easily verified that Z_1 is a well-behaved environment multi-rooted at x. The fact that is it multi-rooted at x is immediate from the construction. The fact that it is well-behaved follows from the fact that Z is well-behaved and that the simulated machines in $\Pi \setminus x$ are clamped. So now, we see that Z_1 is well-behaved, the flow from Z_1 into Π' is continuously bounded by $h'(t)$, and the flow from Z_1 into A' is continuously bounded by f_{ea}; therefore, the flow from A' into Π' is continuously bounded by $q_1(f_{\text{ea}})$ with all but negligible probability (because A' is multi-flow-bounded for Π'), and the running time of A' is continuously bounded by $p_1(f_{\text{ea}} + h'(t))$ with all but negligible probability (because A' is multi-time-bounded for Π'). Since the execution of the system $[\Pi', A', Z_1]$ proceeds identically to the execution of M_0 unless the clamps c_3 or c_4 are triggered, we conclude that with all but negligible probability, neither c_3 nor c_4 are triggered in M_0.

We next argue that c_1 and c_2 are triggered in M_0 with negligible probability. Consider the interaction of Π with an environment Z_2 that comprises Z, the router, and A', but with the clamps c_3 and c_4 in place. This system is illustrated in Fig. 5(d), where Z_2 is outlined by a dashed box. Also, note that technically speaking, the communication lines between $\Pi \setminus x$ and the router, and between Π' are A', actually go through a dummy adversary. Because of the clamp c_4, it is easy to verify that Z_2 is a well-behaved environment multi-rooted at r. Moreover, during the execution of $[\Pi, A_{\text{d}}, Z_2]$, the total flow out of Z_2 is continuously bounded by $f_{\text{ep}} + f_{\text{ea}} + h(q_1(f_{\text{ea}}))$ — the polynomial h is used to account for the translation from messages to dummy instructions. Therefore, because Π is assumed multi-poly-time, it follows that during the execution of $[\Pi, A_{\text{d}}, Z_2]$, the running time of Π, is continuously bounded by $t = p(f_{\text{ep}} + f_{\text{ea}} + h(q_1(f_{\text{ea}})))$ with all but negligible probability. Since the execution of the system $[\Pi, A_{\text{d}}, Z_2]$ proceeds identically to the execution of M_0 unless the clamps c_1 or c_2 are triggered, we conclude that with all but negligible probability, neither c_1 nor c_2 are triggered in M_0.

So now we have established that with all but negligible probability, none of the clamps in M_0 are triggered. As a first consequence of this fact, we derive the first major conclusion of the theorem:

A is multi-bounded for Π.

This follows from the fact that execution of $[\Pi, A, Z]$ proceeds identically to that of M_0 unless one

of the clamps in M_0 is triggered. Since none of these clamps are triggered, with all but negligible probability, we conclude that all of the corresponding bounds in the execution of $[\Pi, A, Z]$ hold continuously, with all but negligible probability. From this, it is clear that A is multi-bounded for Π.

Next, we observe that in the execution of $[\Pi', A', Z_1]$, the clamp c_1 is triggered with only negligible probability. Again, this follows from the fact that the execution of $[\Pi', A', Z_1]$ proceeds identically to that of M_0 unless one of the clamps in M_0 is triggered. Since none of these clamps are triggered, with all but negligible probability, we conclude that in the execution of $[\Pi', A', Z_1]$, the clamp c_1 is triggered with only negligible probability.

From the observation in the previous paragraph, we may conclude that

$$[\Pi, A, Z] \approx [\Pi', A', Z_1].$$

This follows from the fact that the executions of $[\Pi, A, Z]$ and $[\Pi', A', Z_1]$ proceed identically unless the clamp c_1 in Z_1 is triggered, and that happens with negligible probability.

Next, using the fact that Z_1 is a well-behaved environment multi-rooted at x, and the defining property of A', we conclude that

$$[\Pi', A', Z_1] \approx [\Pi'_1, A_{\mathrm{d}}, Z_1].$$

The system $[\Pi'_1, A_{\mathrm{d}}, Z_1]$ is illustrated in Fig. 5(e). Moreover, we observe that in the execution of $[\Pi'_1, A_{\mathrm{d}}, Z_1]$, the clamp c_1 in Z_1 is triggered with negligible probability; indeed, if this did not hold, we could easily build an environment that distinguished between Π'/A' and Π'_1/A_{d}.

Observe that $[\Pi'_1, A_{\mathrm{d}}, Z_1]$ and $[\Pi_1, A_{\mathrm{d}}, Z]$ proceed identically unless the clamp c_1 in Z_1 is triggered. In justifying this step, we are actually making use of the subroutine constraints (Constraint C9 in §4.6.3), as well as the substitutability hypothesis: together, these two conditions ensure that the programs simulated by Z_1 in $[\Pi'_1, A_{\mathrm{d}}, Z_1]$ are really the same as those in the execution of $[\Pi_1, A_{\mathrm{d}}, Z]$. Additionally, we are making use of the fact the invitation filter will *never* trigger an error in the execution of $[\Pi_1, A_{\mathrm{d}}, Z]$.

Now, since the clamp c_1 in Z_1 is triggered with negligible probability, it follows that

$$[\Pi'_1, A_{\mathrm{d}}, Z_1] \approx [\Pi_1, A_{\mathrm{d}}, Z].$$

So we have

$$[\Pi, A, Z] \approx [\Pi', A', Z_1] \approx [\Pi'_1, A_{\mathrm{d}}, Z_1] \approx [\Pi_1, A_{\mathrm{d}}, Z],$$

and we derive the second major conclusion of the theorem:

$$[\Pi, A, Z] \approx [\Pi_1, A_{\mathrm{d}}, Z].$$

Finally, observe that in the execution of $[\Pi'_1, A_{\mathrm{d}}, Z_1]$, the flow out of Z_1 is continuously bounded by $h'(t) + f_{\mathrm{ea}}$. Using the fact that Π'_1 is multi-poly-time, we conclude that in the execution of $[\Pi', A_{\mathrm{d}}, Z_1]$, the running time of Π'_1 is bounded $p'_1(h'(t) + f_{\mathrm{ea}})$, with all but negligible probability, for some fixed polynomial p'_1. Again, since $[\Pi'_1, A_{\mathrm{d}}, Z_1]$ and $[\Pi_1, A_{\mathrm{d}}, Z]$ proceed identically unless the clamp c_1 in Z_1 is triggered, and the latter happens with negligible probability, it follows that with all but negligible probability, in the execution of $[\Pi_1, A_{\mathrm{d}}, Z]$, the running time of Π_1 is bounded by $t + p'_1(h'(t) + f_{\mathrm{ea}})$. This proves the third, and final, major conclusion of the theorem:

Π_1 *is multi-poly-time.*

That completes the proof of the theorem.

8 Conventions regarding corruptions and ideal functionalities

8.1 Machine corruption

In analyzing protocols, one traditionally distinguishes between honest and corrupt parties. Most formal frameworks for multi-party computation include some mechanism for corrupting machines, and we do so as well. However, our mechanism is layered on top of the framework we have already defined, via certain message-passing conventions.

Consider a regular protocol machine. Such a machine starts out "honest", following the logic dictated by its program. The machine becomes "corrupt" when it receives a **special corrupt message** from its *caller*.

The exact format of the special corrupt message is unimportant, but some conventions should be established so that this message is not confused with other types of messages. For example, the special corrupt message could be $\langle \texttt{corrupt} \rangle$, and all other messages could be of the form $\langle \alpha, \ldots \rangle$, where $\alpha \neq \texttt{corrupt}$. We emphasize that to corrupt a regular protocol machine, this machine must receive this special corrupt message from its caller — receiving it from any other machine (a subroutine, its ideal peer, or the adversary) will not corrupt the machine.

When a regular protocol machine M is corrupted, here is what happens:

- In the activation in which M is corrupted, it sends a special message to the adversary that indicates that it has been corrupted; this message includes the state of M as it was at the beginning of the activation (to be clear, in the context of caller ID translation, as discussed in §4.6.1, this state information only includes the state of the inner core); this message also contains several invitations, which are described below. In addition, upon corruption, the internal state of M is updated to record the fact that the machine is corrupted; the state of M will never change again.

- In subsequent activations, M essentially becomes a "slave" to the adversary:

  - any messages received by M are forwarded to the adversary (along with the ID of the source);
  - the adversary may send "slave instructions" to M, telling it to deliver the special corrupt message to a subroutine of M, or an arbitrary message (possibly, but not necessarily, the special corrupt message) to M's ideal peer; the precise format of these "slave instructions" is not critical.

- The invitations that M sends to the adversary upon corruption are the "slave instructions" to send the special corrupt message to M's ideal peer and all *currently existing* subroutines of M.

Note 8.1. We do not make any conventions regarding ideal protocol machines. Such a machine may receive special corrupt messages from its peers, but how these are processed is completely up to the logic of its program. □

Note 8.2. The very first message that a regular machine receives (as an input from its caller) may be the special corrupt message; when this happens, the machine is created as usual, and the steps described above are immediately carried out. □

Note 8.3. Obviously, once a regular protocol machine M is corrupted, the adversary is free to corrupt any of M's subroutines: to see this, notice that after M is corrupted, the adversary may simply use M as a slave to send the special corrupt message to any of M's subroutines; because

the messages are sent through M, the caller of these subroutines, and not sent directly by the adversary, these special corrupt messages will have their intended effect.

Also, observe that the corruption instructions are treated as invited messages if they refer to subroutines that existed at the time of M's corruption. The adversary may also create new, corrupted subroutines, but the instructions to do so will be treated as uninvited. In this way, an adversary may construct a chain of corrupted machines in order to get one that "connects up" (i.e., is a peer of) some ideal machine, which may itself connect up to some honest machine. This ability seems like an essential requirement in any reasonable corruption model. □

Note 8.4. Note that the adversary cannot corrupt any machine that it wants to: considering the dynamic call graph, discussed in §5.3, the environment determines which of the root machines get corrupted; once a root machine is corrupted, the adversary is free to corrupt all the machines in the tree rooted by this machine, using only invited messages. □

Note 8.5. Treating the instructions to corrupt subroutines as invited messages should have no impact on the running-time analysis of a protocol: there was already enough uninvited flow to "pay" for the creation of this subroutine and the computation of its current state, and so there is also enough uninvited flow to "pay" for the corruption step. □

Note 8.6. One could consider a stronger corruption model, wherein after a regular protocol machine M is corrupted, the adversary is allowed to send *arbitrary* messages via M to M's subroutines. This is perhaps not unreasonable; however, it is not clear if this extra power is really useful in any meaningful way; moreover, with this extra power, it may be more difficult to prove that a protocol is poly-time. □

Note 8.7. Our conventions allow one to model the erasure or non-erasure model: it is really up to the logic of the program to determine what is stored in the state of the machine. In the non-erasure model, we would usually insist that the state include the entire history, including all incoming messages, and the results of all coin tosses. □

Note 8.8. As it is, our definitions model adaptive corruptions. Static corruptions can be effectively modeled by changing the conventions regarding corruptions in the following way: a regular protocol machine M will only recognize and obey the special corrupt message if M receives that message from its caller on its very first activation; otherwise, the special corrupt message is ignored, and an error message is sent to the adversary. □

Note 8.9. One could also model so-called "honest but curious" corruptions with a different set of corruption conventions. For example, when a regular protocol machine M is corrupted, its internal state is handed over to the adversary; subsequently, however, M follows its usual logic, except that it will respond to future requests from the adversary to reveal its current state, and that it will honor future requests from the adversary to send the special corrupt message to any of its subroutines. It is easy to arrange that all such special requests are invited messages, with appropriate invitations sent out in response to a corrupt or reveal request. □

Note 8.10. After a machine is corrupted, it still adheres to the constraints imposed in §4. Most of these are clearly reasonable, but the subroutine constraints in §4.6.3 may not seem to be at first sight. This constraint means that the adversary can only create subroutines of a corrupt machine whose protocol names are declared by the machine. Let us call this Constraint X. We shall argue that this constrain is, in fact, reasonable. As we have previously argued (see Note 7.3), the only attack we really care about is an attack on a fully instantiated and concrete protocol as deployed in the real world, carried out by an environment via the dummy adversary. Suppose that the deployed

protocol Π is a poly-time protocol. We need to make a further (quite reasonable) assumption: the ideal machines which may be created during an execution of Π are polynomial-time bounded in the following sense: for any such machine, with overwhelming probability, its total running time is bounded by a polynomial in the total uninvited flow coming into it.

Now, suppose we drop Constraint X. Recall Constraint C4 on the environment (see §4.3), that is, the peer/ancestor free constraint. Because of Constraint C4, any "bogus" machines that the adversary might create by means of flouting Constraint X would never "connect up" with any honest machine, either directly (with one machine being a subroutine of the other), or indirectly (via an ideal machine). Because of the polynomial-time assumption on ideal machines, it follows that all of the computations performed by the "bogus" machines and their ideal peers is polynomially bounded in the flow out of the environment. From this, it follows that Π is poly-time, even if we drop Constraint X; moreover, we see that the computations performed by the "bogus" machines and their ideal peers can be absorbed into the adversary, and this new adversary respects Constraint X (and is bounded for Π).

That completes the argument that Constraint X is reasonable. Of course, this argument hinged on Constraint C4, and one might ask if that is reasonable. Again, in an attack on a fully instantiated, concrete protocol, the instances of the protocol that are created by the environment would typically have SIDs with a very simple structure — there is no compelling reason for this to not be the case, as any non-trivial structure typically arises only from protocol composition, and this protocol is not intended to be used as a subprotocol. In particular, Constraint C4 should be trivially satisfied. □

8.2 Ideal protocols and ideal functionalities

While our formal model allows ideal protocol machines to be used in other ways, the standard (and, really, only) way we expect them to be used is to represent *ideal protocols*.

An **ideal protocol** is a special type of protocol. It defines a single program. When this program is run by a regular protocol machine, the logic of the program makes the machine a simple **dummy party**, which behaves as follows:

- It never invokes any subroutines, and its subroutine declaration is empty.
- All messages it receives from its caller are simply forwarded to its ideal peer, and are not recorded in its state.
- All messages it receives from its ideal peer are simply forwarded to its caller, and are not recorded in its state.
- All messages it receives from the adversary are deflected back to the adversary, and otherwise ignored.

