iUC: Flexible Universal Composability Made Simple (Full Version)

Jan Camenisch
Dfinity
Zurich, Switzerland
jan@dfinity.org

Stephan Krenn
AIT Austrian Institute of Technology GmbH
Vienna, Austria
stephan.krenn@ait.ac.at

Ralf Küsters, Daniel Rausch
University of Stuttgart
Stuttgart, Germany
ralf.kuesters@sec.uni-stuttgart.de
daniel.rausch@sec.uni-stuttgart.de

***Abstract*—Proving the security of complex protocols is a crucial and very challenging task. A widely used approach for reasoning about such protocols in a modular way is universal composability. Several models for this approach exist, including the UC, GNUC, and IITM models. Despite the wide spread use of the universal composability approach, none of the existing models are fully satisfying. They all lack either soundness, flexibility, or usability. As a result, proofs in the literature are very often formally incorrect, protocols cannot be modeled faithfully, and/or using these models is a burden rather than a help.**

Given this dire state of affairs, the goal of this work is to provide a framework for universal composability which combines soundness, flexibility, and usability. Developing such a security framework is a very difficult and delicate task, as the long history of frameworks for universal composability shows.

As the IITM model appears to be closest to this goal in terms of soundness and flexibility, we build our framework, called iUC, on top of the IITM model. At the core of iUC is a single simple template for specifying essentially arbitrary protocols in a convenient, formally precise, and flexible way, which allows protocol designers to concentrate on the core logic of a protocol. We illustrate the main features of our framework with example functionalities and realizations.

I. Introduction

Universal composability [1], [2] is an important concept for reasoning about the security of protocols in a modular way. It has found wide spread use, not only for the modular design and analysis of cryptographic protocols, but also in other areas, for example for modeling and analyzing OpenStack [3], network time protocols [4], OAuth v2.0 [5], the integrity of file systems [6], as well as privacy in email ecosystems [7].

The idea of universal composability is that one first defines an *ideal protocol* (or ideal functionality) $\mathcal{F}$ which specifies the intended behavior of a protocol/system. For a concrete realization (real protocol) π one then proves that π "behaves just like $\mathcal{F}$" in arbitrary contexts. Therefore, it is ensured that the real protocol enjoys the security and functional properties specified by $\mathcal{F}$.

This project was in part funded by the European Commission through grant agreements n°s 321310 (PERCY) and 644962 (PRISMACLOUD), and by the *Deutsche Forschungsgemeinschaft* (DFG) through Grant KU 1434/9-1.

Several models for universal composability have been proposed in the literature [1], [2], [8]–[15]. Ideally, a framework for universal composability should support a protocol designer in easily creating full, precise, and detailed specifications of various applications and in various adversary models, instead of being an additional obstacle. In particular, such frameworks should satisfy at least the following requirements:

Soundness: This includes the soundness of the framework itself and the general theorems, such as composition theorems, proven in it. Soundness also means that it should be possible to (easily) carry out formally/mathematically correct proofs about applications based on this framework.

Flexibility: The framework must be flexible enough to allow for the precise design and analysis of a wide range of protocols and applications as well as security models, e.g., in terms of corruption, setup assumptions, etc.

Usability: Specifying and analyzing a protocol should be simple and easy. A protocol designer should be able to focus on the core logic of her protocol instead of having to deal with technical details of the framework or repeatedly taking care of recurrent issues, such as modeling standard corruption behavior.

Unfortunately, despite the wide spread use of the universal composability approach, the existing models and frameworks are still unsatisfying in these respects as none meets all of these requirements simultaneously. We discuss this for the currently most relevant models for analyzing both simple and complex protocols, namely the UC, GNUC, and IITM models, with the UC model being currently the predominate model for universal composability.

The UC Model: Canetti's UC model [1] (see [16] for the latest version), together with its extensions for joint and global states [8], [9], are quite *flexible*. Yet, despite multiple changes and updates, frequently issues regarding the *soundness* and *usability* of the model have been pointed out, including the soundness of composition theorems and the formally correct modeling and analysis of protocols, see, e.g., [10], [11], [17], [18]. For example, the model contains several artifacts, such as the need for artificial notifications to the adversary, padding of messages, and exhaustion of machines, which are very hard to get right. In [10, p. 21],

Canetti et al. conclude: "We are not aware of any written proof in the UC framework that actually takes these details into account."

The GNUC Model: The GNUC model [11] aims to be a formally correct alternative to the UC model, i.e., it resolves *soundness* issues of the UC model. In terms of *usability*, this model provides detailed guidance for the protocol designer, including, for example, fixing corruption conventions and using a simpler runtime notion, which causes less problems than in the UC model. However, the GNUC model is not very *flexible* as it imposes many restrictions on the protocol structure and corruption conventions are quite limited, supporting only top-down corruption, such that many interesting protocols and settings cannot directly be modeled. Also, the runtime notion used in the GNUC model involves so-called flow bounds that make the specification of certain ideal functionalities impossible, such as non-interactive versions of functionalities for encryption and digital signatures [1], [18]–[20].

The IITM Model: The IITM model [14] (see [17] for the latest version) is *sound* and *flexible*. It offers a certain degree of *usability* in that protocol designers do not need to worry about technical details which cause problems in other models. For example, the IITM model offers a very general and at the same time simple runtime notion so that protocol designers essentially do not have to care much about runtime issues. Also, it offers a general and easy to use mechanism to address instances of machines. However, this model does not provide design conventions, for example, for dealing with party IDs, sessions, or corruption; all of this is left to the protocol designer to manually specify for each design and analysis task, distracting from the actual core logic of a protocol.

Our contributions: Given this unsatisfying state of affairs, the goal of this paper is to provide a universal composability framework, that is *sound*, *flexible*, and *easy to use*, and hence constitutes a solid framework for designing and analyzing essentially any protocol and application in a modular, universally composable, and sound way. As the IITM model appears to be closest to this goal, we build our framework, called "iUC", on top of the IITM model (and its extension to responsive environments [21], see Section II), where "iUC" stands for "IITM based Universal Composability".

Developing such a security framework is a difficult and very delicate task that takes multiple years if not decades (cf., the many subtle changes in the UC model since it was first published [16]). Indeed, iUC as put forth in this paper is the result of many years of iterations, refinements, and discussions.

At the core of iUC is *one* convenient template that supports protocol designers in specifying arbitrary types of protocols, including ideal protocols and real protocols with shared, joint, and global state as well as subroutines in a precise, intuitive, and compact way. This is made possible by new concepts, including the concept of entities as well as public and private roles.

Our framework, in particular the template and the machinery behind it, improves upon *usability* of the IITM model by providing missing conventions for many of the repetitive aspects of modeling a protocol (e.g., regarding corruption modeling, wiring of machines, addressing mechanisms, session modeling) such that protocol designers can focus on the core logic of their protocols. The framwork comes with a clear and intuitive syntax to facilitate specifications and to allow others to quickly pick up protocol specifications and use them as subroutines in their higher-level protocols.

At the same time, iUC preserves the *flexibility* of the IITM model. For example, protocols can be structured essentially in arbitrary ways, with a flexible way of addressing different parts of protocols. This, in particular, allows for easily modeling shared, joint, and global state, features for which other frameworks need to rework and extend the core models, including new composition theorems, see, e.g., JUC [8], GUC [9], an extension of GUC [22], and GNUC [11]. We solve the tension between flexibility and usability essentially by providing conventions that on the one hand side take off the burden of repetitive and standard specifications and on the other hand side allow for great flexibility and customization, still all in one uniform framework.