These are the rules that apply to a dummy party when it is not corrupted; assuming the standard conventions for corruptions (see §8.1), the same rules apply to these machines as apply to any regular protocol machine:

- The dummy party becomes corrupted when it receives the special corrupt message from its caller, at which point it sends a special message to the adversary which includes its state, along with an invitation for an instruction to forward the special corrupt message via this dummy party to its ideal peer. The state information of the dummy party will contain no useful information.

- After the dummy party is corrupted, the adversary may send arbitrary messages via the dummy party to its ideal peer; these messages include the special (invited) corrupt message, but they may be quite arbitrary. In addition, any messages sent from the ideal peer to the dummy party are forwarded directly to the adversary. In this way, the adversary may interact with the ideal peer as if it were the dummy party, either with or without informing the ideal peer that the dummy party is corrupt (but see Note 8.12 below).

Because the logic of the regular protocol machines is fixed, the only interesting logic is in the ideal protocol machine itself. Therefore, in defining an ideal functionality, it suffices to define the logic of the ideal machine itself. The program describing thus logic is called an **ideal functionality**. We will sometime be a bit sloppy in the use of the terms "ideal functionality" and "ideal protocols". For example, we might say that an ideal functionality $\mathcal{F}$ is poly-time, when what we mean is the corresponding ideal protocol is poly-time; or we might say that a protocol Π emulates $\mathcal{F}$, when again, what we mean is that Π emulates the corresponding ideal protocol.

Besides ideal protocols, which define a program for which only the ideal machines perform any non-trivial tasks, it is also useful to define IM programs for which only the regular machines are relevant. Let us call an IM program **totally regular** if the behavior it defines for ideal machines is as follows: whenever it receives a message of any kind, it simply sends an error message to the adversary. For all practical purposes, such ideal machines can be safely ignored.

Although our formal model does not require it, typically, a protocol Π is a library defining a collection of totally regular programs, along with some ideal functionalities, $\mathcal{F}_1, \ldots, \mathcal{F}_k$, and no other programs. In this case, we say that Π is an $(\mathcal{F}_1, \ldots, \mathcal{F}_k)$-hybrid protocol; when $k = 1$, we just say that Π is an $\mathcal{F}$-hybrid protocol.

In §12.1, we discuss on number of typical ideal functionalities, and how they can be represented in our framework.

Note 8.11. Observe that even in the non-erasure model (see Note 8.7), dummy parties do not remember anything. This simplifies the definitions and simplifies reasoning about ideal protocols. Moreover, nothing is lost, as the ideal functionality can provide to the adversary whatever information is appropriate when the ideal functionality itself receives the special corrupt instruction. □

Note 8.12. For a typical ideal functionality $\mathcal{F}$ used by some protocol Π as a subroutine, in designing a simulator Π to show that Π emulates some other protocol, one can usually assume (without loss of generality) that when the adversary attacking Π corrupts a dummy party belonging to an instance of $\mathcal{F}$, it immediately notifies the ideal peer. Indeed, for a typical, well-designed ideal functionality, notifying the functionality that a dummy party is corrupt should only increase the rights and privileges of the adversary, and not diminish them in any way. If this assumption holds, it can simplify security proofs by reducing the number of cases that need to be considered. □

Note 8.13. Another observation that can simplify security proofs is the following. Typically, the goal is to design a simulator for an $(\mathcal{F}_1, \ldots, \mathcal{F}_k)$-hybrid protocol Π to show that Π emulates some other protocol, where Π defines just a single regular program π. Thus, in an execution of Π, the only subroutines of a machine M running π will be dummy parties belonging to one of the ideal functionalities. Without loss of generality, if an adversary attacking Π corrupts M, we may assume that it immediately corrupts all the extant dummy parties. This always holds, and for typical ideal functionalities, as discussed in the previous note, we may also assume that the adversary immediately notifies all the corresponding ideal functionalities as well. In this way, when

M is corrupted, we can assume that it gets "completely corrupted", which may simplify simulator design, because all of the intermediate corruption states can be ignored. □

9 Protocols with joint state

9.1 Motivation and goal

Consider a protocol Π that uses as a subprotocol an ideal functionality $\mathcal{F}$. Note that this means that even a single instance of Π can use multiple instances of $\mathcal{F}$. These subprotocol instances will be independent, and thus their potential implementations will also be. Specifically, it is not possible to implement several instances of an authentication functionality $\mathcal{F}$ using, say, the same signing key.

In this section, we will be seeking to combine *all* instances of $\mathcal{F}$ into a single instance of a suitable merged ideal functionality $\widehat{\mathcal{F}}$. This essentially only constitutes a rearranging of machines and messages, such that a single instance of $\widehat{\mathcal{F}}$ internally executes many instances of $\mathcal{F}$. However, now we can replace $\widehat{\mathcal{F}}$ with a secure implementation that uses a *joint state* (such as reused signing keys).

Of course, it has to be argued in the security proof for an implementation of $\widehat{\mathcal{F}}$ that the potential use of joint state does not affect the security of $\mathcal{F}$-instances. This cannot be argued generically, but depends on $\mathcal{F}$ and of course on its implementation. In this section, we provide the technical means to argue about joint state implementations.

In §12.2, we present a detailed example of how the results in this section can be used to realize secure channels using a PKI in our framework.

Our construction in this section is reminiscent of that in [CR03], although the details differ significantly.

9.2 Multi-session functionalities

To formalize the situation informally described above, let $\mathcal{F}$ be an ideal functionality with protocol name x. We define the **multi-session extension $\widehat{\mathcal{F}}$ of $\mathcal{F}$** as follows. Like $\mathcal{F}$, $\widehat{\mathcal{F}}$ is an ideal functionality. $\widehat{\mathcal{F}}$ thus defines only one program, which is named, say, $\widehat{x}$. Internally, $\widehat{\mathcal{F}}$ will simulate many instances of $\mathcal{F}$ (*without* the corresponding dummy parties of $\mathcal{F}$).

Concretely, $\widehat{\mathcal{F}}$ parses any incoming message m from a peer with ID $\langle\, pid, sid \,\rangle$ as $m = \langle\, vsid, m' \,\rangle$. $\widehat{\mathcal{F}}$ then forwards the "inner message" m' to an internal simulation of a machine $\mathcal{F}$ with ID $\langle \langle\, \texttt{ideal} \,\rangle, vsid \,\rangle$. This message is forwarded *as if* coming from a machine with ID $\langle\, pid, vsid \,\rangle$. If no such $\mathcal{F}$-instance exists, it is created. In this context, the SID *vsid* is called a **virtual SID**. Similarly, messages of the form $m = \langle\, vsid, m' \,\rangle$ from the adversary are forwarded (with *vsid* removed) to the internal $\mathcal{F}$-simulation with ID $\langle \langle\, \texttt{ideal} \,\rangle, vsid \,\rangle$.

Outgoing messages m' (either to the adversary or to some peer with PID *pid*) of a simulated machine with virtual SID *vsid* are encoded as $\langle\, vsid, m' \,\rangle$ and forwarded to the adversary, resp. $\widehat{\mathcal{F}}$'s own peer with PID *pid*. Intuitively, $\widehat{\mathcal{F}}$ thus internally simulates many instances of $\mathcal{F}$, with suitably prefixed incoming and outgoing communication.

To account for certain corner cases that can arise during the processing of inputs by $\widehat{\mathcal{F}}$, we clarify and highlight a few fine points of the above definition:

$\widehat{\mathcal{F}}$'s SID. An instance of $\widehat{\mathcal{F}}$ expects its session parameter (see §4.1) to be $\langle\, \rangle$, otherwise, all incoming messages are flagged as an error and bounced to the adversary.

Malformed messages. With the exception of special corruption messages (see below), incoming messages *not* of the form $m = \langle\, vsid, m' \,\rangle$ are flagged as an error and forwarded to the adversary. Furthermore, messages with **impossible** virtual SIDs *vsid* are forwarded to the adversary.

By a **possible** *vsid*, we mean that if the SID of this $\widehat{\mathcal{F}}$ is $\langle\, \alpha_1, \ldots, \alpha_{k-1}, \alpha_k \,\rangle$, then $vsid = \langle\, \alpha_1, \ldots, \alpha_{k-1}, \beta_1, \ldots, \beta_\ell \,\rangle$, where (i) $\beta_1, \ldots, \beta_\ell$ are syntactically valid basenames, (ii) β_ℓ specifies protocol name x, and (iii) none of $\beta_1, \ldots, \beta_{\ell-1}$ specify protocol name x. Of course, we say that *vsid* is *impossible* if it is not possible.

Corruption requests. Upon a special $\langle\, \texttt{corrupt} \,\rangle$ message from a dummy party of $\widehat{\mathcal{F}}$ with PID *pid* (which can only be sent once that dummy party is actually corrupted), $\widehat{\mathcal{F}}$ does the following. Suppose that $\widehat{\mathcal{F}}$ currently simulates k $\mathcal{F}$-instances with virtual SIDs $vsid_1, \ldots, vsid_k$ that have previously received a message whose apparent source PID is *pid*. Then, $\widehat{\mathcal{F}}$ invites the adversary to send messages $\langle\, \texttt{corrupt}, pid, vsid_i \,\rangle$ (for $i = 1, \ldots, k$) back to $\widehat{\mathcal{F}}$. Subsequently, upon receipt of *any* message of the form $\langle\, \texttt{corrupt}, pid, vsid \,\rangle$, where *vsid* is possible, $\widehat{\mathcal{F}}$ sends a $\langle\, \texttt{corrupt} \,\rangle$ message to the corresponding $\mathcal{F}$-instance, as if coming from ID $\langle\, pid, vsid \,\rangle$.

These invitations will be crucial when considering an implementation of $\widehat{\mathcal{F}}$, in which corrupting one regular machine might require a simulator to corrupt many virtual $\mathcal{F}$-instances in $\widehat{\mathcal{F}}$. Without these invitations, it would be difficult, if not impossible, for a simulator to maintain the flow-boundedness constraint.

Virtual Constraint C5. We let $\widehat{\mathcal{F}}$ enforce a "virtual version" of constraint C5 (see §4.4). Namely, $\widehat{\mathcal{F}}$ forwards messages from the adversary to internal $\mathcal{F}$-instances only if the adversary has already received a message from that machine. If the adversary attempts to send a message to an $\mathcal{F}$-instance it has not heard from yet, $\widehat{\mathcal{F}}$ replies with an error message to the adversary.

Relayed invitations. Invitations from a simulated machine are translated into invitations from $\widehat{\mathcal{F}}$. Concretely, if some simulated $\mathcal{F}$-instance with virtual SID *vsid* invites the adversary to send $i_1, \ldots, i_k$, $\widehat{\mathcal{F}}$ will invite the adversary to send $\langle\, vsid, i_1 \,\rangle, \ldots, \langle\, vsid, i_k \,\rangle$.

This completes our specification of $\widehat{\mathcal{F}}$.

Lemma 1. *Suppose $\mathcal{F}$ is a poly-time ideal functionality. Then the multi-session extension $\widehat{\mathcal{F}}$ of $\mathcal{F}$ is poly-time as well.*

Proof. First, recall that if $\mathcal{F}$ is poly-time, then it is also *multi*-poly-time (Theorem 2). The theorem will then follow from this claim:

Claim. *Suppose Z is a well-behaved environment rooted at $\widehat{x}$. Then there exists an environment Z' that is well-behaved and multi-rooted at x, such that if*

- *$\hat{f}$ is the flow out of Z in $[\widehat{\mathcal{F}}, A_{\mathrm{d}}, Z]$,*
- *$\hat{t}$ is the running time of $\widehat{\mathcal{F}}$ in $[\widehat{\mathcal{F}}, A_{\mathrm{d}}, Z]$,*
- *f is the flow out of Z' in $[\mathcal{F}, A_{\mathrm{d}}, Z']$, and*
- *t is the running time of all the $\mathcal{F}$-instances in $[\mathcal{F}, A_{\mathrm{d}}, Z']$,*

then for fixed polynomials p_1 and p_2, we have $\hat{t} \leq p_1(t + \hat{f})$ and $f \leq p_2(\hat{f})$, with probability 1.

Note that by flow, we mean, as usual, uninvited flow. Also, in comparing random variables between the two experiments, we assume that all the machines in both experiments obtain their random bits from a common point $\omega \in \{0,1\}^\infty$ in the sample space. To prove the claim, we need to show how to construct Z' given Z. For the most part, Z and Z' are the same, except for some simple message translation.

We leave it to the reader to verify the details. However, one technical point that needs to be considered is that when Z sends the special corrupt message through a corrupted dummy party to $\widehat{\mathcal{F}}$, it receives invitations to send corresponding corrupt messages to certain $\mathcal{F}$-instances inside $\widehat{\mathcal{F}}$; the corresponding corrupt messages in the execution of $[\mathcal{F}, A_{\mathrm{d}}, Z']$ will not be invited. However, each of these $\mathcal{F}$-instances in $[\mathcal{F}, A_{\mathrm{d}}, Z']$ have already received an uninvited message from Z'. Thus, the extra uninvited corrupt messages that Z' needs to send will not increase the uninvited flow by a substantial amount, which allows one to establish the stated flow bound.

Another technical point to be considered is that in our formal model, each activation of $\widehat{\mathcal{F}}$, even in response to an invited message, will require the entire state of $\widehat{\mathcal{F}}$ to be read, and this state includes the state information of all the virtual $\mathcal{F}$-instances. Nevertheless, the stated running-time bound can be shown to hold, by first verifying that the *total* flow (invited and uninvited) out of Z is bounded by a polynomial in $t + \hat{f}$.

Finally, one needs to verify that the constructed Z' satisfies all the necessary constraints. This is mostly straightforward, but we remark that the definition of impossible virtual SID will guarantee that Z' satisfies Constraint C4 in §4.3.

To finish the proof of the lemma, first observe that we have $t \leq q(f)$ with overwhelming probability, for some polynomial q, by virtue of the fact that $\mathcal{F}$ is multi-poly-time. We may assume that q, p_1, and p_2 are non-negative and non-decreasing. Thus, with overwhelming probability, we have

$$\hat{t} \leq p_1(t + \hat{f}) \leq p_1(q(f) + \hat{f}) \leq p_1(q(p_2(\hat{f})) + \hat{f}),$$

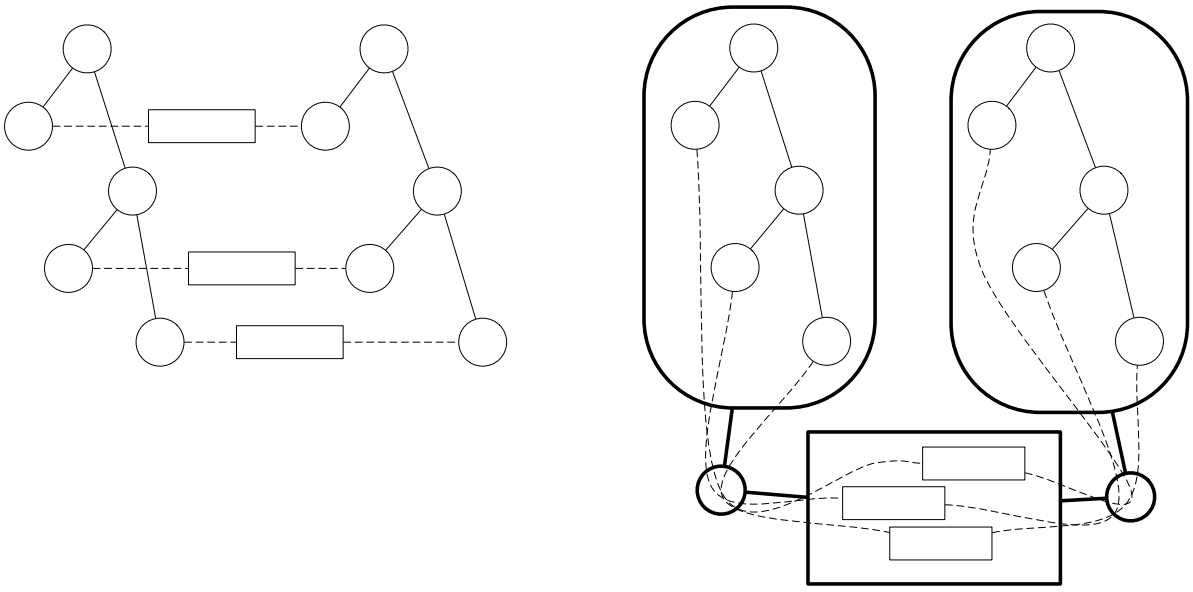
which proves the lemma. □

9.3 Boxed protocols

Observe that access to a multi-session functionality $\widehat{\mathcal{F}}$ as defined above is essentially the same as access to many instances of the single-session protocol $\mathcal{F}$. So now consider a larger protocol Π (rooted at r) that uses $\mathcal{F}$ (rooted at x) as a subprotocol. By our constraints, each instance of $\mathcal{F}$ has a unique caller. However, this may no longer hold once we subsume many $\mathcal{F}$-instances in a single $\widehat{\mathcal{F}}$-instance: each caller of *any* $\mathcal{F}$-instance is now a caller of $\widehat{\mathcal{F}}$.

To avoid a violation of our hierarchical protocol structure, we will thus have to put any protocol Π that uses $\widehat{\mathcal{F}}$ into a special form. Loosely speaking, we will let one single program internally execute virtual machines belonging to $\Pi \setminus x$, where calls to $\mathcal{F}$ are "trapped" and handed off to $\widehat{\mathcal{F}}$.

More formally, assume an $\mathcal{F}$-hybrid protocol Π (such that in particular, Π uses no ideal machines beyond $\mathcal{F}$) which is rooted at r. Then, the **boxed protocol** $[\Pi]_{\mathcal{F}}$ consists of only two programs, named r and $\widehat{x}$. Here, the program named $\widehat{x}$ is given by the multi-session extension $\widehat{\mathcal{F}}$ of $\mathcal{F}$; the program named r declares $\widehat{x}$ as its (only) subroutine and proceeds as follows. For clarity, we will call a machine running r in $[\Pi]_{\mathcal{F}}$ a $[\Pi \setminus x]$-machine. A $[\Pi \setminus x]$-machine internally simulates the execution of tree of regular machines belonging Π, including the dummy parties of $\mathcal{F}$. Concretely, any input directly passed to a $[\Pi \setminus x]$-machine by its caller is relayed to a simulation of Π's root program r. We call such simulated Π-machines **virtual machines**. When a virtual machine invokes a subroutine, a suitable virtual machine is created and simulated inside the $[\Pi \setminus x]$-machine. If one of these virtual machines represents a dummy party of $\mathcal{F}$, a message from that virtual dummy party to the corresponding instance of $\mathcal{F}$ is processed by sending a corresponding message to $\widehat{\mathcal{F}}$; the



(a) (b)

Figure 6: (a) unboxed dynamic call graph, (b) boxed dynamic call graph

corresponding message is prefixed with the SID of the virtual dummy party, and sent to $\widehat{\mathcal{F}}$ (via an actual dummy party that is the unique subroutine of the $[\Pi \setminus x]$-machine). Messages going from $\widehat{\mathcal{F}}$ to the $[\Pi \setminus x]$-machine (via this actual dummy party) are routed to the appropriate virtual machine inside the $[\Pi \setminus x]$-machine, after stripping the virtual SID. Similarly, communication between the $[\Pi \setminus x]$-machine and the adversary is dealt with by prefixing and stripping virtual SIDs, and routing messages to/from appropriate virtual machines inside the $[\Pi \setminus x]$-machine.

As with $\widehat{\mathcal{F}}$, illegally addressed incoming communication is bounced back to the adversary. Furthermore, each $[\Pi \setminus x]$-machine enforces a "virtual constraint C5": adversarial messages sent to internal virtual machines that have not yet spoken to the adversary are answered with an error message to the adversary. Finally, invitations from simulated machines within a $[\Pi \setminus x]$-machine are translated into corresponding invitations from the $[\Pi \setminus x]$-machine.

Intuitively, the boxed protocol $[\Pi]_{\mathcal{F}}$ is a compartmentalized version of protocol Π. In each $[\Pi \setminus x]$-machine, all computations corresponding to $\mathcal{F}$-calls are outsourced into one multi-session functionality $\widehat{\mathcal{F}}$. The remaining part of Π is executed in one $[\Pi \setminus x]$-machine. Fig. 6 illustrates the construction. In Fig. 6(a) we see a dynamic call graph of a single instance of a two-party protocol. The circles are regular machines, and the rectangles are ideal functionalities. In Fig. 6(b), we see the corresponding dynamic call graph for the boxed version of the protocol. For each party, all of its regular machines become virtual machines inside of a single $[\Pi \setminus x]$-machine (the two large ovals). Each such $[\Pi \setminus x]$-machine has a single subroutine, which is a dummy party of $\widehat{\mathcal{F}}$. In the figure, one sees the virtual instances of $\mathcal{F}$ inside the large rectangular box, which represents one instance of $\widehat{\mathcal{F}}$. The dotted lines in Fig. 6(b) now represent virtual communication lines between the virtual dummy parties inside the $[\Pi \setminus x]$-machines, and the corresponding virtual $\mathcal{F}$-instances inside $\widehat{\mathcal{F}}$.

9.4 The JUC theorem

We are now ready to present the main result of this section. In other frameworks commonly referred to as **joint state composition theorem** or simply **JUC theorem**.

Theorem 9 (JUC theorem). *Suppose $\mathcal{F}$ is a poly-time ideal functionality, and that Π is a poly-time $\mathcal{F}$-hybrid protocol. Then $[\Pi]_{\mathcal{F}}$ is poly-time and emulates Π.*

Proof. We focus on proving that $[\Pi]_{\mathcal{F}}$ emulates Π; the fact that $[\Pi]_{\mathcal{F}}$ is poly-time will fall out naturally from the proof. We assume that Π is rooted at r and $\mathcal{F}$ is rooted at x. By Theorem 5, it suffices to construct an adversary A that is bounded for Π such that

$$\mathrm{Exec}[\Pi, A, Z] \approx \mathrm{Exec}[[\Pi]_{\mathcal{F}}, A_{\mathrm{d}}, Z]. \tag{11}$$

for every well-behaved environment Z rooted at r.

We will construct A in two steps.

We begin by constructing an adversary A' such that

$$\mathrm{Exec}[[\Pi]_{\mathcal{F}}, A', Z] = \mathrm{Exec}[[\Pi]_{\mathcal{F}}, A_{\mathrm{d}}, Z] \tag{12}$$

for all well-behaved Z, so that the execution of $[[\Pi]_{\mathcal{F}}, A', Z]$ much more closely resembles an execution of Π.

Before we describe A', recall that communication between a $[\Pi \setminus x]$-machine and the $\widehat{\mathcal{F}}$ (via a dummy party) consists of messages $\langle\, \mathit{vsid}, m' \,\rangle$, where *vsid* denotes a virtual SID of a simulated $\mathcal{F}$-instance the simulation inside $\widehat{\mathcal{F}}$. We call such a *vsid* **Π-impossible** if it is impossible in the (more syntactic) sense defined above, or if it extends the SID of the $[\Pi \setminus x]$-machine in a way that is not allowed by Π's static call graph. Messages with Π-impossible *vsid*s have no counterpart in an execution of the unboxed protocol Π. Furthermore, Π-impossible *vsid*s can never occur in messages from an *uncorrupted* $[\Pi \setminus x]$-machine in $[\Pi]_{\mathcal{F}}$. However, as soon as a $[\Pi \setminus x]$-machine corrupted, the adversary may send messages to $\widehat{\mathcal{F}}$, apparently from the $[\Pi \setminus x]$-machine (actually, its dummy party subroutine) with Π-impossible *vsid*s. Unfortunately, since $\widehat{\mathcal{F}}$ does not know anything about its superprotocol Π, it can only check for impossibility (but not Π-impossibility) of virtual SIDs. We will hence need to take special care of Π-impossible virtual SIDs. The key observation is that such "impossible" $\mathcal{F}$-instances will never connect up with any uncorrupted machines, and so can simply be absorbed into the adversary.

In addition to dealing with these Π-impossible virtual SIDs, A' will clean up a couple of other issues as well. Specifically, A' does the following:

- Whenever Z sends a message with an impossible virtual SID, A' just responds to Z with an appropriate error message.
- Whenever Z sends a message with a Π-impossible (but not impossible) virtual SID (through a corrupted $[\Pi \setminus x]$-machine) to $\widehat{\mathcal{F}}$, A' processes this message itself, maintaining its own a internal simulated instance of $\mathcal{F}$. This is justified by the remark above that such "impossible" $\mathcal{F}$-instances will never connect up with any uncorrupted party.
- Normally, a $[\Pi \setminus x]$-machine has a unique subroutine, which itself is a dummy party that is a peer of an instance of $\widehat{\mathcal{F}}$, and this dummy party and $\widehat{\mathcal{F}}$-instance have a particular SID, uniquely determined by that of the $[\Pi \setminus x]$-machine. Once a machine $[\Pi \setminus x]$-machine is corrupted, Z may attempt to create other subroutines (e.g., with different SP parts of their SID) of the $[\Pi \setminus x]$-machine, which would be dummy parties of *other* instances of $\widehat{\mathcal{F}}$. These

other instances of $\widehat{\mathcal{F}}$ would have inappropriate SIDs, and so any interaction with them would simply result in an error message. Whenever this would happen, we just have A' generate the error message immediately, without actually creating any new instances of $\widehat{\mathcal{F}}$ in the first place.

- Whenever Z sends a message that would violate virtual Constraint C5, A' just sends back an appropriate error message to Z. Recall that Constraint C5 (see §4.4) says that the adversary can only send messages to machines from which it has previously heard. Protocol $[\Pi]_{\mathcal{F}}$ already enforces this constraint, but let us move this enforcement into A', so that $[\Pi]_{\mathcal{F}}$ is never even bothered with such messages.

It should be clear by construction that (12) holds. All we have done, essentially, is to move some computations out of $[\Pi]_{\mathcal{F}}$ into A'.

The next step is to construct an adversary A such that

$$\text{Exec}[\Pi, A, Z] = [[\Pi]_{\mathcal{F}}, A', Z] \tag{13}$$

for all well-behaved Z.

Adversary A will be interacting with an instance of Π and Z. It will internally run a copy of A', so that messages from Z are first sent to this internal copy of A'. Environment-bound messages from A' are sent to Z, and protocol bound messages from A' (who "thinks" he is talking to an instance of $[\Pi]_{\mathcal{F}}$) are processed by A.

Here is how A works:

SID translation and message passing. Generally, A will translate any message coming from a Π-machine with SID *sid* into a message with virtual SID *vsid* = *sid*, and pass it to A', as if coming from a corresponding $[\Pi \setminus x]$-machine, or from $\widehat{\mathcal{F}}$. For this, A will have to compute the correct "base SID" of the corresponding $[\Pi \setminus x]$-machine $\widehat{\mathcal{F}}$. However, since Π is rooted at r and its static call graph is acyclic, this base SID will have to be the longest prefix of *sid* that ends in r, in the case of a $[\Pi \setminus x]$-machine, or this prefix extended by a particular component, in the case of $\widehat{\mathcal{F}}$.

Conversely, A will translate messages from A' addressed to some $[\Pi \setminus x]$-machine or to $\widehat{\mathcal{F}}$ into the appropriate Π-messages. Because of the design of A', Constraint C5 will never be violated.

Corruption of Π. When a root machine of Π is corrupted (by receiving the special corrupt message from Z), A is informed of this, and obtains the internal state of that machine. A then corrupts all regular machines that are descendents of this machine in the dynamic call graph. This can all be done using invited messages (see Note 8.4). Using the information collected, A can construct a simulation of the current internal state of the corresponding $[\Pi \setminus x]$-machine of $[\Pi]_{\mathcal{F}}$, sending this to A' as a part of a notification that the $[\Pi \setminus x]$-machine is corrupted. This implicitly assumes that $[\Pi]_{\mathcal{F}}$ is designed in such a way that this simulation is feasible, which is not hard to do.

Instructions to corrupted $[\Pi \setminus x]$-machines. Once A' is informed that some $[\Pi \setminus x]$-machine is corrupted, it may corrupt the subroutine of this $[\Pi \setminus x]$-machine, which is a dummy party of $\widehat{\mathcal{F}}$. After this, A' may instruct this dummy party to send a corrupt message to $\widehat{\mathcal{F}}$. And after this, A' may instruct $\widehat{\mathcal{F}}$ to send arbitrary messages to corresponding virtual instances of $\mathcal{F}$ inside $\widehat{\mathcal{F}}$. By the design of A', these virtual $\mathcal{F}$-instances will not have Π-impossible virtual

SIDs. So A will be able to communicate with corresponding instances of $\mathcal{F}$ in the execution of Π, by generating a suitable *chain of corrupted machines*, if necessary, to connect up to that instance. Because the virtual SID is not Π-impossible, it will be compatible with the static call graph of Π, and so this is a legal operation of A.