We give a complete formal mapping from iUC to the IITM model. In particular, we show how protocols specified with our templates in iUC are mapped to protocols in the sense of the IITM model. This provides precise semantics of protocol specifications and guarantees that all theorems of the IITM model, such as composition theorems, carry over to iUC. Hence, *soundness* is preserved. Moreover, and importantly, as illustrated by our case studies (see Section IV), we can write complete specifications, without sweeping details under the rug, mainly because there are no technical details and artifacts of the underlying model one needs to take care of. This is crucial for writing formally correct proofs. As discussed for the UC model in [10] and mentioned above, there are several model specific technical details and artifacts which for a formally correct proof one has to consider, but no one has done so far. Often the reasoning is on a level of abstraction similar to what actually suffices to do in our framework.

Structure of this paper: In Section II, we briefly recall the IITM model along with the extension to responsive environments as far as relevant for iUC. The iUC framework is then described in Section III, with a case study illustrating and highlighting some features of iUC in Section IV. Section V discusses further applications of iUC for modeling various types of protocols. We conclude in Section VI. Further details are provided in the appendix.

II. Relevant Parts of the IITM Model

The IITM model was first introduced in [14] and revised in [17] with a more general and simpler runtime notion. In [21], the IITM model was extended to handle *responsive environments*, a general concept (see below) that can also

be applied to other models. As mentioned, our framework is based on the IITM model with responsive environments. In this section, we provide a brief overview of those parts of the responsive IITM model that suffice to understand and use the iUC framework. More details are given in Appendix E.

Inexhaustible interactive Turing machines: An inexhaustible interactive Turing machine (IITM or simply ITM) is a probabilistic Turing machine with a number of named input and output tapes which determine how different ITMs are connected in a system of ITMs (see below). There might exist several instances of an ITM, called ITIs, in a run of a system of ITMs. As detailed below, an instance of an ITM M runs in one of two modes: **CheckAddress** and **Compute**. The former is used to address the different instances of an ITM in a run, whereas in the latter the actual computation is performed. In **CheckAddress** mode, deterministic computation is performed with a runtime bounded by a (fixed) polynomial in the length of the security parameter, the current input message, and the current configuration of the machine. The runtime limit in **Compute** is discussed later.

Systems of ITMs: A *system* $\mathcal{Q}$ of ITMs is a set $\mathcal{Q} = \{M_1, \ldots, M_k\}$[1] of ITMs $M_1, \ldots, M_k$, where the way ITMs in this system are *connected* is defined by the names of the tapes of those machines. More specifically, for every tape named t, it is required that at most two of these ITMs have a tape named t, which are then connected via this tape. Tapes in $\mathcal{Q}$ that already connect two machines are called *internal tapes* of $\mathcal{Q}$ and all other (unconnected) tapes are called *external tapes* of $\mathcal{Q}$. External tapes are further grouped into an *I/O interface* and a *network interface* of $\mathcal{Q}$, where tapes in the I/O interface are used to communicate securely and directly with protocols and tapes in the network interface are used to communicate with the adversary on the network.

There are two special tapes, named start and decision, both of which may occur only in one machine. A machine with tape start is called the *master ITM*.

A system $\mathcal{Q}_2$ is said to be *connectable* to a system $\mathcal{Q}_1$ if $\mathcal{Q}_2$ connects to the external tapes of $\mathcal{Q}_1$ only, i.e., tapes with the same name in $\mathcal{Q}_2$ and $\mathcal{Q}_1$ are external tapes of both $\mathcal{Q}_2$ and $\mathcal{Q}_1$. By $\{\mathcal{Q}_1, \mathcal{Q}_2\}$ one denotes the composition of the connectable systems $\mathcal{Q}_1$ and $\mathcal{Q}_2$, defined in the obvious way. Note that $\{\mathcal{Q}_1, \mathcal{Q}_2\}$ again is a system of ITMs as defined above. For example, if $\mathcal{Q}_1 = \{M_1, M_2\}$ and $\mathcal{Q}_2 = \{M_3, M_4, M_5\}$, then $\{\mathcal{Q}_1, \mathcal{Q}_2\} = \{M_1, \ldots, M_5\}$.

Running a system: In a run of a system $\mathcal{Q}$, an unbounded number of instances of each ITM in $\mathcal{Q}$ may be spawned. An instance of a machine, say an instance of M_i in $\mathcal{Q}$, can send a message to an instance of another machine, say M_j, in $\mathcal{Q}$ if and only if M_i and M_j are connected via tapes, in the sense defined above. Which instance of M_j gets to process the message sent by the instance of M_i is determined by running the instances of M_j in **CheckAddress** mode.

More specifically, in a run of a system $\mathcal{Q}(1^\eta)$ with security parameter η, only one ITI is active at any time and all other ITIs wait for new input. The first machine to be activated is the master ITM in $\mathcal{Q}$ by writing the empty message on start;[2] if no master ITM exists, the run of $\mathcal{Q}$ terminates immediately. If a message m is written by some instance of a machine on one of its tapes, say on t (initially, as mentioned, the empty message is written on start), and there is a different machine, say M, in $\mathcal{Q}$ with another tape also named t, then which instance of M gets to process m is decided as follows.

The instances of M are run in **CheckAddress** mode in the order of their creation until one instance accepts m. This instance (if any) then runs in **Compute** mode with input m written on its tape t. If no instance has accepted m, a fresh instance of M is spawned and run in mode **CheckAddress** and if it accepts m, it gets to process m on its tape t in **Compute** mode. Otherwise (if the freshly created instance also does not accept the message), the freshly created instance is deleted again, m is dropped, and the empty message is written on start to trigger the master ITM (of which there might also be several instances, where again their **CheckAddress** is used to decide which one gets to process the message). After running an ITI in mode **CheckAddress**, the configuration is set back to the state it was in before it was run in **CheckAddress**; in this sense, this mode does not change the configuration of a machine.

When an instance of M processes a message in mode **Compute**, it may write at most one message, say m', on one of its tapes, say t', and then stops. If there is another ITM with a tape also named t' in the system, the message m' is delivered to one instance of that ITM on tape t' as described above. If the instance of M stops without outputting a message or there is no other ITM with a tape t', then (an instance of) the master ITM is activated. A run stops as soon as a message is written on decision, no master instance accepted an incoming message, or a master ITI stopped without output in mode **Compute**. The *overall output* of a run is defined to be the one-bit message that is output on decision, or zero if decision was not written to. The probability that the overall output of a run of $\mathcal{Q}(1^\eta)$ is $b \in \{0,1\}$ is denoted by $\Pr[\mathcal{Q}(1^\eta) = b]$, where the probability is taken over the random choices of all ITIs in runs of $\mathcal{Q}$.

Indistinguishability of systems: Two systems that produce overall output 1 with almost the same probability are called indistinguishable: two systems $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are *indistinguishable* ($\mathcal{Q}_1 \equiv \mathcal{Q}_2$) if and only if $|\Pr[\mathcal{Q}_1(1^\eta) = 1] - \Pr[\mathcal{Q}_2(1^\eta) = 1]|$ is negligible in η.

[1] In the notation of the original IITM paper this notation corresponds to $\mathcal{Q} = !\,M_1 \mid \cdots \mid !\,M_k$.

[2] If a system is run with external input, then this input is written on start. Note that the IITM model supports both uniform and non-uniform environments/machines. For ease of presentation, we consider only the uniform setting in this paper, however, our framework also supports the non-uniform setting.