In addition, in the design of $\widehat{\mathcal{F}}$, after a corruption, A' is invited to send certain corruption messages to $\widehat{\mathcal{F}}$. When A' sends these invited messages, A will have to send corresponding corruption messages to $\mathcal{F}$-instances in Π. However, A will already have received invitations for these. These observations are critical in proving that A is flow-bounded for Π.

By construction it is clear that (13) holds. From (12) and (13), it follows that (11) holds.

To complete the proof of the theorem, we have to argue that A is bounded for Π and that $[\Pi]_{\mathcal{F}}$ is poly-time.

Observe the computations performed by Π and A in $[\Pi, A, Z]$ are essentially the same as those performed by $[\Pi]_{\mathcal{F}}$ in $[[\Pi]_{\mathcal{F}}, A_{\mathrm{d}}, Z]$. Therefore, up to simulation overhead, the running times of these computations are the same. So it will suffice to show that A is time- and flow-bounded for Π.

Flow-boundedness should be clear from the construction, taking into account the remarks above in the description of processing instructions to corrupted $[\Pi \setminus x]$-machines. From flow-boundedness and the fact that Π is poly-time, it follows that the running time of Π in $[\Pi, A, Z]$ is polynomially bounded in the flow out of Z. To finish the proof, one has to take into account the time spent by A in $[\Pi, A, Z]$ simulating "impossible" $\mathcal{F}$-instances. However, because $\mathcal{F}$ is poly-time, and hence multi-poly-time (by Theorem 2), it is not hard to show (as in Lemma 1) that these simulations are polynomially bounded. □

10 An extension: common functionalities

We now present an extension to our framework, introducing the notion of *common functionalities*. This extension will involve revisions to some of the definitions and theorems in the previous sections.

The goal is to build into the framework the ability for protocol machines that are not necessarily peers, and also (potentially) for the environment, to have access to a "shared" or "common" functionality.

For example, such a common functionality may represent a *system parameter* generated by a trusted party that is accessible to all machines in the system (and, in practice, is just "hardwired" into their code). Our revisions will allow us to optionally restrict the environment's access to such a system parameter, allowing it to only access it indirectly via the adversary. With this restriction, a system parameter essentially becomes a *common reference string (CRS)* — the difference being that a simulator is allowed to "program" a CRS, but not a system parameter.

It is well known that without some kind of a *set up assumption*, such as a CRS, it is impossible to realize many interesting and important ideal functionalities [CF01]. Moreover, with a CRS, it is feasible to realize any "reasonable" ideal functionality [CLOS02] under reasonable computational assumptions.

One can model a CRS simply as an ideal functionality (as in [CF01, CLOS02]). However, if this is done, a different CRS is needed for every instance of a protocol that uses the CRS, which is unrealistic and impractical. One way around this problem is to use the JUC theorem (see §9). However, this is somewhat limiting and awkward (in particular, it may be difficult, if not impossible, to use such a CRS in the construction in [BCL$^+$05], which is discussed in §12.3).

Therefore, we have chosen to build the necessary mechanisms into the model, so that a CRS that is accessible by all protocol machines and by the adversary (but not by the environment).

While the simulator may still "program" the CRS (in the sense that it may invent a CRS for an environment), the composition of protocols becomes more intricate (see below). Moreover, the same mechanism we use to model CRSs allows us to model system parameters as well, with no extra complications.

In contrast to CRSs, system parameters are useful mainly to allow for more practical protocols in a variety of situations. Indeed, it may be convenient to assume that a trusted party generates an elliptic curve or RSA modulus once and for all, which may be used across many protocols. Such parameters may also be used in the definition of ideal functionalities. For example, the relation associated with a zero knowledge functionality may be parameterized in terms of such a system parameter — a CRS would really not make any sense here.

While our notion of common functionalities allows us to model system parameters and CRSs, as well as some other useful types of common functionalities (such as non-programmable and programmable random oracles), it is by no means completely general — it is not nearly as general as the Generalized UC framework [CDPW07]. Simply put: given the complexities of designing a consistent framework, we have opted for a limited and conservative extension to our basic framework. Designing a more elaborate extension is perhaps a project for the future.

10.1 Changes to §4 (structured systems)

We begin with some changes to the definitions in §4. In addition to the three basic classes of machines (environment, adversary, and protocol), we introduce a fourth class: **common functionality**.

Syntactically, a common functionality is distinguished by its machine ID, which is of the form $\langle$ com-*name* $\rangle$, where *name* is an arbitrary string. The name map of a structured system will map such a machine ID to the program name com-*name*. Such a program name is called a **common functionality name** (in contrast to the program names of protocol machines, which we call *protocol names*). It follows from this syntax that in any execution of a structured system, there will be only one machine executing a common functionality with a given name.

We now introduce a rule that will essentially make common functionalities behave as servers that are oblivious to the identity of their client. Namely, we shall require that the program of a common functionality is structured as a sandbox and an inner core, as follows: when activated on input $\langle\, id, state, id_0, msg_0 \,\rangle$, the sandbox passes the string $\langle\, id, state, msg_0 \,\rangle$ to the inner core (stripping the ID of the client); when the inner core outputs a string $\langle\, state', msg_1 \,\rangle$, the sandbox outputs $\langle\, state', id_0, msg_1 \,\rangle$ (which will direct msg_1 back to the client).

Constraints C1 and C2 do not change. In particular, common functionalities cannot send invitations.

Constraint C3 is changed to allow the environment to send messages to common functionalities (which may or may not be defined by the library).

Constraint C5 is changed to allow the adversary to send messages to common functionalities, without restriction.

Constraint C6 is changed to allow ideal protocol machines to send messages to common functionalities, subject to some restrictions (see below Constraint C9 below).

Constraint C8 is changed to allow regular protocol machines to send messages to common functionalities, subject to some restrictions (see below Constraint C9 below).

Constraint C9 is modified as follows. Similar to the subroutine declarations, we require that the program for a protocol machine declare the common functionalities that it is allowed to use. Both ideal and regular machines running the program will allow the machine to access only those common functionalities that are declared by the program.

To differentiate it from the notion of a structured system, defined in §4, we will call this new type of system an **extended structured system**. For emphasis, structured systems as defined in §4, may be called **basic structured systems**.

10.2 Changes to §5 (protocols)

The defining properties P1–P4 of a protocol (§5.1) are modified as follows:

- P1 is modified to allow common functionality names, in addition to protocol names.
- P2 is modified to include those constraints that apply to common functionalities.
- P3 is modified to require that all declared common functionalities are defined by the library.
- The definition static call graph is modified so that the set of nodes includes the common functionality names defined by the library, and so that an edge in the graph is included for each declaration of a common functionality. Thus, there will be edges from protocol names to common functionality names.

  P4 is changed to read as follows: *the static call graph is acyclic and has a unique node of in-degree 0, and that node is a protocol name.*

  This implies that all common functionalities defined in the library are actually declared by some program in the library. It also implies that the *root* of a protocol is a protocol name (not a common functionality name).

The definitions of subprotocols and substitutability (§5.2) do not change at all.

The definitions regarding protocol execution (§5.3) do not change at all. However, we note that in the discussion of the dynamic call graph, we do not consider common functionalities to be nodes in this graph. We do not consider common functionalities to belong to any protocol instance. Also, note that during an epoch, when Z is activated, control may pass back and forth between Z and various common functionalities, before passing from Z to a protocol machine or A.

The definition of the dummy adversary (§4.7) needs to be changed: when the dummy adversary receives a message $\langle\, id, m \,\rangle$ from the environment, if id is of the form $\langle\, comName \,\rangle$ for some common functionality name $comName$, the dummy adversary forwards this message m to the indicated common functionality (regardless of whether the library even defines it). The response from the common functionality will be processed by the dummy adversary as usual (although the response could be an "error message" from the "master machine", rather than from the common functionality, if the latter is not defined).

10.3 Changes to §6 (resource bounds)

The definitions measuring time and flow in §6 are modified as follows. We modify the definition of $\text{Flow}^*_Z[\Pi, A, Z](\lambda)$ so as to include the flow from Z into any common functionalities. None of the other definitions measuring time and flow change *at all*; in particular, they do *not* count the running time of, or the flows into or out of, any common functionalities. These are dealt with separately.

The definition of a *well-behaved environment* (see Definition 1) stays precisely as is.

In the definition of a *poly-time protocol* (see Definition 2), the only additional requirement is that each common functionality defined by Π is *multi-activation polynomial-time* and *I/O-bounded* (see §3.3).

Theorems 1 and 2 remain valid, precisely as stated. The proofs are almost identical, as we leave to the reader to verify.

10.4 Changes to §7 (protocol emulation)

With all of these modifications in place, all of the definitions in §7 can be left to stand without any changes, and all the theorems in §7 remain valid, precisely as stated.

Most of the proofs are quite straightforward adaptations of the originals. However, the proof of the composition theorem (Theorem 7) in this setting deserves some comment.

- We are assuming that we have an adversary A' that is multi-bounded for Π' such that for every well-behaved environment Z that is multi-rooted at x, we have $\text{Exec}[\Pi', A', Z] \approx \text{Exec}[\Pi'_1, A_{\text{d}}, Z]$.
- Additionally, we should assume (which we may, without loss of generality) that A' does not attempt to call any common functionalities not defined in Π'. So in the design of A, whenever A' calls a common functionality, A simply carries this out on behalf of A.

  The idea is we want to ensure that A' does not call any common functionalities defined in $\Pi \setminus x$ that are not defined in Π'.
- We also need to define how A handles an instruction from Z to send a message m to a common functionality. This should be done as follows: if the common functionality is defined in $\Pi \setminus x$, A sends m directly to the functionality, returning the result (properly translated) back to Z; otherwise, A forwards the instruction to A'.

This modified framework allows us to directly model system parameters. Moreover, although common functionalities are oblivious to the identity of their callers, the environment may still play a privileged role, based on the fact that it is the initial machine in the system (i.e., is activated first). By exploiting this fact, we can model system parameters in which the adversary is allowed to see the randomness used to generate the system parameter (called a "public coin system parameter" in [CCGS10]), or to even specify the system parameter (perhaps subject to certain constraints).

10.5 Restricted emulation

This modified framework does not, however, by itself, allow us to model CRSs. To this end, we introduce a new notion of emulation.

We begin with the following definition. Let S be a finite set of common functionality names. We say that an environment Z is **restricted by** S if it never sends messages directly to any common functionality named in S.

Definition 8 (restricted emulation). *Let Π and Π_1 be (multi-)poly-time protocols rooted at r. We say that Π_1 **(multi-)emulates** Π **restricted by** S if the following holds: for every adversary A_1 that is (multi-)bounded for Π_1, there exists an adversary A that is (multi-)bounded for Π, such that for every well-behaved environment Z that is* restricted by *S and (multi-)rooted at r, we have*

$$\text{Exec}[\Pi, A, Z] \approx \text{Exec}[\Pi_1, A_1, Z].$$

We note that the *only* difference between this and Definition 7 is that we quantify only over all environments that are restricted by S. Of course, if $S = \emptyset$, these two definitions are equivalent. Also, it is clear that if $S \subset S'$, then (multi)-emulation restricted by S implies (multi)-emulation restricted by S'.

The analogs of the four central theorems (Theorems 5–8) must now be reconsidered.

Theorem 5 essentially remains valid, with the appropriate modifications:

Theorem 10 (completeness of the dummy adversary). *Let Π and Π_1 be (multi-)poly-time protocols rooted at r. Let S be a finite set of common functionality names. Suppose that there exists an adversary A that is (multi-)bounded for Π, such that for every well-behaved environment Z that is restricted by S and (multi-)rooted at r, we have $\mathrm{Exec}[\Pi, A, Z] \approx \mathrm{Exec}[\Pi_1, A_{\mathrm{d}}, Z]$. Then Π_1 (multi-)emulates Π restricted by S.*

The proof of this theorem is almost identical to that of Theorem 5, as we leave for the reader to verify.

Theorem 6 (emulates implies multi-emulates) simply has no valid analog. To establish that one protocol multi-emulates another restricted by S, when $S \neq \emptyset$, one will simply have to prove it directly (although there may be some general tools that could be developed to simplify the analysis certain types of protocols). Because of this, while the notion of restricted multi-emulation is useful, the notion of restricted emulation is not very useful.

Theorem 7 essentially remains valid, with the appropriate modifications:

Theorem 11 (composition theorem). *Let S be a finite set of common functionality names. Suppose Π is a poly-time protocol rooted at r. Suppose Π' is a poly-time subprotocol of Π rooted at x. Suppose Π'_1 is a poly-time protocol also rooted at x that multi-emulates Π' restricted by S and that is substitutable for Π' in Π. In addition, suppose that the domain of $\Pi \setminus x$ is disjoint from S. Then $\Pi_1 := \Pi[\Pi'/\Pi'_1]$ is poly-time and multi-emulates Π restricted by S.*

Note the essential differences are the hypothesis and conclusion are stated in terms of restricted *multi*-emulation, and we have the additional hypothesis that the domain of $\Pi \setminus x$ is disjoint from S. The proof follows the same lines as discussed in §10.4.

Finally, Theorem 8 essentially remains valid, with the appropriate modifications:

Theorem 12 (reflexivity and transitivity of emulation). *Let Π, Π_1, and Π_2 be poly-time protocols, and let S and S_1 be finite sets of common functionality names. Then Π (multi-)emulates Π restricted by S. In addition, if Π_2 (multi-)emulates Π_1 restricted by S_1, and Π_1 (multi-)emulates Π restricted by S, then Π_2 (multi-)emulates Π restricted by $S_1 \cup S$.*

The proof of this theorem is almost identical to that of Theorem 8, as we leave for the reader to verify.

CRSs can be modeled as common functionalities, but where we consider emulation restricted by the set consisting of their names. Thus, we do not give the environment direct access to CRSs — only indirect access via the adversary. This allows an adversary to "program" a simulated CRS, which is a common technique used in security proofs.

We also mention that whether or not a common functionality represents a CRS or system parameter can be dependent on context, which may be convenient. For example, we may design a protocol that securely evaluates arithmetic circuits modulo N, where N is a system parameter (assumed to be generated according to a certain distribution). In this context, we may view N as a system parameter, using the usual notion of unrestricted emulation. We might then use this

protocol to design a protocol that, say, performs some other task in which N is really not a part of the description of the task, but is best viewed as an implementation detail. This can be done by simply using the notion of restricted multi-emulation.