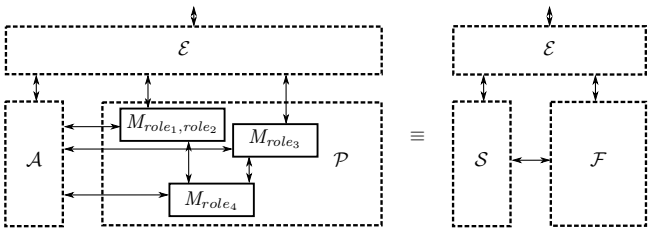


Fig. 1: The setup for the universal composability experiment ($\mathcal{P} \leq \mathcal{F}$) and internal structure of protocols. Here $\mathcal{E}$ is an environmental system, $\mathcal{A}$ and $\mathcal{S}$ are adversarial systems, and $\mathcal{P}$ and $\mathcal{F}$ are protocol systems. Arrows between $\mathcal{E}$ and adversarial systems, and arrows between adversarial systems and protocol systems represent network tapes. All other arrows represent I/O tapes. The boxes M_i in $\mathcal{P}$ are different machines modeling various tasks in the protocol. Note that this is just an example and that all other systems internally also consist of one or more machines.

Types of systems: To define simulation, one distinguishes between *protocol systems*, *adversarial systems*, and *environmental systems*. These are arbitrary systems (in the sense defined above), but only environmental systems may have start and decision tapes; in particular, only the environment may contain the master ITM. There are no other restrictions on these systems. For simulation and universal composability, adversarial systems ($\mathcal{A}$ and $\mathcal{S}$), environmental systems ($\mathcal{E}$), and protocol systems ($\mathcal{P}$ and $\mathcal{F}$) are connected as illustrated in Figure 1. In the IITM model neither any specific internal structure of $\mathcal{P}$ or $\mathcal{F}$ nor any specific addressing mechanism or corruption behavior is fixed; $\mathcal{P}$ and $\mathcal{F}$ are arbitrary systems which can be freely specified by the protocol designer. (Note that Figure 1 contains merely examples of how $\mathcal{P}$ and $\mathcal{F}$ could look like internally.)

Responsiveness of environments and adversaries: In the specifications of protocols, it is often required for the adversary/environment to provide to the protocol some modeling related (meta-) information or to receive some (meta-) information, such as the initial corruption status of protocol instances. Protocols typically exchange this information via the network interface, with the protocol sending some message/request to the adversary (or the environment). As discussed in [21], it is often natural to expect the adversary to send an immediate response to such requests, as otherwise one has to deal with additional artificial complications in protocol specifications and security proofs. For this reason, [21] introduced the concept of responsive environments and proposed an extension of the IITM model. Informally, if a protocol sends what is called a *restricting message*, then both the adversary and environment are forced to send an immediate response. We provide further technical details on responsive environments and adversaries and on how restricting messages are formally defined in Appendix E.

Runtime requirements for environmental and protocol systems: Compared to other frameworks, the IITM model uses very general and simple runtime notions. More specifically, for the simulation notions, we have the following requirements for the runtime of environmental, adversarial, and protocol systems. An environmental system $\mathcal{E}$ has to be *universally bounded*, i.e., there exists a polynomial p such that for every system $\mathcal{Q}$ connectable to $\mathcal{E}$ the overall runtime of $\mathcal{E}$ in mode **Compute** is bounded by $p(\eta)$ in every run of $\{\mathcal{E}, \mathcal{Q}\}$ with security parameter η. Given a system $\mathcal{Q}$, $\mathsf{Env}_R(\mathcal{Q})$ denotes the set of all universally bounded environmental systems that can be connected to $\mathcal{Q}$ and are responsive for $\mathcal{Q}$. A protocol system $\mathcal{P}$ has to be *R-environmentally bounded*, i.e., for every $\mathcal{E} \in \mathsf{Env}_R(\mathcal{P})$ there exists a polynomial p such that for every η the overall runtime of $\mathcal{P}$ in mode **Compute** is bounded by $p(\eta)$ in every run of $\{\mathcal{E}, \mathcal{P}\}$ with security parameter η, except for a negligible set of runs. An adversarial system $\mathcal{A}$ for protocol system $\mathcal{P}$ must satisfy that the combined system $\{\mathcal{A}, \mathcal{P}\}$ is R-environmentally bounded. We define the set $\mathsf{Adv}_R(\mathcal{P})$ to contain all such adversarial systems for $\mathcal{P}$ that, in addition, are responsive and connect only to the network interface of $\mathcal{P}$. Note that the dummy adversary, which simply forwards messages between $\mathcal{P}$ and the environment, always belongs to this set.

Simulation and universal composability: We can now define what it means for a protocol $\mathcal{P}$ to realize/emulate another protocol $\mathcal{F}$: $\mathcal{P}$ *realizes* or *emulates* $\mathcal{F}$, denoted by $\mathcal{P} \leq \mathcal{F}$, if and only if both protocols have the same I/O interfaces and for all $\mathcal{A} \in \mathsf{Adv}_R(\mathcal{P})$ there exists $\mathcal{S} \in \mathsf{Adv}_R(\mathcal{F})$ such that for all $\mathcal{E} \in \mathsf{Env}_R(\{\mathcal{A}, \mathcal{P}\})$ it is the case that $\{\mathcal{E}, \mathcal{A}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$ (cf. Figure 1). Intuitively, $\mathcal{F}$ usually is a so-called *ideal protocol* or *ideal functionality* which specifies a task in an ideal and perfectly secure way, whereas $\mathcal{P}$ usually is a so-called *real protocol* which tries to realize this task in a real setting. If $\mathcal{P} \leq \mathcal{F}$, then for all attacks on $\mathcal{P}$ there is one on $\mathcal{F}$ such that both attacks are indistinguishable for any environment, and hence $\mathcal{P}$ is as secure as $\mathcal{F}$ where the latter is secure by definition. Due to this intuition, one often refers to the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}\}$ as the *real world* system, and $\{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$ as the *ideal world* system.

In the IITM model, also the simulation notions dummy UC, strong simulatability, black-box simulatability, and reactive simulatability have been formulated and shown to be equivalent to the above notion [17], [21], which is an important sanity check for a UC-like model. In particular, *strong simulatability* is a much simpler notion that we will thus use in the following: $\mathcal{P}$ *(strongly) realizes* or *emulates* $\mathcal{F}$, denoted by $\mathcal{P} \leq \mathcal{F}$, if and only if both protocols have the same I/O interfaces and there exists $\mathcal{S} \in \mathsf{Adv}_R(\mathcal{F})$ such that for all $\mathcal{E} \in \mathsf{Env}_R(\mathcal{P})$ it is the case that $\{\mathcal{E}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$. Note that in this case $\mathcal{E}$ may connect to both the network and the I/O interface of $\mathcal{P}$.

Composition theorems: The core of every universal composability model are the composition theorems. The IITM model comes with two general composition theorems.

The first composition theorem handles concurrent composition of any (fixed) number of potentially different protocols. It says that the ideal protocols can be replaced by the real ones:

Theorem 1. [21] Let $\mathcal{Q}$ be a protocol, and $\mathcal{P}, \mathcal{F}$ be protocols such that $\mathcal{P} \leq \mathcal{F}$. If $\{\mathcal{Q}, \mathcal{P}\}$ is R-environmentally bounded, then:

$$\{\mathcal{Q}, \mathcal{P}\} \leq \{\mathcal{Q}, \mathcal{F}\} .$$

This theorem also immediately implies joint-state and global state theorems in the IITM model [17], [18]. The other composition theorem guarantees the secure composition of an unbounded number of sessions of the same protocol system, given that a single session of the protocol system in isolation is secure [21]. Both theorems can be combined to securely compose increasingly complex protocols. In particular, if a single session of $\mathcal{P}$ realizes a single session of $\mathcal{F}$, then the composition theorems imply that $\mathcal{Q}$ using multiple sessions of $\mathcal{P}$ realizes $\mathcal{Q}$ using multiple sessions of $\mathcal{F}$. We provide further details on the composition theorems in Appendix E.

III. The iUC Framework

In this section, we present the iUC framework. We start by explaining the general structure of protocols in §III-A, with corruption explained in §III-B. We then present our template in §III-C. In §III-D, we explain how protocol specifications can be composed in iUC to create new, more complex protocol specification. Finally, in §III-E, we present the realization relation and the composition theorem of iUC. As mentioned, concrete examples are given in our case study (cf. §IV). We provide a precise mapping from iUC to the underlying IITM model in Appendix F, which is crucial to verify that our framework inherits soundness and all theorems of the IITM model. We note, however, that it is not necessary to read this technical mapping to be able to use our framework. The abstraction level provided by iUC is entirely sufficient to understand and use this framework.

A. Structure of Protocols

A protocol $\mathcal{P}$ in our framework is specified via a system of machines $\{M_1, \ldots, M_l\}$. Each machine M_i implements one or more roles of the protocol, where a role describes a piece of code that performs a specific task. For example, a (real) protocol $\mathcal{P}_{\text{sig}}$ for digital signatures might contain a `signer` role for signing messages and a `verifier` role for verifying signatures. In a run of a protocol, there can be several instances of every machine, interacting with each other (and the environment) via I/O interfaces and interacting with the adversary (and possibly the environment) via network interfaces. An instance of a machine M_i manages one or more so-called *entities*. An entity is identified by a tuple $(pid, sid, role)$ and describes a specific party with party ID (PID) *pid* running in a session with session ID (SID) *sid* and executing some code defined by the role *role* where this role has to be (one of) the role(s) of M_i according to the specification of M_i. Entities can send messages to and receive messages from other entities and the adversary using the I/O and network interfaces of their respective machine instances. In the following, we explain each of these parts in more detail, including roles and entities; we also provide examples of the static and dynamic structure of various protocols in Figure 2.

Roles: As already mentioned, a role is a piece of code that performs a specific task in a protocol $\mathcal{P}$. Every role in $\mathcal{P}$ is implemented by a single unique machine M_i, but one machine can implement more than one role. This is useful for sharing state between several roles: for example, consider an ideal functionality $\mathcal{F}_{\text{sig}}$ for digital signatures consisting of a `signer` and a `verifier` role. Such an ideal protocol usually stores all messages signed by the `signer` role in some global set that the `verifier` role can then use to prevent forgery. To share such a set between roles, both roles must run on the same (instance of a) machine, i.e., $\mathcal{F}_{\text{sig}}$ generally consists of a single machine $M_{\texttt{signer,verifier}}$ implementing both roles. In contrast, the real protocol $\mathcal{P}_{\text{sig}}$ uses two machines $M_{\texttt{signer}}$ and $M_{\texttt{verifier}}$ as those roles do not and cannot directly share state in a real implementation (cf. left side of Figure 2). Machines provide an I/O interface and a network interface for every role that they implement such that (entities in) this role can receive and send messages. For addressing purposes, we assume that each role in $\mathcal{P}$ has a unique name. Thus, we can use the role name for calling a specific piece of code, i.e., sending a message to the correct machine.

Public and private roles: We, in addition, introduce the concept of public and private roles, which, as we will see, is a very powerful tool. Every role of a protocol $\mathcal{P}$ is either *private* or *public*. Intuitively, a private role can be called/used only internally by other roles of $\mathcal{P}$ whereas a public role can be called/used by any protocol and the environment. Thus, private roles provide their functionality only internally within $\mathcal{P}$, whereas public roles provide their functionality also to other protocols and the environment. More precisely, a private role connects via its I/O interface only to other roles in $\mathcal{P}$ such that only those roles can send messages to and receive messages from a private role; a public role additionally provides an I/O interface for arbitrary other protocols and the environment such that they can also send messages to and receive messages from a public role. We illustrate the concept of public and private roles by man example below.

Using other protocols as subroutines: Protocols can be combined to construct new, more complex protocols. Intuitively, two protocols $\mathcal{P}$ and $\mathcal{R}$ can be combined if they connect to each other only via their public roles. (We give a formal definition of connectable protocols in §III-D.) The new combined protocol $\mathcal{Q}$ consists of all roles of $\mathcal{P}$ and $\mathcal{R}$, where private roles remain private while public roles can be either public or private in $\mathcal{Q}$; this is up to the protocol designer to decide. To keep role names unique within $\mathcal{Q}$, even if the same role name was used in both $\mathcal{P}$ and $\mathcal{R}$, we (implicitly) assume that role names are prefixed with the

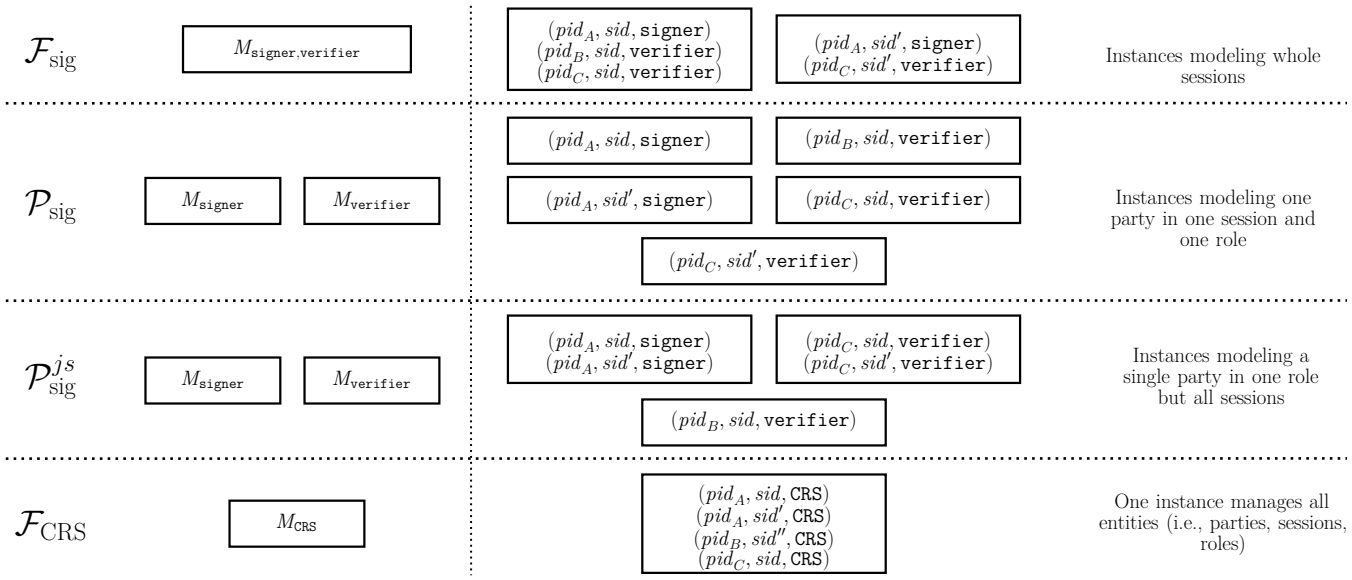
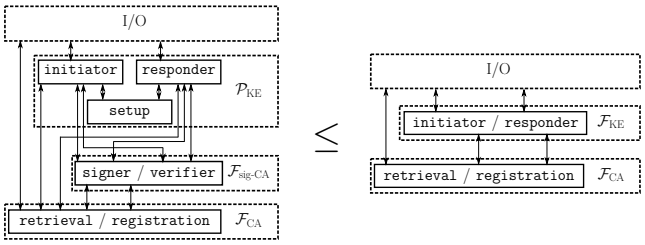