10.6 The JUC theorem

The changes to the JUC theorem construction in §9 are quite straightforward. We simply augment the definition of boxed protocols $\widehat{\mathcal{F}}$ and $[\Pi]_{\mathcal{F}}$, so that when any of the internally simulated virtual machines call a common functionality, these calls are simply passed through. Because common functionalities are oblivious to the identities of their callers, this will not really change anything.

With this simple modification, Theorem 9 holds verbatim, even with common functionalities.

11 Comparison with UC05 and other frameworks

In this section, we compare our proposed UC framework with Canetti's UC05 framework [Can05], and (more briefly) with some other frameworks. The comparison with UC05 relies on a number of specific details in [Can05], all of which can be found on pages 27–30, 32–33, and 39–42 in [Can05].

11.1 Libraries vs explicit programs

In our framework, we use a statically defined library to associate programs with machines. In UC05, the program of a machine M is specified by the machine N that creates M — N is, by definition, the machine that first sends any type of message to M (and N can may be the environment, the adversary, a caller of M, or even a subroutine of M).

In our framework, protocol composition is a static operation on libraries, whereas in UC05, protocol composition is an action that occurs at runtime, explicitly replacing one program by another.

This is mainly a matter of taste, but we believe the library mechanism more directly corresponds to how protocols are designed and deployed in the real world.

11.2 Binding IDs to programs

In our framework, there is an explicit binding of machine IDs to programs, and moreover, the subroutine/caller relationship is explicitly encoded in the machine ID. In UC05, there is no such binding. In particular, when the adversary sends a message to or receives a message from a machine M, the adversary knows the ID of M but it does not (in general) know which program M is running, nor does the adversary know which machines are subroutines of M, or which machines M is a subroutine of.

This lack of information available to the adversary makes the composition theorem difficult, if not impossible, to prove. The proof of the composition theorem in [Can05] describes the construction of a simulator whose job it is to route messages that belong to an instance of a given subprotocol to an appropriate simulator. This is, in fact, the crux of the entire proof. However, because of the lack of information, it is not at all clear how this is to be done; the description in [Can05] is simply incomplete in this regard.

Indeed, we claim that the UC05 composition theorem is simply false. Here is a counter-example.

We start with a one-party protocol Π' that works as follows. It expects an initialization message from the environment Z, which it forwards to A. After this, it awaits a bit b from A, which it forwards to Z. If it does not receive precisely these messages in precisely this order, Π' sends

a special error message to Z, and any further messages it receives also result in the same error message to Z.

We next define a protocol Π'_1, which works exactly the same as Π', except that upon receipt of the bit b from A, it sends $1 - b$ to Z.

We claim that in any reasonable UC framework, Π'_1 emulates Π'. The simulator would work as follows. As usual, the simulator A' is attacking Π', and uses an internal copy of an adversary A'_1 that is supposed to be attacking Π'_1. When A'_1 attempts to send a bit b to Π'_1, A' instead sends the bit $1 - b$ to Π'.

We hope that the reader agrees that any reasonable UC framework (including UC05) should allow the above claim to be proved along the lines suggested.

So now consider a one-party protocol Π that works as follows. Π expects an initial message from Z, specifying a bit c; if $c = 0$, it initializes a subroutine running Π', and if $c = 1$, it initializes a subroutine running Π'_1. However, the machine ID assigned to the subroutine is the same in either case. When Π receives a bit from its subroutine, it forwards that bit to Z. Any deviation from these message flows results in an error message to Z.

The composition theorem says that we should be able to substitute Π'_1 for Π' in Π, obtaining a protocol Π_1 that emulates Π. Note that in Π_1, the subroutine called is Π'_1, regardless of the value of c.

Now consider an environment Z, designed to interact with Π_1 and the dummy adversary A_{d}, that works as follows. Z chooses $c \in \{0, 1\}$ at random, and invokes Π_1 with input c. Control will pass to the subroutine, and then to the dummy adversary, who will forward the ID of this subroutine to Z. Z responds to this by sending to the subroutine (via the dummy adversary) the bit 0. After this, the subroutine sends a bit b to Π_1, which is passed to Z. Finally, Z outputs 1 of $b = 1$, and it outputs 0 if $b = 0$ or if it receives any unexpected message.

Clearly, by construction, $[\Pi_1, A_{\mathrm{d}}, Z]$ outputs 1 with probability 1.

But now consider $[\Pi, A, Z]$ for any simulator A. A's view is independent of c — it sees exactly the same information, regardless of the program being run by the subroutine. So, whatever bit the adversary sends to the subroutine, it will get flipped with probability 1/2. Hence the probability that Z outputs 1 in this experiment is (at most) 1/2.

To make this counter-example more complete, there are a few details to take care of. The first is that in UC05, it is required that Z invokes the adversary before any protocol machines. This we can do, by just having Z send some junk message to the adversary first. Second, in designing a simulator A, since A will get control before any protocol machines are created, one might attempt to have A first create a subroutine with a program of its choice (this is allowed in UC05, but not in our framework). However, this will not work: in UC05, when Π invokes its subroutine, if the subroutine already exists but is running the wrong program, then Π will receive an error signal, which it can forward to Z.

Note that any fix to this problem must involve more than just providing a mechanism that informs the adversary of the code of any machine that sends it a message. To prove the composition theorem, the adversary essentially must be able to determine the entire protocol stack associated with a given machine, in order to determine whether it belongs to the relevant subprotocol or not. But this ultimately leads to a solution much like the one presented here.

11.3 A trust hierarchy

In our framework, the machine IDs not only explicitly describe the subroutine/caller relationship, the rules for creating and corrupting machines ensures a corresponding hierarchy of trust. Specifically,

if any subroutine of a machine M is corrupt, then M itself should be viewed as corrupt.

We believe this is a general trust principle to which any reasonable UC framework should adhere. Philosophically, a protocol is only as good as its subprotocols, and if the latter are unreliable, then so is the former. In addition to this, it is clear that security proofs in the literature assume this principle, even if it is never stated explicitly.

Here is a more concrete example that will illustrate the point. Suppose that we have some typical multi-party protocol for a typical task, such as secure function evaluation, and moreover, assume that this protocol uses a secure channel ideal functionality as a subroutine. If the adversary is allowed to corrupt a party's secure channel subroutines, then from that point forward, the adversary can essentially replace that party for the remainder of the execution of the protocol. If the adversary does this for sufficiently many parties sufficiently early in the protocol, then the adversary will be in complete control — it will be able to learn all inputs of all remaining parties, and to force the function to evaluate to a value of its choice. So even though the adversary never explicitly corrupted any machines of the secure evaluation protocol, but only subroutines thereof, we must obviously view the corresponding machines of the function evaluation protocol to be corrupt as well.

The UC05 framework does not adhere to this trust hierarchy principle at all. Because of this, many (if not most) typical security claims in the literature are simply false. Two different issues will serve to illustrate this point.

The first issue is one of simple mechanics. In UC05, a machine M is corrupted when the adversary sends a special corrupt message to a M. When this happens, *no machine other than M is corrupted*. (UC05 certainly does not define any notion of PID-wise corruption, nor is it even clear how such a notion should be defined, even though this does not stop some authors from using the notion, e.g., [CDPW07].) In addition, the adversary is only allowed to corrupt M after the environment sends the adversary a special message that authorizes this — this special authorization message explicitly names the machine ID of M.

Consider again the secure function evaluation example above. We are assuming that Π is a concrete protocol that uses as a subroutine a secure channels ideal functionality $\mathcal{G}$, and that the goal is to prove that Π emulates $\mathcal{F}$, where $\mathcal{F}$ is the secure function evaluation ideal functionality. So we are given an adversary A, and want to define a simulator S such that $\text{Exec}[\Pi, A, Z] \approx \text{Exec}[\mathcal{F}, S, Z]$ for all Z. Again, consider an attack as above where Z instructs A to corrupt many machines belonging to $\mathcal{G}$. This means that Z sends a number of authorization messages to A — but these authorizations name machines that belong to $\mathcal{G}$, and *do not name any machines that belong to $\mathcal{F}$*. So this means that in the execution of $[\mathcal{F}, S, Z]$, the simulator S is not allowed to corrupt any machines belonging to $\mathcal{F}$. Clearly, given this constraint, it will be impossible for S to do its job.

The second issue is more subtle, and relates to the mechanism by which machines are created in UC05. In UC05, an adversary is allowed to create machines as it pleases, specifying their machine IDs and their programs. In our running example, then, the adversary may create machines that "look like" dummy parties of $\mathcal{G}$ (i.e., they have machine IDs that are compatible with whatever naming scheme Π uses), but are really completely under the control of the adversary. Also, we observe that in UC05, an ideal machine of $\mathcal{G}$ carries out its task based solely on the IDs of the dummy parties, and not their programs — indeed, in UC05, an ideal machine of $\mathcal{G}$ cannot discover the program of any of its peers. So in this way, the adversary can arrange to "hijack" many secure channels, without formally corrupting any machines. Now, in UC05, when the adversary sets the program of a subroutine in this way, its caller will detect this and receive an error signal. Thus, parties with hijacked subroutines will be able to detect that this has happened, but other parties will not be able to know this. Therefore, the same attack will work.

There may be ways to design a protocol that makes the attack in the above paragraph more difficult to carry out. However, few (if any) protocols in the literature are so designed. Moreover, we feel that in any reasonable UC framework, the issue should simply not arise. Fundamentally, the problem is that many ideal functionalities, such as secure channels, zero knowledge, and others, all use machine IDs to identify parties — however, since in UC05 there is no strong binding between machine IDs and their programs, and no correspondence between machine IDs and the subroutine/caller relationship, these machine IDs do not carry much, if any, useful information that would imply any meaningful security guarantees.

One might consider trying to fix the problem by changing the framework so that a machine that receives a message, at least in certain situations, is told the program of the sender. This would prevent the above attack — but it would be easy to still come up with others. Indeed, consider the more general situation where we have a protocol stack, at the bottom of which is the secure channels functionality. The secure channels ideal functionality could verify the code of its dummy parties, but still, the adversary could hijack any machine above this machine in the protocol stack, and carry out the same type of attack. To prevent this more general type of attack, one would have to introduce conventions by which machines would securely maintain a stack of programs and machine IDs, representing the protocol stack down to that machine, and the secure channels ideal functionality would transmit all of this information, which could be inspected by the receiver. All of this could be done, but one would end up with a mechanism not much different from that which we have proposed here.

11.4 Joint subroutines

In our framework, every subroutine has a unique caller. In UC05, this is not necessarily the case. While it may be convenient to allow subroutines with multiple callers, some issues arise that need to be carefully addressed.

Recall our trust hierarchy principle, discussed in §11.3. Assume the following relationships among machines. Machine M has two subroutines, N_1 and N_2, and P is a subroutine of both N_1 and N_2. Now suppose the following machines are corrupted, in this order: M, N_1, P. The question is, should N_2 be considered corrupt or not? Our trust hierarchy principle would say yes, but none of the rules in UC05 address this. We would advocate that if a framework allows a joint subroutine, such as P, then before P is corrupted, all of its callers must be corrupted as well — in this example, it means that both N_1 and N_2 must be corrupted before P is corrupted.

While one could add fully general support for joint subroutines to our framework, we have chosen not to, for the sake of simplicity; moreover, we believe our framework is sufficiently expressive without them.

One fundamental application of joint subroutines that is found in the literature is in the JUC theorem construction. Using the notation in §9, the construction in [CR03, Can05] makes $\widehat{\mathcal{F}}$ a joint subroutine of many machines in $\Pi \setminus x$. Instead, we put all of the machines in $\Pi \setminus x$ inside a single $[\Pi \setminus x]$-machine, so that all of the machines belonging to $\Pi \setminus x$ become virtual machines running inside of the $[\Pi \setminus x]$-machine. Formally, this has very little impact, except for the following: when a single $[\Pi \setminus x]$-machine becomes corrupted, in effect, all of its internal virtual machines become corrupted as well. This may seem extreme, but yet, in light of the trust hierarchy principle, it is the right thing to do, because it forces all of these virtual machines belonging to $\Pi \setminus x$ to be corrupted before $\widehat{\mathcal{F}}$ is corrupted. Indeed, consider the example where $\widehat{\mathcal{F}}$ implements many authenticated channels using one signing key. Once $\widehat{\mathcal{F}}$ is corrupted and the signing key is exposed, all of the machines above it must be considered corrupt as well, and this is precisely what the mechanics of our approach ensures.

11.5 Restrictiveness of session IDs

Our framework places much greater restrictions on the format of session IDs than does UC05. The question is: are these restrictions excessive? We believe not.

First, one should not conflate our session IDs, which represent a logical, UC-specific protocol stack, with an actual protocol stack running on a real-world machine. These may be quite different. Typically, the UC-specific protocol stack would be determined, e.g., by the PKI (see §12.2).

Second, the virtual boxing technique in §9 can be used, if necessary, to break the naming conventions, so that the virtual, boxed machines use a different set of SIDs than those outside of the box. One example of this is discussed in §12.3, in the context of the construction of Barak *et al.* for secure computation without authentication [BCL+05].

11.6 Uniform vs non-uniform computation

In UC05, the environment receives an additional external input, which may be an arbitrary string; in contrast, our framework does not allow this.

What this means, essentially, is that the UC05 system execution is a non-uniform computation, whereas ours is uniform. In particular, to prove the security of protocols in UC05, one would typically have to make non-uniform complexity assumptions, whereas in our framework, one would only have to make uniform complexity assumptions.

We believe this choice is mostly a matter of taste. We have opted for the uniform model, mainly because it is simpler. It is not clear if the non-uniform model truly captures any attacks in the real world not captured by the uniform model — this seems more of a philosophical debate. Nevertheless, we believe that modifying our framework to make it non-uniform should be straightforward.

11.7 Running time

Our definition of a poly-time protocol is completely different than that in UC05. Before going further, we summarize the definition in UC05.

In UC05, each machine has an input tape, a subroutine output tape, and a communication tape. The idea is that when a machine calls a subroutine, the former writes the input message on the latter's input tape; when the subroutine wishes to pass an output to its caller, the former writes the output message on the subroutine output tape of the latter; the communication tape is used to send and receive "network" messages; more specifically, all communication between a protocol machine and the adversary is done via communication tapes. Additionally, for the sake of these definitions, the adversary is considered a subroutine of the environment, and ideal machines are considered to be (joint) subroutines of their regular peers.