Fig. 2: Examples of static and dynamic structures of various protocol types. $\mathcal{F}_{\text{sig}}$ is an ideal protocol, $\mathcal{P}_{\text{sig}}$ a real protocol, $\mathcal{P}^{js}_{\text{sig}}$ a so-called joint-state realization, and $\mathcal{F}_{\text{CRS}}$ a global state protocol. On the left-hand side: static structures, i.e., (specifications of) machines/protocols. On the right-hand side: possible dynamic structures (i.e., several machine instances managing various entities)



(a) Real world (b) Ideal world

Fig. 3: The static structures of the ideal key exchange functionality $\mathcal{F}_{\text{KE}}$ (right side) and its realization $\mathcal{P}_{\text{KE}}$ (left side), including their subroutines, in our case study. Arrows denote direct I/O connections; network connections are omitted for simplicity. Solid boxes denote individual machines, dotted boxes denote (sub-)protocols that are specified by one instance of our template each (cf. §III-C).

name of their original protocol. We will often also explicitly write down this prefix in the protocol specification for better readability (cf. §III-C).

Examples illustrating the above concepts: Figure 3a, which is further explained in our case study (cf. §IV), illustrates the structure of the protocols we use to model a real key exchange protocol. This protocol as a whole forms a protocol in the above sense and at the same time consists of three separate (sub-) protocols: The highest-level protocol $\mathcal{P}_{\text{KE}}$ has two public roles `initiator` and `responder` executing the actual key exchange and one private role `setup` that generates some global system parameters. The protocol $\mathcal{P}_{\text{KE}}$ uses two other protocols as subroutines, namely the ideal functionality $\mathcal{F}_{\text{sig-CA}}$ for digital signatures with roles `signer` and `verifier`, for signing and verifying messages, and an ideal functionality $\mathcal{F}_{\text{CA}}$ for certificate authorities with roles `registration` and `retrieval`, for registering and retrieving public keys (public key infrastructure). Now, in the context of the combined key exchange protocol, the `registration` role of $\mathcal{F}_{\text{CA}}$ is private as it should be used by $\mathcal{F}_{\text{sig-CA}}$ only; if everyone could register keys, then it would not be possible to give any security guarantees in the key exchange. The `retrieval` role of $\mathcal{F}_{\text{CA}}$ remains public, modeling that public keys are generally considered to be known to everyone, so not only $\mathcal{P}_{\text{KE}}$ but also the environment (and possibly other protocols later using $\mathcal{P}_{\text{KE}}$) should be able to access those keys. This models so-called global state. Similarly to role `registration`, the `signer` role of $\mathcal{F}_{\text{sig-CA}}$ is private too. For simplicity of presentation, we made the `verifier` role private, although it could be made public. Note, however, that the environment/adversary knows the verification key in any case. Altogether, with the concept of public and private roles, we can easily decide whether we want to model global state or make parts of a machine globally available while others remain local subroutines. We can even change globally available roles to be only locally available in the context of a new combined protocol.

As it is important to specify which roles of a (potentially combined) protocol are public and which ones are private, we introduce a simple notation for this. We write $(role_1, \ldots, role_n \mid role_{n+1}, \ldots, role_m)$ to denote a protocol $\mathcal{P}$ with public roles $role_1, \ldots, role_n$ and private roles $role_{n+1}, \ldots, role_m$. If there are no private roles, we just write $(role_1, \ldots, role_n)$, i.e., we omit the "|". Using this notation, the example key exchange protocol from Figure 3a

can be written as (`initiator`, `responder`, `retrieval` | `setup`, `signer`, `verifier`, `registration`).

Entities and Instances: As mentioned before, in a run of a protocol there can be several instances of every protocol machine, and every instance of a protocol machine can manage one or more, what we call, *entities*. Recall that an entity is identified by a tuple $(pid, sid, role)$, which represents party pid running in a session with SID sid and executing some code defined by the role $role$. As also mentioned, such an entity can be managed by an instance of a machine only if this machine implements $role$. We note that sid does not necessarily identify a protocol session in a classical sense. The general purpose is to identify multiple instantiations of the role $role$ executed by party pid. In particular, entities with different SIDs may very well interact with each other, if so desired, unlike in many other frameworks.

The novel concept of entities allows for easily customizing the interpretation of a machine instance by managing appropriate sets of entities. An important property of entities managed by the same instance is that they have access to the same internal state, i.e., they can share state; entities managed by different instances cannot access each others internal state directly. This property is usually the main factor for deciding which entities should be managed in the same instance. With this concept of entities, we obtain a *single* definitional framework for modeling various types of protocols and protocol components in a uniform way, as illustrated by the examples in Figure 2, explained next.

One instance of an ideal protocol in the literature, such as a signature functionality $\mathcal{F}_{\text{sig}}$, often models a single session of a protocol. In particular, such an instance contains all entities for all parties and all roles of one session. Figure 2 shows two instances of the machine $M_{\texttt{signer,verifier}}$, managing sessions sid and sid', respectively. In contrast, instances of real protocols in the literature, such as the realization $\mathcal{P}_{\text{sig}}$ of $\mathcal{F}_{\text{sig}}$, often model a single party in a single session of a single role, i.e., every instance manages just a single unique entity, as also illustrated in Figure 2. If, instead, we want to model one global common reference string (CRS), for example, we have one instance of a machine $M_{\texttt{CRS}}$ which manages all entities, for all sessions, parties, and roles. To give another example, the literature also considers so-called joint-state realizations [8], [18] where a party re-uses some state, such as a cryptographic key, in multiple sessions. An instance of such a joint-state realization thus contains entities for a single party in one role and in all sessions. Figure 2 shows an example joint-state realization $\mathcal{P}^{js}_{\text{sig}}$ of $\mathcal{F}_{\text{sig}}$ where a party uses the same signing key in all sessions. As illustrated by these examples, instances model different things depending on the entities they manage.

Exchanging messages: Entities can send and receive messages using the I/O and network interfaces belonging to their respective roles. When an entity sends a message it has to specify the receiver, which is either the adversary in case of the network interface or some other entity in case of the I/O interface. If a message is sent to another entity $(pid_{rcv}, sid_{rcv}, role_{rcv})$, then the message is sent to the machine M implementing $role_{rcv}$; the **CheckAddress** mode (see Section II) is then used to find the instance of M that manages $(pid_{rcv}, sid_{rcv}, role_{rcv})$. When an entity $(pid_{rcv}, sid_{rcv}, role_{rcv})$ receives a message on the I/O interface, i.e., from another entity $(pid_{snd}, sid_{snd}, role_{snd})$, then the receiver learns pid_{snd}, sid_{snd}[3] and either the actual role name $role_{snd}$ (if the sender is a known subroutine of the receiver, cf. §III-C) or some number $i \in \mathbb{N}$ denoting a specific I/O connection to some sender role (if the sender is an unknown higher-level protocol or the environment). The latter models that a receiver/subroutine does not necessarily know the exact machine code of a caller in some arbitrary higher-level protocol, but the receiver can at least address the caller for sending a response. If a message is received from the network interface, then the receiving entity learns only that it was sent from the adversary.