In UC05, a machine is poly-time essentially if it runs in time polynomial in Δ, where Δ is equal to the number of bits received on its input tape minus the number of bits sent as input to its subroutines. Actually, the definition in UC05 is slightly more complicated, but for our purposes, the definition we give here is sufficient.

Finally, in UC05, a protocol is poly-time if all of the machines comprising it are poly-time. When considering the execution of a system consisting of a protocol, adversary, and environment, the protocol must be poly-time, as well as the adversary and environment.

One nice thing about this definition is that it is fairly simple. However, we argue that it is overly restrictive. The paper [HUMQ09] gives a number of reasons why. One reason, given in [HUMQ09], is most easily seen by way of an example. Consider two parties P and Q that wish to use a secure channels ideal functionality $\mathcal{F}$ to communicate. P may receive a very long message m from the environment, and attempt to send m to Q via $\mathcal{F}$. The problem is that Q receives m as an output

from a subroutine. In fact, Q may have received little or no input so far. Because of this, Q will not have enough time to even read m.

The only solution to this, and similar types of problems, is to use some kind of artificial padding mechanism. That is, the environment must feed Q some long string as input just to give it enough "juice" to read m.

Such padding issues arise in several places in UC05. The dummy adversary, when it receives a long message from a protocol machine, will not, in general, have enough juice to forward that message to the environment. The same applies to dummy parties associated with ideal functionalities. Similarly, achieving any notion of liveness (see Note 7.5) will be difficult, since an adversary may flood a protocol machine with useless messages — to be able to keep up with this flood, the machine will need padding.

Dealing with all of this padding would lead to a great deal of complexity if one attempted to actually specify a protocol completely. We believe this would be quite impractical, if one wanted to actually design, build, and deploy a protocol using the framework. Many (if not most) protocols in the literature that are designed to use the UC05 framework completely leave out such padding details.

In addition to the sheer complexity, there is also a modularity issue. Suppose we want to design a protocol Π for some task, and to prove that it emulates some ideal functionality $\mathcal{G}$. The protocol Π may be designed to use many subroutines, and all of these subroutine calls will reduce the amount of juice available to Π to do any work. In fact, without extra padding, Π may not have time to any work at all, other than call subroutines. The only solution, again, is padding. Π will have to be fed some padding from the environment to do its work. Since the environment's interface to Π must look exactly the same as the environment's interface to $\mathcal{G}$, all of these padding messages must be in the interface to $\mathcal{G}$, as well. In general, the amount of padding required will depend on the details of Π. So if we want to have a single ideal functionality $\mathcal{G}$ that supports multiple implementations, each with its own padding requirement, the specification of $\mathcal{G}$ will become somewhat delicate, although it can be done.

There are other technical problems with the definition of poly-time in UC05. For example, because of time-related issues, the theorem in [Can05] that states that the dummy adversary is complete in UC05 is simply false — [HUMQ09] provides a counter-example. This theorem is used in an essential way in the proof of the composition theorem, and so this gives us another fundamental problem with the composition theorem in UC05.

We also raise a somewhat philosophical objection. Namely, the definition in UC05 is not robust with respect to data encodings, which is a generally desirable property in defining resource bounds in general. Specifically, since the running-time bound is determined by the *difference* of the length of two strings, it becomes very sensitive to encodings; for example, this difference may be positive or negative, depending on whether some numbers are encoded in decimal or binary. This goes against the general "robustness principle" in complexity theory, which say that such definitions should be relatively independent of encoding details.

Our definition is closely related to, and inspired by, the definition in [HUMQ09]. Both definitions avoid all of the problems discussed above with the UC05 definition. Except in a few very unusual use cases, no artificial padding should be required. The main advantage of our definition over that in [HUMQ09] is that it is closed under composition of protocols. To compare, in the composition theorem in [HUMQ09] (the analog of our Theorem 7), it must be assumed as a hypothesis, rather than derived as a conclusion (as in our theorem), that Π_1 is poly-time. This is a significant advantage, as it allows for a purely modular protocol design — in [HUMQ09], one must (in general) carry out a run-time analysis "from scratch" whenever one applies the composition theorem. There

is a trade-off, however. In our definition, we have the flow-boundedness restriction on adversaries, which can lead to difficulties. Our "invitation" mechanism mitigates these difficulties to some degree, but nevertheless, there can still be some challenging cases — see §12.1 for some examples of the issues and how to deal with them. While dealing with the flow-boundedness restrictions may require some care in designing ideal functionalities, based on our experience so far, it very rarely requires any artificial modification of actual protocols.

To put all of this in perspective, we summarize the main goals we tried to achieve in devising our notion of poly-time:

(i) It should be natural and non-restrictive; in particular, common situations as described above, such as sending a message over a secure channel, should not require special padding or other "hacks".

(ii) It should be preserved under composition.

We believe we have come reasonably close to achieving all of these goals. Arguably, the flow-boundedness restriction is not ideal, but we believe it is not a major drawback: in the use cases we have studied, it seem that it usually causes no problem at all, and even when it does, easy fixes are at hand.

11.8 Comparison to the IITM framework

In [Küs06], Küsters presents the IITM framework for the modeling and analysis of multi-party protocols (including several theorems for securely composing protocols). Syntactically, the IITM framework is very different from UC05 and our framework. As such, direct comparison is somewhat difficult. Nevertheless, we can make the following observations:

- The IITM framework does not directly support protocols with an unbounded number of parties. While it may be possible to implement such "many-party" protocols on top of his framework (e.g., [KT09] points in this direction), this would involve a significant amount of work, in terms of developing appropriate conventions and theorems.
- The IITM framework does not define corruptions, leaving this completely up to the logic of the protocol. One could establish conventions regarding corruptions, but this has not been done.
- The IITM definition of poly-time closely resembles that if UC05, and is subject to many of the same padding issues.
- While it is ultimately a matter of taste, we believe that our framework (like UC05) is more closely aligned with the methodology of protocol design traditionally used by practitioners, and as such, may perhaps be easier to use.

11.9 Comparison to the Reactive Simulatability framework

In [BPW07] (see also [PW01]), Backes, Pfitzmann, and Waidner put forward the Reactive Simulatability (RS) framework for multi-party protocols. Like the IITM framework, the RS framework is syntactically quite different from UC05. Furthermore, RS is traditionally used rather for the formulation of computational soundness results (e.g., [BPW03]) than for the analysis of concrete protocol constructions. A few more specific observations follow:

- Like the IITM framework, the plain RS framework does not support an unbounded number of parties. Now, there exist RS generalizations which support a variable number of parties (e.g., [BPW04]); however, these generalizations are rather formalistic and seem like a proof of concept. In particular, to the best of our knowledge, these generalizations have not been used to model or analyze cryptographic protocols.
- In the original RS framework, parties halt after a (fixed) polynomial number of overall steps. This requires a somewhat inconvenient parameterization of functionalities over a number of usages, or concrete running time bounds. This can be somewhat mitigated by adapting the notion of poly-time, e.g., as in [HMQU05].
- Like UC05, the RS framework does not define any form of trust hierarchy.

12 Examples

In this section, we give some examples that illustrate the use of our framework.

12.1 Some fundamental ideal functionalities

We present here some fundamental ideal functionalities, designed in a way that is compatible with our framework. These are similar to corresponding functionalities in [CCGS10]; among other things, they have been modified to mesh well with our poly-time and flow-boundedness requirements.

Before we begin, we recall that the machine ID of any protocol machine (regular or ideal) is of the form $\langle\, pid, sid \,\rangle$, where *pid* is called the *party ID (PID)* and *sid* is called the *session ID (SID)*. A session ID is of the form $\langle \ldots, basename \,\rangle$, and a *basename* is of the form $\langle\, protName, sp \,\rangle$. Here, *protName* is the name of the program run by the machine, and *sp* is an application specific *session parameter* (or *SP*).

Typically, the behavior of an ideal functionality should not depend in any significant way on its protocol name. While this could be formalized, we refrain from doing so. More generally, we would expect that protocols and simulators do not depend in any significant way on the protocol name. Again, this could be formalized, but we do not do so here.

From the above discussion, it follows that when describing a protocol or ideal functionality, the only part of the SID that requires description is the session parameter.

12.1.1 Authenticated Channels

We present here an ideal functionality for an authenticated channel. This ideal functionality is called $\mathcal{F}_{\text{ach}}$.

The SP for this functionality is of the form $\langle\, P_{\text{pid}}, Q_{\text{pid}}, label \,\rangle$, where P is the sender and Q is the receiver, and *label* is an arbitrary string that may be used to distinguish different channels. To be clear, P and Q are dummy parties that are peers of the ideal functionality (so they all share the same SID) whose PIDs are P_{pid} and Q_{pid}, respectively. We now present the logic of the *ideal functionality* $\mathcal{F}_{\text{ach}}$, interacting with an adversary A. The notation is explained below.

`send`: accept $\langle\, \texttt{send}, x \,\rangle$ from P; $\bar{x} \leftarrow x$; send $\langle\, \texttt{send}, x \,\rangle$ to A.

`ready`: accept $\langle\, \texttt{ready} \,\rangle$ from Q; send $\langle\, \texttt{ready} \,\rangle$ to A.

`done` [`send`]: accept $\langle\, \texttt{done} \,\rangle$ from A; send $\langle\, \texttt{done} \,\rangle$ to P.

`deliver` [`send` $\wedge$ `ready`]: accept $\langle$ `deliver`, x $\rangle$ from A, where $x = \bar{x}$; send $\langle$ `deliver`, $\bar{x}$ $\rangle$ to Q.

`corrupt-sender`: accept $\langle$ `corrupt` $\rangle$ from P; send $\langle$ `corrupt-sender` $\rangle$ to A.

`reset` [`corrupt-sender`]: accept $\langle$ `reset`, x $\rangle$ from A; $\bar{x} \leftarrow x$.

Note 12.1. Each step is labeled by a name. By convention, each step may only be triggered once. A logical expression in $[\cdots]$ is a guard that must be satisfied in order to trigger the step; a step name in such an expression denotes the event that the corresponding step has been triggered. Each step begins with an *accept clause*, which describes the form of the message that triggers this step; such an accept clause may itself have logical conditions which must be satisfied in order to trigger the step.

Any message that the ideal functionality receives that does not trigger one of these steps is processed by simply sending an error message to A.

These notational conventions shall be in force in all of the descriptions of ideal functionalities presented here. □

Note 12.2. Note that in the `deliver` step, A is required to send the message $\langle$ `deliver`, $\bar{x}$ $\rangle$. Including $\bar{x}$ in this message would seem unnecessary; however, including it forces A to inject some flow into $\mathcal{F}_{\text{ach}}$, which will help to maintain the flow-boundedness condition in the analysis of higher-level protocols.

It is expected that in any protocol Π that emulates $\mathcal{F}_{\text{ach}}$, any adversary attacking Π must inject flow whose length is polynomially related to that of $\bar{x}$; thus, in designing a simulator to prove that Π that emulates $\mathcal{F}_{\text{ach}}$, the flow-boundedness constraint will be easily satisfied.

By adhering to analogous conventions in designing ideal functionalities, it should be straightforward, in most cases, to maintain the flow-boundedness condition when composing protocols. We will see more examples of this below. □

Note 12.3. The `done` step is there to allow control to return to P directly from the ideal functionality, rather than from P's caller. This may be convenient in the design of higher-level protocols, in that it makes $\mathcal{F}_{\text{ach}}$ behave more like a traditional subroutine, with a call and a return step. One could certainly design a variant of $\mathcal{F}_{\text{ach}}$ which leaves out this step. □

Note 12.4. Like the corresponding functionality in [Can05, Section 6], this one allows delivery of a single message per session. Multiple sessions should be used to send multiple messages. Alternatively, one could also define a multi-message functionality. □

Note 12.5. Unlike the corresponding functionality in [Can05, Section 6], the receiver here must explicitly initialize the channel before receiving a message. This conforms to our constraints (see Constraint C6 in §4.5). In [Can05], the sender can essentially spontaneously create a party on the receiving end just by sending it a message via the ideal functionality. While in some settings such behavior may be convenient or even necessary, this seems to add significant complexity to the model. Moreover, such behavior makes denial-of-service attacks much easier, since the receiving side may be forced to start executing arbitrarily complex protocols (not just the communication protocol, but protocols "up the stack" which may be triggered by it) that are entirely unrelated to any computation it actually wants to perform. □

Note 12.6. If P' is an arbitrary regular protocol machine that wants to send a message to a peer Q', then P' and Q' must invoke subroutines P and Q, where P and Q are dummy parties of the

$\mathcal{F}_{\text{ach}}$ functionality with the same session ID. This means that P' and Q' must agree (typically by some simple, protocol-specific convention) to identify the channel using the label in the session parameter. In many cases, a trivial numbering scheme will suffice. □

Note 12.7. In describing the corruption rule, we mean the reaction of $\mathcal{F}_{\text{ach}}$ when it receives the special corrupt message from one of its peers. The peer itself may be corrupted without the ideal functionality being notified of this, and no special actions occur at that time. However, in typical security proofs, one may assume without loss of generality that both events occur, one right after the other. A similar comment applies to all the ideal functionalities presented here. □

12.1.2 Secure channels

Secure channels provide both authentication and secrecy. We present a ideal functionality $\mathcal{F}_{\text{sch}}$ that is tuned to adhere to our conventions. It is designed so that it can be realized assuming secure erasures.

An SP for $\mathcal{F}_{\text{sch}}$ has the same form as that of $\mathcal{F}_{\text{ach}}$, that is, $\langle P_{\text{pid}}, Q_{\text{pid}}, \mathit{label} \rangle$, where P is the sender and Q is the receiver. Also, $\mathcal{F}_{\text{sch}}$ is parameterized by a "leakage" function $\ell : \mathbf{\Sigma}^* \to \mathbf{\Sigma}^*$, so that $\ell(x)$ represents the information (such as length) that is allowed to be leaked when the message x is sent over the channel. The length of $\ell(x)$ should be polynomially related to the length of x; for example, $\ell(x) = 1^{|x|}$ is a likely candidate function. For an adversary A, the ideal functionality $\mathcal{F}_{\text{sch}}$ runs as follows.

`send`: accept $\langle$ `send`, x $\rangle$ from P; $\bar{x} \leftarrow x$; send $\langle$ `send`, $\ell(x)$ $\rangle$ to A.

`ready`: accept $\langle$ `ready` $\rangle$ from Q; send $\langle$ `ready` $\rangle$ to A.

`lock` [`send` $\wedge$ `ready`]: accept $\langle$ `lock` $\rangle$ from A; send $\langle\ \rangle$ to A.