We note that we do not restrict which entities can communicate with each other as long as their roles are connected, i.e., entities need not share the same SID or PID to communicate via an I/O connection. This, for example, facilitates modeling entities in different sessions using the same resource, as illustrated in our case studies. It, for example, also allows us to model the global functionality $\mathcal{F}_{\text{CRS}}$ from Figure 2 in the following natural way: $\mathcal{F}_{\text{CRS}}$ could manage only a single (dummy) entity $(\epsilon, \epsilon, \texttt{CRS})$ in one machine instance, which can be accessed by all entities of higher-level protocols.

B. Corruption of Protocols

We now explain on an abstract level how our framework models corruption of entities. In §III-C, we then explain in detail how particular aspects of the corruption model are specified and implemented. Our framework supports five different modes of corruption: *incorruptible, static corruption, dynamic corruption with/without secure erasures,* and *custom corruption*. Incorruptible protocols do not allow the adversary to corrupt any entities; this can, e.g., be used to model setup assumptions such as common reference strings which should not be controllable by an adversary. Static corruption allows adversaries to corrupt entities when they are first created, but not later on, whereas dynamic corruption allows for corruption at arbitrary points in time. In the case of dynamic corruption, one can additionally choose whether by default only the current internal state (known as dynamic corruption *with secure erasures*) or also a history of the entire state, including all messages and internal random coins (known as dynamic corruption *without secure erasures*) is given to the adversary upon corruption. Finally, custom corruption is a special case that allows a protocol designer to disable corruption handling of our framework and instead define her own corruption model while still taking advantage of our template and the

[3]The environment can claim arbitrary PIDs and SIDs as sender.

defaults that we provide; we will ignore this custom case in the following description.

To corrupt an entity $(pid, sid, role)$ in a run, the adversary can send the special message corrupt on the network interface to that entity. Note that, depending on the corruption model, such a request might automatically be rejected (e.g., because the entity is part of an incorruptible protocol). In addition to this automatic check, protocol designers are also able to specify arbitrary other conditions that must be met for a corrupt request to be accepted. For example, one could require that all subroutines must be corrupted before a corruption request is accepted, modeling that an adversary must corrupt the entire protocol stack running on some computer instead of just individual programs, which is often easier to analyze.

If a corrupt request for some entity $(pid, sid, role)$ passes all checks and is accepted, the entity is considered *explicitly corrupted* for the rest of the protocol run. The adversary gains full control over explicitly corrupted entities: messages arriving on the I/O interface of $(pid, sid, role)$ are forwarded on the network interface to the adversary, while the adversary can tell $(pid, sid, role)$ (via its network interface) to send messages to arbitrary other entities on behalf of the corrupted entity (as long as both entities have connected I/O interfaces). The protocol designer can control which messages the adversary can send in the name of a corrupted instance, e.g., to prevent him from accessing uncorrupted instances or from communicating between (disjoint) sessions, as detailed in Section III-C.

In addition to the corruption mechanism described above, entities that are activated for the first time also determine their initial corruption status by actively asking the adversary whether he wants to corrupt them. More precisely, once an entity $(pid, sid, role)$ has finished its initialization in mode **Compute** (see Section III-C), it asks the adversary via a *restricting message*[4] whether he wants to corrupt $(pid, sid, role)$ before performing any other computations. The answer of the adversary is processed as discussed before, i.e., the entity decides whether to accept or reject a corruption request. This gives the adversary the power to corrupt new entities right from the start, if he desires; note that in the case of static corruption, this is also the last point in time where an adversary can explicitly corrupt $(pid, sid, role)$.

For modeling purposes, we allow other entities and the environment to obtain the current corruption status of an entity $(pid, sid, role)$.[5] This is done by sending a special `CorruptionStatus`? request on the I/O interface of $(pid, sid, role)$. If $(pid, sid, role)$ has been explicitly corrupted by the adversary, the entity returns `true` immediately. Otherwise, the entity is free to decide whether `true` or `false` is returned, i.e., whether it considers itself corrupted nevertheless. For example, a higher level protocol might consider itself corrupted if at least one of its subroutines is (explicitly or implicitly) corrupted, which models that no security guarantees can be given if certain subroutines are controlled by the adversary. To figure out whether subroutines are corrupted, a higher level protocol can send `CorruptionStatus`? requests to subroutines itself. We call an entity that was not explicitly corrupted but still returns `true` *implicitly corrupted*. We note that the responses to `CorruptionStatus`? request are guaranteed to be consistent in the sense that if an entity returns `true` once, it will always return `true`. Also, according to the defaults of our framework, `CorruptionStatus`? request are answered immediately (without intervention of the adversary) and processing these requests does not change state. These are important features which allow for a smooth handling of corruption.

C. Specifying Protocols

We now present our template for fully specifying a protocol $\mathcal{Q}$, including its uncorrupted behavior, its corruption model, and its connections to other protocols. As mentioned previously, the template is sufficiently general to capture many different types of protocols (real, ideal, hybrid, joint-state, global, ...) and includes several optional parts with reasonable defaults. Thus, our template combines freedom with ease of specification.

The template is given in Figure 4. Some parts are self-explanatory; the other parts are described in more detail in the following.

The first section of the template specifies properties of the whole protocol that apply to all machines.

Participating roles*:* This list of sets of roles specifies which roles are (jointly) implemented by a machine. For example, the list "$\{role_1, role_2\}$, $role_3$, $\{role_4, role_5, role_6\}$" specifies a protocol $\mathcal{Q}$ consisting of three machines $M_{role_1,role_2}$, M_{role_3}, and $M_{role_4,role_5,role_6}$, where $M_{role_1,role_2}$ implements $role_1$ and $role_2$, and so on.

Corruption model*:* This fixes one of the default corruption models supported by iUC, as explained in §III-B: *incorruptible*, *static*, *dynamic with erasures*, and *dynamic without erasures*. Moreover, if the corruption model is set to *custom*, the protocol designer has to manually define his own corruption model and process corruption related messages, such as `CorruptionStatus`?, using the algorithms **MessagePreprocessing** and/or **Main** (see below), providing full flexibility.

Apart from the protocol setup, one has to specify each procotol machine M_i, and hence, the behavior of each set of roles listed in the protocol setup.

Subroutines*:* Here the protocol designer lists all roles that M_i uses as subroutines. These roles may be part of this or potentially other protocols, but may not include roles that are implemented by M_i. The I/O interface of

[4]Recall from Section II that by sending a restricting message, the adversary is forced to answer, and hence, decide upon corruption, right away, before he can interact in any other way with the protocol, preventing artificial interference with the protocol run.

[5]This is particularly important for the realization relation: The environment needs a way to check that the simulator simulates the corruption states of the real world correctly. Otherwise, the simulator could just corrupt all entities in the ideal world, which would allow for a trivial simulation of arbitrary protocols.

Setup for the protocol $\mathcal{Q} = \{M_1, \ldots, M_n\}$:

Participating roles: list of all n sets of roles participating in this protocol. Each set corresponds to one machine M_i.
Corruption model: incorruptible, static, dynamic with/without erasures, custom.
Protocol parameters*: e.g., externally provided algorithms parametrizing a machine.

Implementation of M_i for each set of roles:

Implemented role(s): the set of roles that is implemented by this machine.
Subroutines*: a list of all (other) roles that this machine uses as subroutines.
Internal state*: state variables used to store data across different invocations.
CheckID*: algorithm for deciding whether this machine is responsible for an entity $(pid, sid, role)$.
Corruption behavior*: description of **DetermineCorrStatus**, **AllowCorruption**, **LeakedData**, and/or **AllowAdvMessage** algorithms.
Initialization*: this block is executed only the first time an instance of the machine accepts a message; useful to, e.g., assign initial values that are globally used for all entities managed by this instance.
EntityInitialization*: this block is executed only the first time that some message for a (new) entity is received; useful to, e.g., assign initial values that are specific for single entities.
MessagePreprocessing*: this algorithm is executed every time a message for an uncorrupted entity is received.
Main: specification of the actual behavior of an uncorrupted entity.

Fig. 4: Template for specifying protocols. Blocks labeled with an asterisk (*) are optional. **CheckID** is part of the **CheckAddress** mode, whereas **Corruption behavior**, ..., **Main** are all executed within the **Compute** mode of the machine.

(all roles of) the machine M_i will then be connected to the I/O interfaces of those roles, allowing M_i to access those subroutines.[6] We note that (subroutine) roles are uniquely specified by their name since we assume globally unique names for each role.

If roles of some other protocol $\mathcal{R}$ are used, then protocol authors should prefix the roles with the protocol name to improve readability, e.g., "$\mathcal{R}$: `roleInR`" to denote a connection to the role `roleInR` in the protocol $\mathcal{R}$. This is mandatory if the same role name is used in several protocols to avoid ambiguity. If a machine is supposed to connect to all roles of some protocol $\mathcal{R}$, then, as a short-hand notation, one can list the name $\mathcal{R}$ of the protocol instead.

Internal state*:* State variables declared here (henceforth denoted by sans-serif fonts, e.g., a, b) preserve their values across different activations of an instance of M_i.

In addition to these user-specified state variables, every machine has some additional framework-specific state variables that are set and changed automatically according to our conventions. Most of these variables are for internal bookkeeping and need not be accessed by protocol designers. Those that might be useful in certain algorithms are mentioned and explained further below (we provide a complete list of all framework specific variables in Appendix F-B).

CheckID*:* As mentioned before, instances of machines in our framework manage (potentially several) entities $(pid_i, sid_i, role_i)$. The algorithm **CheckID** allows an instance of a machine to decide which of those entities are accepted and thus managed by that instance, and which are not. Furthermore, it allows for imposing a certain structure on pid_i and sid_i; for example, SIDs might only be accepted if they encode certain session parameters, e.g., $sid_i = (parameter_1, parameter_2, sid_i')$.

[6]We emphasize that we do not put any restrictions on the graph that the subroutine relationships of machines of several protocols form. For example, it is entirely possible to have machines in two different protocols that specify each other as subroutines.

More precisely, the algorithm **CheckID**$(pid, sid, role)$ is a *deterministic algorithm* (computing on the input $(pid, sid, role)$, internal state, and security parameter) running in *polynomial time* (in the length of the current input, internal state, and security parameter) that outputs `accept` or `reject`. Every time a machine instance is invoked with a message m for some entity $(pid, sid, role)$, it runs **CheckID**$(pid, sid, role)$ in **CheckAddress** mode to determine whether it manages $(pid, sid, role)$, i.e., whether the message m should be accepted. We require that **CheckID** behaves consistently, i.e., it never accepts an entity that has previously been rejected, and it never rejects an entity that has previously been accepted; this ensures that there are no two instances that manage the same entity. We note that **CheckID** cannot change the (internal) state of the instance; all changes are dropped after generating an output.[7] In Appendix A we provide a simple syntax for easily specifying the most common cases in the **CheckID** algorithm.

If **CheckID** is not specified, its default behavior is as follows: Given input $(pid, sid, role)$, if the machine instance in which **CheckID** is running has not accepted an entity yet, it outputs `accept`. If it has already accepted another entity $(pid', sid', role')$, then it outputs `accept` iff $pid = pid'$ and $sid = sid'$. Otherwise, it outputs `reject`. Thus, by default, a machine instance accepts, and hence, manages, not more than one entity per role for the roles the machine implements.

Corruption behavior*:* This element of the template allows for customization of corruption related behavior of machines by specifying one or more of the optional algorithms **DetermineCorrStatus**, **AllowCorruption**, **LeakedData**, and **AllowAdvMessage**. These algorithms are used to customize our corruption model, as explained in §III-B (cf. that section for possible use cases).

[7]This is because **CheckID** is part of mode **CheckAddress** which resets all state changes after it has finished its computation.

As these algorithms are part of our corruption conventions, they are used only if **Corruption model** is not set to *custom*. A detailed description of the purpose of every algorithm, including its default behavior if not specified, is given below.

The **DetermineCorrStatus**$(pid, sid, role)$ algorithm is used to customize the response to `CorruptionStatus`? requests (recall that other roles/the environment can send this message on the I/O interface to obtain the current corruption status of $(pid, sid, role)$); the algorithm must output either `true` or `false`. More precisely, upon receiving a `CorruptionStatus`? request from a sender for some receiving entity $(pid, sid, role)$, an instance does the following right at the start of mode **Compute**: if $(pid, sid, role)$ has not yet received a message $\neq$ `CorruptionStatus`? (and thus in particular has not yet determined whether it is explicitly corrupted), then $(\texttt{CorruptionStatus}, \texttt{false})$ is sent back to the sender immediately. Otherwise, the instance checks whether $(pid, sid, role)$ has been explicitly corrupted by the adversary. If so, the instance sends $(\texttt{CorruptionStatus}, \texttt{true})$ back to the sender. The same also happens if, at some point in the past, the instance has already responded with `true` for $(pid, sid, role)$ at least once. Finally, in all other cases the **DetermineCorrStatus** algorithm is called to decide whether the instance responds with $(\texttt{CorruptionStatus}, \texttt{true})$ or $(\texttt{CorruptionStatus}, \texttt{false})$. As mentioned previously, this decision might depend on, e.g., the corruption status of subroutines. In particular, **DetermineCorrStatus** might ask for the corruption status of (entities in) subroutines by sending `CorruptionStatus`? to (entities in) these subroutines. We note that `CorruptionStatus`? requests are processed and answered without running any other algorithms, such as **Initialization** or **Main**. This ensures that a `CorruptionStatus`? request does not accidentally change the state of the instance (cf. §III-B). If not specified, the **DetermineCorrStatus** algorithm always returns `false`. We note that *no other actions are performed* after responding to a `CorruptionStatus`? request, such as running the **Initialization** algorithm or asking the adversary for corruption (see below).[8]

The **AllowCorruption**$(pid, sid, role)$ algorithm is used to decide whether an adversary may explicitly corrupt an entity $(pid, sid, role)$; it must output `true` or `false`. More precisely, when an the adversary asks to corrupt some uncorrupted entity $(pid, sid, role)$ (either by sending a corrupt request or when $(pid, sid, role)$ determines its initial corruption status), the entity first checks whether, for the specified corruption mode of the protocol, corruption of this entity is possible at the current point in time, and rejects the request if not. Otherwise, **AllowCorruption** is called to decide whether corruption of that entity is accepted. The decision can, e.g., depend on the corruption states of (entities in) subroutines. If the algorithm outputs `true`, then the entity is henceforth considered explicitly corrupted. The entity is then also added to a framework-specific set CorruptionSet that keeps track of all explicitly corrupted entities managed by the current machine instance; this set can be used by a protocol designer, e.g., to let the behavior of algorithms depend on which entities are explicitly corrupted. If the **AllowCorruption** algorithm is not specified, then the default is to always return `true`, i.e., the adversary is not further restricted in which entities he may corrupt.