`done` [`lock`]: accept $\langle$ `done` $\rangle$ from A; send $\langle$ `done` $\rangle$ to P.

`deliver` [`lock`]: accept $\langle$ `deliver`, L $\rangle$ from A, where $L = \ell(\bar{x}) \vee$ `corrupt-receiver`; send $\langle$ `deliver`, $\bar{x}$ $\rangle$ to Q.

`corrupt-sender`: accept $\langle$ `corrupt` $\rangle$ from P; send $\langle$ `corrupt-sender` $\rangle$ to A, along with an invitation for the message $\langle$ `expose` $\rangle$.

`corrupt-receiver`: accept $\langle$ `corrupt` $\rangle$ from Q; send $\langle$ `corrupt-receiver` $\rangle$ to A.

`reset` [$\neg$`lock` $\wedge$ `corrupt-sender`]: accept $\langle$ `reset`, x $\rangle$ from A; $\bar{x} \leftarrow x$; send $\langle\ \rangle$ to A.

`expose` [`send` $\wedge\ \neg$`lock` $\wedge$ `corrupt-sender`]: accept $\langle$ `expose` $\rangle$ from A; send $\langle$ `expose`, $\bar{x}$ $\rangle$ to A.

Note 12.8. For flow-related reasons similar to those in $\mathcal{F}_{\text{ach}}$, A must inject flow corresponding to P's input into the functionality already at the `deliver` step, but only if Q is not already corrupted. In any reasonable implementation, this flow will certainly be available when the simulator needs it. Note that if Q is corrupted before the `deliver` step, A may obtain P's input (via Q) by triggering the `deliver` step (after the `lock` step), but without having to inject the extra flow. □

Note 12.9. As in $\mathcal{F}_{\text{ach}}$, this functionality only allows a single message per session to be transmitted. □

Note 12.10. As in $\mathcal{F}_{\text{ach}}$, a message can only be delivered to a receiver who has initialized the channel. □

Note 12.11. The locking and corruption logic implies the following security properties:

- after the `done` step, corrupting P will not allow P's input to be exposed or reset;
- after the `deliver` step, corrupting either P or Q will not expose P's input.

□

Note 12.12. $\mathcal{F}_{\text{sch}}$ can be easily realized using $\mathcal{F}_{\text{ach}}$ and some cryptography — for example, Diffie-Hellman key exchange, and a one-time pad and a one-time MAC to actually encrypt and authenticate the message. Such an implementation crucially depends on secure erasures. The `lock` step in $\mathcal{F}_{\text{sch}}$ would correspond to a step in the implementation in which the sender P erases both the one-time pad and the one-time MAC key (along with any ephemeral, unerased Diffie-Hellman secret keys), and then sends (in either order) the `done` message to its caller, and the MAC-authenticated ciphertext to the adversary (for eventual delivery to Q). □

Note 12.13. Note the use of invitations in the `corrupt-sender` step; this streamlines the specification of $\mathcal{F}_{\text{sch}}$, but could be avoided. □

12.1.3 Zero knowledge

Let R be a binary relation, consisting of pairs (x, w): for such a pair, x is called the "statement" and w is called the "witness". We describe an ideal functionality $\mathcal{F}_{\text{zk}}$, parameterized by the relation R, as well as a "leakage" function $\ell : \mathbf{\Sigma}^* \times \mathbf{\Sigma}^* \to \mathbf{\Sigma}^*$. Again, the length of $\ell(x, w)$ should be polynomially related to the length of $\langle\, x, w \,\rangle$; however, to facilitate implementation, it may be useful to include some information about the "structure" of x and w. We stress that this functionality is designed to be realized in the secure erasures model. It provides somewhat stronger security guarantees than more traditional zero-knowledge notions.

An SP for $\mathcal{F}_{\text{zk}}$ is of the form $\langle\, P_{\text{pid}}, Q_{\text{pid}}, \mathit{label} \,\rangle$, where P is the prover and Q is the verifier. Interacting with an adversary A, the ideal functionality $\mathcal{F}_{\text{zk}}$ runs as follows.

`send`: accept $\langle\, \texttt{send}, x, w \,\rangle$ from P, where $(x, w) \in R$; $\bar{x} \leftarrow x$; $\bar{w} \leftarrow w$; send $\langle\, \texttt{send}, \ell(x, w) \,\rangle$ to A.

`ready`: accept $\langle\, \texttt{ready} \,\rangle$ from Q; send $\langle\, \texttt{ready} \,\rangle$ to A.

`lock` [`send` $\wedge$ `ready`]: accept $\langle\, \texttt{lock} \,\rangle$ from A; send $\langle\, \rangle$ to A.

`done` [`lock`]: accept $\langle\, \texttt{done} \,\rangle$ from A; send $\langle\, \texttt{done} \,\rangle$ to P.

`deliver` [`lock`]: accept $\langle\, \texttt{deliver}, L \,\rangle$ from A, where $L = \ell(\bar{x}, \bar{w}) \vee$ `corrupt-receiver`; send $\langle\, \texttt{deliver}, \bar{x} \,\rangle$ to Q.

`corrupt-sender`: accept $\langle\, \texttt{corrupt} \,\rangle$ from P; send $\langle\, \texttt{corrupt-sender} \,\rangle$ to A, along with an invitation for the message $\langle\, \texttt{expose} \,\rangle$.

`corrupt-receiver`: accept $\langle\, \texttt{corrupt} \,\rangle$ from Q; send $\langle\, \texttt{corrupt-receiver} \,\rangle$ to A.

`reset` [$\neg$`lock` $\wedge$ `corrupt-sender`]: accept $\langle$ `reset`, x, w $\rangle$ from A, where $(x, w) \in R$; $\bar{x} \leftarrow x$; $\bar{w} \leftarrow w$; send $\langle\ \rangle$ to A.

`expose` [`send`$\wedge\neg$`lock`$\wedge$`corrupt-sender`]: accept $\langle$ `expose` $\rangle$ from A; send $\langle$ `expose`, $\bar{x}, \bar{w}$ $\rangle$ to A.

Note the similarity with our secure channels functionality.

In the above, the relation R was considered to be a fixed relation — more precisely, there is one relation per value of the security parameter. However, for many applications, it is convenient to let R be parameterized by a some system parameter (see §10), such as a prime number, an RSA modulus, or an elliptic curve. $\mathcal{F}_{\text{zk}}$ can be efficiently realized in the CRS model for many useful relations using $\mathcal{F}_{\text{sch}}$ and techniques such as those in [MY04, JL00, CKY09].

12.1.4 Commitment

Here is an ideal functionality $\mathcal{F}_{\text{com}}$ for commitment. Again, it is parameterized by a leakage function $\ell : \mathbf{\Sigma}^* \to \mathbf{\Sigma}^*$, and is designed to be realized in the secure erasures model.

An SP for $\mathcal{F}_{\text{com}}$ is of the form $\langle P_{\text{pid}}, Q_{\text{pid}}, \mathit{label} \rangle$, where P is the sender and Q is the receiver. Interacting with an adversary A, the ideal functionality $\mathcal{F}_{\text{com}}$ runs as follows.

`send`: accept $\langle$ `send`, x $\rangle$ from P; $\bar{x} \leftarrow x$; send $\langle$ `send`, $\ell(x)$ $\rangle$ to A.

`ready1`: accept $\langle$ `ready1` $\rangle$ from Q; send $\langle$ `ready1` $\rangle$ to A.

`done1` [`send`]: accept $\langle$ `done1` $\rangle$ from A; send $\langle$ `done1` $\rangle$ to P.

`commit` [`send` $\wedge$ `ready1`]: accept $\langle$ `commit`, L $\rangle$ from A, where $L = \ell(\bar{x})$; send $\langle$ `commit` $\rangle$ to Q.

`open` [`send`]: accept $\langle$ `open` $\rangle$ from P; send $\langle$ `open`, $\bar{x}$ $\rangle$ to A.

`done2` [`open`]: accept $\langle$ `done2` $\rangle$ from A; send $\langle$ `done2` $\rangle$ to P.

`ready2` [`commit`]: accept $\langle$ `ready2` $\rangle$ from Q; send $\langle$ `ready2` $\rangle$ to A.

`deliver` [`open` $\wedge$ `ready2`]: accept $\langle$ `deliver` $\rangle$ from A; send $\langle$ `deliver`, $\bar{x}$ $\rangle$ to Q.

`corrupt-sender`: accept $\langle$ `corrupt` $\rangle$ from P; send $\langle$ `corrupt-sender` $\rangle$ to A, along with an invitation for the message $\langle$ `expose` $\rangle$.

`reset` [$\neg$`commit` $\wedge$ `corrupt-sender`]: accept $\langle$ `reset`, x $\rangle$ from A; $\bar{x} \leftarrow x$; send $\langle\ \rangle$ to A.

`expose` [`send` $\wedge$ `corrupt-sender`]: accept $\langle$ `expose` $\rangle$ from A; send $\langle$ `expose`, $\bar{x}$ $\rangle$ to A.

Note 12.14. This commitment functionality does not preserve the secrecy of the sender's input x. In that sense, it is more like $\mathcal{F}_{\text{ach}}$ than $\mathcal{F}_{\text{sch}}$. □

Note 12.15. Variants of this general $\mathcal{F}_{\text{com}}$ may restrict the set of inputs from P to a specific set, typically reflecting limitations of the expected implementation. □

Note 12.16. A must inject flow corresponding to P's input into the functionality already at the `commit` step. In any reasonable implementation, this flow will certainly be available when the simulator needs it, unless Q is already corrupted; however, if Q is already corrupted, there is no need for A to ever trigger this step. Moreover, having this flow injected at this time facilitates

the maintenance of the flow-boundedness constraint in the analysis of protocols that use $\mathcal{F}_{\text{com}}$ as a subroutine.

For example, suppose we have a version of $\mathcal{F}_{\text{com}}$ that allows commitment to strings of a particular form (but whose the length is not a priori bounded). Using this, we can easily build a protocol that securely performs commitment to a pair of such strings: the protocol performs the commitment to each string, one after the other. The sender could be corrupted after having committed to the first string, but not to the second. The simulator should be able to reset the input pair of the sender — it will only be able to reset the second element of the pair, but the specification requires it to give the entire pair. Fortunately, the simulator will already have flow corresponding to the committed first element, and the flow for the second will naturally come from the environment. Without having this flow from the already committed first element, it would not be possible to design a simulator that maintained flow-boundedness. This illustrates how flow-boundedness can be maintained using appropriate conventions. One could have also dealt with this issue by defining a more *ad hoc* specification of $\mathcal{F}_{\text{com}}$; however, we believe the general conventions outlined here are simpler, and more generally useful. □

Note 12.17. A generally useful convention for designing ideal functionalities that maintain flow bounds runs something like this: whenever any reasonable implementation would inject a certain amount of flow into the protocol, the ideal functionality should force the simulator to inject a corresponding (polynomially related) amount of flow into the functionality. Of course, such a strategy may make ideal functionalities *somewhat* dependent on their implementation; however, such dependencies are hard to completely avoid in general; moreover, these particular dependencies only affect the adversary/functionality interface, and not the (more important) environment/protocol (i.e., I/O) interface. □

12.1.5 Secure function evaluation

Here is an ideal functionality $\mathcal{F}_{\text{eval}}$ for secure two-party function evaluation. Again, it is parameterized by a leakage function $\ell : \mathbf{\Sigma}^* \to \mathbf{\Sigma}^*$, as well as a poly-time computable function $f : \mathbf{\Sigma}^* \times \mathbf{\Sigma}^* \to \mathbf{\Sigma}^*$. As above, it is designed to be realized in the secure erasures model.

An SP for $\mathcal{F}_{\text{eval}}$ is of the form $\langle P^{(0)}_{\text{pid}}, P^{(1)}_{\text{pid}}, \mathit{label} \rangle$, where $P^{(0)}$ and $P^{(1)}$ are the participants. Interacting with an adversary A, the ideal functionality $\mathcal{F}_{\text{eval}}$ runs as follows. As the functionality is symmetric, most of the rules come in pairs, as indicated by the notation "$(i = 0, 1)$".

- $(i = 0, 1)$ `input-`i: accept $\langle$ `input`$, x \rangle$ from $P^{(i)}$; $\bar{x}_i \leftarrow x$; send $\langle$ `input-`$i, \ell(x) \rangle$ to A.
- $(i = 0, 1)$ `commit-`i [`input-`i]: accept $\langle$ `commit-`$i, L \rangle$ from A, where $L = \ell(\bar{x}_i) \vee$ `corrupt-`$(1 - i)$; send $\langle\ \rangle$ to A.
- `lock` [`commit-0` $\wedge$ `commit-1`]: accept $\langle$ `lock` $\rangle$ from A; $\bar{y} \leftarrow f(\bar{x}_0, \bar{x}_1)$; send $\langle\ \rangle$ to A.
- $(i = 0, 1)$ `output-`i [`lock`]: accept $\langle$ `output-`i $\rangle$ from A; send $\bar{y}$ to $P^{(i)}$.
- $(i = 0, 1)$ `corrupt-`i: accept $\langle$ `corrupt` $\rangle$ from $P^{(i)}$; send $\langle$ `corrupt-`i $\rangle$ to A, along with an invitation for the message $\langle$ `expose-`i $\rangle$.
- $(i = 0, 1)$ `expose-`i [`corrupt-`$i \wedge$`input-`i]: accept $\langle$ `expose-`i $\rangle$ from A; send $\langle$ `expose-`$i, \bar{x}_i$ $\rangle$ to A.
- $(i = 0, 1)$ `reset-`i [`corrupt-`$i \wedge \neg$`commit-`i]: accept $\langle$ `reset-`i, x $\rangle$ from A; $\bar{x}_i \leftarrow x$; send $\langle\ \rangle$ to A.

Note 12.18. Variants of this general $\mathcal{F}_{\text{eval}}$ may restrict the set of inputs from $P^{(0)}$ and $P^{(1)}$ to specific sets, and also specify two different functions f_0 and f_1, in place of f, so that $P^{(0)}$ and $P^{(1)}$ receive different outputs. □

Note 12.19. As in $\mathcal{F}_{\text{com}}$, A is required to inject flow corresponding to the input of each party when that party commits (at least when the other party is not already corrupt). Again, in any reasonable implementation, this flow should be available to the simulator, and injecting this flow will help to preserve flow bounds in the analysis of higher-level protocols. □

12.1.6 Some problematic functionalities

There are some ideal functionalities which may cause some trouble with respect to flow-boundedness and running time. Two such functionalities are $\mathcal{F}_{\text{sig}}$ and $\mathcal{F}_{\text{pke}}$, which are meant to model signatures and encryption, respectively. We will not present these functionalities in detail, but rather, we refer the reader to §7.2 of [Can05]. Nevertheless, we can sketch the problem, and suggest several different solutions.

In the formulation of $\mathcal{F}_{\text{sig}}$ (resp., $\mathcal{F}_{\text{pke}}$) in [Can05], the ideal-world adversary sends the signing (resp., encryption) algorithm to the functionality. In the intended implementation, nothing like this happens — the algorithm is a part of the implementation, and no corresponding interaction occurs between the implementation and the real-world adversary. So the problem is that any simulator used to show that the implementation realizes the ideal functionality will send a message to the ideal functionality that was not provoked by any corresponding message from the environment, and thus, will not be flow-bounded (and we saw in Note 7.2 how precisely this type of situation can lead to very real running-time problems).

Here are three possible work-arounds.

1. Simply do not use these functionalities: in practice (i.e., the practice of proving security theorems), these functionalities tend to be no easier to use than directly using the security definitions of the primitives themselves.

2. Parameterize the functionalities by the algorithms: instead of having the ideal-world adversary deliver the algorithm, just have the algorithm "hardwired" into the functionality, so that the functionality is really a family of functionalities parameterized by such algorithms. Such parameterizations are nothing new — in fact, in the formulations of these functionalities in [Can05] are parameterized by a "message space". Again, in practice, parameterizing these functionalities in this way should not make them any harder to use in proving the security of protocols that use them as subprotocols.

3. Modify the implementation: insist that the implementation requests from the real-world adversary some string long enough to ensure that flow bounds will be preserved in the security proof. This makes the implementation look much more like the ideal functionality. One possible objection to this approach is that this extra interaction may somehow "weaken" the security of the implementation. However, this is a spurious objection, since presumably, only the properties of the ideal functionality, and not the implementation, will be used in the analysis of higher-level protocols.

Work-around (3) requires the implementation to be modified, introducing some artificial "padding". Based on our experience so far, this is the *only* type of example we have yet encountered of a protocol that requires any such artificial modification in order to satisfy our flow-boundedness constraint.

Besides these flow problems, there is another running-time-related problem with these ideal functionalities, namely, the ideal functionality will execute a program supplied by the adversary, and the running time of this program may not be bounded in any reasonable way so as to ensure the ideal functionality is itself poly-time (under *any* reasonable definition of poly-time). This is a problem for the framework in [Can05] as well as for ours. Work-arounds (1) and (2) above will also solve this problem, and there are probably other work-arounds as well.

12.2 Modeling a PKI

We sketch here a simple method for modeling a public-key infrastructure (PKI).

We start with an ideal functionality $\mathcal{F}_{\text{ca}}$, representing a certificate authority. This functionality is parameterized by a secure signature scheme (so different signature schemes yield different functionalities). The SP for $\mathcal{F}_{\text{ca}}$ is empty. The behavior of $\mathcal{F}_{\text{ca}}$ is quite simple. Upon its first activation, it generates a verification/signing key pair (vk, sk) for the signature scheme. When it receives a message $\langle\, \texttt{request}, m \,\rangle$ from a peer with machine ID id, it signs the message $\langle\, id, m \,\rangle$ using its signing key sk, obtaining the signature σ, and returns the message $\langle\, \texttt{response}, \sigma, vk \,\rangle$ to that machine. Any other messages it receives are ignored (which means an error message is sent to the adversary).

Unlike most ideal functionalities, which are best thought of as imaginary machines that exist for the purposes of modular protocol design and analysis, $\mathcal{F}_{\text{ca}}$ directly models a real-world service. For this model to be valid, the communication links between $\mathcal{F}_{\text{ca}}$ and its peers must be securely authenticated, in both directions. How this is done is outside the model, but may, for example, be realized using physical assumptions. The point is that these authenticated links may be quite costly to use, but each user in the system will only have to interact with $\mathcal{F}_{\text{ca}}$ once.

Now let $\mathcal{F} = \mathcal{F}_{\text{ach}}$ be the authenticated channel functionality described above in §12.1.1. Using $\mathcal{F}_{\text{ca}}$, it is easy to realize the multi-session extension $\widehat{\mathcal{F}}$ of $\mathcal{F}$, as discussed in the context of the JUC theorem in §9. The idea is straightforward. A participant in the protocol $\widehat{\mathcal{F}}$, whose machine ID is id, generates its own verification/signing key pair (vk', sk') using a secure signature scheme (not necessarily the same scheme used by $\mathcal{F}_{\text{ca}}$) and calls $\mathcal{F}_{\text{ca}}$ to obtain σ and vk, where σ is a signature on the message $\langle\, id, vk' \,\rangle$ and vk is the verification key of $\mathcal{F}_{\text{ca}}$. In this way, σ plays the role of a certificate. Authenticated messages can be sent between participants of $\widehat{\mathcal{F}}$ by using these signing keys and certificates. The messages signed by the participants should be augmented to include the corresponding virtual SID.

Given a realization of $\widehat{\mathcal{F}}$, we can then apply the JUC theorem (Theorem 9). The idea is this. We start with an $\mathcal{F}$-hybrid protocol Π. The protocol Π may itself have been derived by successive applications of the composition theorem, successively refining the protocol until a very concrete protocol, which only depends on the authenticated channels ideal functionality, is obtained. In addition, the protocol Π may represent a suite of useful protocols, with the top-level machine in Π serving as a multiplexer to instances of the various protocols in the suite. This multiplexer could also assist in establishing session IDs for its subroutines, for example, by exchanging random nonces with its peers, and concatenating these together to form a unique identifier embedded in the session ID of the subroutine (the nonce would be chosen locally by the multiplexer's caller). Moreover, the derivation of Π might itself involve the JUC theorem: given relatively slow, signature-based authenticated channels, some subprotocols may use these, along with a key-exchange protocol, to get very fast secure channels based on symmetric-key cryptography.

So we apply the JUC theorem to Π and $\mathcal{F}$, obtaining $[\Pi]_{\mathcal{F}}$, and use the ordinary composition theorem to replace $\widehat{F}$ by the instantiation based on $\mathcal{F}_{\text{ca}}$ described above. The resulting protocol is an $\mathcal{F}_{\text{ca}}$-hybrid protocol that quite closely resembles the way such protocols are traditionally

designed in practice.

12.3 Secure computation without authentication

It should be straightforward to translate the results in [BCL+05] into our framework, using a variant of the virtual boxing technique from our JUC theorem construction (see §9); however, a careful verification of this claim should be the subject of future work.

In a nutshell, the idea in [BCL+05] is to design protocols that work without any authentication infrastructure (like a PKI), but still offer some meaningful security. The results hold in a multi-party setting, but for simplicity, consider just a two-party protocol. Suppose $\mathcal{F}$ is some authenticated or secure channels ideal functionality, and that Π is a two-party $\mathcal{F}$-hybrid protocol that emulates some ideal functionality $\mathcal{G}$.

Now, using a very simple construction, one can transform Π into a protocol Π' that uses no ideal functionalities, and emulates an ideal functionality $\mathcal{G}'$, which is (roughly) defined as follows: if P and Q are the participants in the protocol, then the adversary A must first decide if P and Q are to be *isolated* or *joined*, and inform $\mathcal{G}'$ of this decision. Subsequently, if P and Q are joined, then $\mathcal{G}'$ behaves essentially like $\mathcal{G}$. If they are isolated, $\mathcal{G}'$ internally runs two independent virtual copies of $\mathcal{G}$, where in one copy, A is allowed to play the role of P, and in the other, A is allowed to play the role of Q.

In the implementation, each party running Π' first performs "handshake protocol" that in effect establishes an "ephemeral PKI". Then, using the results of this step, each party internally runs a simulated copy of Π, but with an appropriately "mangled" virtual SID. Whenever the virtual copy of Π needs to use $\mathcal{F}$, the ephemeral PKI is used to implement it. The idea is that if both parties end up using the same virtual SID, they will effectively be joined, and otherwise, they will be isolated.

Apart from the ephemeral PKI and the name mangling, the construction is not much different from that in §12.2. This construction can be used to get interesting and quite practical authentication and key exchange protocols, based on passwords or other types of credentials (see [CCGS10]).

References

[Bar05] Boaz Barak. How to play almost any mental game over the net - Concurrent composition via super-polynomial simulation. In *46th Annual Symposium on Foundations of Computer Science*, pages 543–552, Pittsburgh, PA, USA, October 23–25, 2005. IEEE Computer Society Press.

[BCD+09] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *FC 2009: 13th International Conference on Financial Cryptography and Data Security*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343, Accra Beach, Barbados, February 23–26, 2009. Springer, Berlin, Germany.

[BCL+05] Boaz Barak, Ran Canetti, Yehuda Lindell, Rafael Pass, and Tal Rabin. Secure computation without authentication. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 361–377, Santa Barbara, CA, USA, August 14–18, 2005. Springer, Berlin, Germany.

[Bea92] Donald Beaver. Foundations of secure interactive computing. In Joan Feigenbaum, editor, *Advances in Cryptology – CRYPTO'91*, volume 576 of *Lecture Notes in Computer Science*, pages 377–391, Santa Barbara, CA, USA, August 11–15, 1992. Springer, Berlin, Germany.

[BPW03] Michael Backes, Birgit Pfitzmann, and Michael Waidner. A composable cryptographic library with nested operations. In Sushil Jajodia, Vijayalakshmi Atluri, and Trent Jaeger, editors, *ACM CCS 03: 10th Conference on Computer and Communications Security*, pages 220–230, Washington D.C., USA, October 27–30, 2003. ACM Press.

[BPW04] Michael Backes, Birgit Pfitzmann, and Michael Waidner. A general composition theorem for secure reactive systems. In Moni Naor, editor, *TCC 2004: 1st Theory of Cryptography Conference*, volume 2951 of *Lecture Notes in Computer Science*, pages 336–354, Cambridge, MA, USA, February 19–21, 2004. Springer, Berlin, Germany.

[BPW07] Michael Backes, Birgit Pfitzmann, and Michael Waidner. The reactive simulatability (RSIM) framework for asynchronous systems. *Inf. Comput.*, 205(12):1685–1720, 2007.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, Nevada, USA, October 14–17, 2001. IEEE Computer Society Press.

[Can05] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, December 2005. Full and updated version of [Can01], `http://eprint.iacr.org/`.

[CCGS10] Jan Camenisch, Nathalie Casati, Thomas Groß, and Victor Shoup. Credential authenticated identification and key exchange. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 255–276, Santa Barbara, CA, USA, August 15–19, 2010. Springer, Berlin, Germany.

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007: 4th Theory of Cryptography Conference*, volume 4392 of *Lecture Notes in Computer Science*, pages 61–85, Amsterdam, The Netherlands, February 21–24, 2007. Springer, Berlin, Germany.

[CF01] Ran Canetti and Marc Fischlin. Universally composable commitments. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 19–40, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Berlin, Germany.

[CKY09] Jan Camenisch, Aggelos Kiayias, and Moti Yung. On the portability of generalized schnorr proofs. In Antoine Joux, editor, *Advances in Cryptology – EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 425–442, Cologne, Germany, April 26–30, 2009. Springer, Berlin, Germany.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *34th Annual ACM Symposium on Theory of Computing*, pages 494–503, Montréal, Québec, Canada, May 19–21, 2002. ACM Press.

[CR03] Ran Canetti and Tal Rabin. Universal composition with joint state. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 265–281, Santa Barbara, CA, USA, August 17–21, 2003. Springer, Berlin, Germany.

[GK96] Oded Goldreich and Hugo Krawczyk. On the composition of zero-knowledge proof systems. *SIAM J. Comput.*, 25(1):169–192, 1996.

[GMP+08] Sebastian Gajek, Mark Manulis, Olivier Pereira, Ahmad-Reza Sadeghi, and Jörg Schwenk. Universally composable security analysis of tls. In *ProvSec*, pages 313–327, 2008.

[GMW86] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity and a methodology of cryptographic protocol design. In *27th Annual Symposium on Foundations of Computer Science*, pages 174–187, Toronto, Ontario, Canada, October 27–29, 1986. IEEE Computer Society Press.

[Hås88] Johan Håstad. Solving simultaneous modular equations of low degree. *SIAM J. Comput.*, 17(2):336–341, 1988.

[HMQU05] Dennis Hofheinz, Jörn Müller-Quade, and Dominique Unruh. Polynomial runtime in simulatability definitions. In *CSFW*, pages 156–169, 2005.

[HUMQ09] Dennis Hofheinz, Dominique Unruh, and Jörn Müller-Quade. Polynomial runtime and composability. Cryptology ePrint Archive, Report 2009/023, 2009. `http://eprint.iacr.org/`.

[JL00] Stanislaw Jarecki and Anna Lysyanskaya. Adaptively secure threshold cryptography: Introducing concurrency, removing erasures. In Bart Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 221–242, Bruges, Belgium, May 14–18, 2000. Springer, Berlin, Germany.

[KT09] Ralf Küsters and Max Tuengerthal. Computational soundness for key exchange protocols with symmetric encryption. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *ACM CCS 09: 16th Conference on Computer and Communications Security*, pages 91–100, Chicago, Illinois, USA, November 9–13, 2009. ACM Press.

[Küs06] Ralf Küsters. Simulation-based security with inexhaustible interactive turing machines. In *CSFW*, pages 309–320. IEEE Computer Society, 2006.

[MR92] Silvio Micali and Phillip Rogaway. Secure computation (abstract). In Joan Feigenbaum, editor, *Advances in Cryptology – CRYPTO'91*, volume 576 of *Lecture Notes in Computer Science*, pages 392–404, Santa Barbara, CA, USA, August 11–15, 1992. Springer, Berlin, Germany.

[MR11] Ueli Maurer and Renato Renner. Abstract cryptography. In Bernard Chazelle, editor, *The Second Symposium in Innovations in Computer Science, ICS 2011*, pages 1–21. Tsinghua University Press, January 2011.

[MY04] Philip D. MacKenzie and Ke Yang. On simulation-sound trapdoor commitments. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology – EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 382–400, Interlaken, Switzerland, May 2–6, 2004. Springer, Berlin, Germany.

[PW01] Birgit Pfitzmann and Michael Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *IEEE Symposium on Security and Privacy*, pages 184–, 2001.