The **LeakedData** algorithm is used to determine the data that is leaked to the adversary upon successful (explicit) corruption of an entity; it outputs an arbitrary bit string. Formally, this algorithm is called directly after **AllowCorruption** has allowed corruption by outputting `true`; the output of **LeakedData** is then sent as part of a confirmation of successful corruption back to the network. To make the specification of **LeakedData** easier, authors can use the framework specific variable transcript that contains a list of all messages that have been received and sent by the **MessagePreprocessing** and **Main** algorithms (see below) of this instance; in other words, this variable generally contains all messages except for (meta) messages related to initialization and corruption. If **LeakedData** is not specified, the following default behavior is used: If an entity is currently determining its initial corruption status after receiving some (first) message m from some sender *sender* and gets corrupted during this, then $(m, sender)$ is output by **LeakedData**. Thus, the adversary learns all information that this entity ever obtained, modeling that the entity was corrupted before the protocol started. If corruption occurs later on, which is possible only for dynamic corruption modes, then the leakage contains either the **Internal state** (in the case of dynamic corruption with secure erasures) or the **Internal state**, the transcript of all messages transcript, and all random coins that have been used so far (in the case of dynamic corruption without secure erasures). We note that, generally, this default for dynamic corruption is suitable only for instances that manage exactly one entity because the full **Internal state**, transcript, and random coins are leaked.

The **AllowAdvMessage**$(pid, sid, role, pid_{\text{receiver}}, sid_{\text{receiver}}, role_{receiver}, m)$ algorithm is used to decide whether an adversary may send a message m via an explicitly corrupted entity $(pid, sid, role)$ to another entity $(pid_{\text{receiver}}, sid_{\text{receiver}}, role_{\text{receiver}})$ (where $role_{receiver}$ is a role that is connected to the current machine, i.e., it is a subroutine or a higher level protocol). The algorithm must output either `true` or `false`, depending on whether the message is allowed. If a message m is not allowed, then the entity stops the current activation without forwarding m by returning an error message to the adversary. If **AllowAdvMessage** is not specified, it defaults to outputting `true` iff $pid = pid_{\text{receiver}}$. In other words, by default an adversary can interact only with

[8]This ensures that `CorruptionStatus`? requests, by default, do not change the internal state of machines, which would complicate simulations needlessly.

subroutines/higher level protocols in the name of the same party *pid*. This default has been chosen as in most cases we expect protocols to be designed such that parties call other protocols only in their own name; in such a setting, an adversary should not be able to directly use, e.g., subroutines belonging to other (uncorrupted) parties.

Initialization, **EntityInitialization**, **MessagePreprocessing**, **Main**: These algorithms specify the actual behavior of a machine in mode **Compute** for (uncorrupted) entities.

The **Initialization** algorithm is run exactly once per machine instance (*not per entity* in that instance) and is mainly supposed to be used for initializing the internal state of that instance. For example, one can generate global parameters or cryptographic key material in this algorithm.

The **EntityInitialization**$(pid, sid, role)$ algorithm is analogous to **Initialization** but is run once for each entity instead of once for each machine instance. More precisely, it runs directly after a potential execution of **Initialization** if **EntityInitialization** has not been run for the current entity $(pid, sid, role)$ yet. This is particularly useful if a machine instance manages several entities, where not all of them might be known from the beginning.

After the algorithms **Initialization** and, for the current entity, the algorithm **EntityInitialization** have finished (or have been skipped), the current entity determines its initial corruption status (if not done yet) and processes a corrupt request from the network/adversary, if any. Note that this allows for using the initialization algorithms to setup some internal state that can be used by the entity to determine its corruption status.

Finally, after all of the previous steps, if the current entity has not been explicitly corrupted,[9] the algorithms **MessagePreprocessing** and **Main** are run. The **MessagePreprocessing** algorithm is executed first. If it does not end the current activation, **Main** is executed directly afterwards. While we do not fix how authors have to use these algorithms, usually one would use **MessagePreprocessing** to prepare the input m for the **Main** algorithm, e.g., by dropping malformed messages or extracting some key information from m, whereas **Main** would contain the core logic of the protocol.

If any of the optional algorithms are not specified, then they are simply skipped during computation. We provide a convenient syntax for specifying these algorithms in the full version; see our case study in §IV for examples.

This concludes the description of our template. As already mentioned, in Appendix F-B we give a formal mapping of this template to protocols in the sense of the IITM model, which provides a precise semantics for the templates and also allows us to carry over all definitions, such as realization relations, and theorems, such as composition theorems, of the IITM model to iUC (see §III-E).

[9] As mentioned in §III-B, if an entity is explicitly corrupted, it instead acts as a forwarder for messages to and from the adversary.

D. Composing Protocol Specifications

Protocols in our framework can be composed to obtain more complex protocols. More precisely, two protocols $\mathcal{Q}$ and $\mathcal{Q}'$ that are specified using our template are called *connectable* if they connect via their public roles only. That is, if a machine in $\mathcal{Q}$ specifies a subroutine role of $\mathcal{Q}'$, then this subroutine role has to be public in $\mathcal{Q}'$, and vice versa.

Two connectable protocols can be composed to obtain a new protocol $\mathcal{R}$ containing all roles of $\mathcal{Q}$ and $\mathcal{Q}'$ such that the public roles of $\mathcal{R}$ are a subset of the public roles of $\mathcal{Q}$ and $\mathcal{Q}'$. Which potentially public roles of $\mathcal{R}$ are actually declared to be public in $\mathcal{R}$ is up the protocol designer and depends on the type of protocol that is to be modeled (see §III-A and our case study in §IV). In any case, the notation from §III-A of the form $(role_1^{pub} \dots role_i^{pub} \mid role_1^{priv} \dots role_j^{priv})$ should be used for this purpose.

For pairwise connectable protocols $\mathcal{Q}_1, \dots, \mathcal{Q}_n$ we define $\mathsf{Comb}(\mathcal{Q}_1, \dots, \mathcal{Q}_n)$ to be the (finite) set of all protocols $\mathcal{R}$ that can be obtained by connecting $\mathcal{Q}_1, \dots, \mathcal{Q}_n$. Note that all protocols $\mathcal{R}$ in this set differ only by their sets of public roles. We define two shorthand notations for easily specifying the most common types of combined protocols: by $(\mathcal{Q}_1, \dots, \mathcal{Q}_i \mid \mathcal{Q}_{i+1}, \dots, \mathcal{Q}_n)$ we denote the protocol $\mathcal{R} \in \mathsf{Comb}(\mathcal{Q}_1, \dots, \mathcal{Q}_n)$, where the public roles of $\mathcal{Q}_1, \dots, \mathcal{Q}_i$ remain public in $\mathcal{R}$ and all other roles are private. This notation can be mixed with the notation from §III-A in the natural way by replacing a protocol $\mathcal{Q}_j$ with its roles, some of which might be public while others might be private in $\mathcal{R}$. Furthermore, by $\mathcal{Q}_1 \,||\, \mathcal{Q}_2$ we denote the protocol $\mathcal{R} \in \mathsf{Comb}(\mathcal{Q}_1, \mathcal{Q}_2)$ where exactly those public roles of $\mathcal{Q}_1$ and $\mathcal{Q}_2$ remain public that are not used as a subroutine by any machine in $\mathcal{Q}_1$ or $\mathcal{Q}_2$.

We call a protocol $\mathcal{Q}$ *complete* if every subroutine *role* used by a machine in $\mathcal{Q}$ is also part of $\mathcal{Q}$. In other words, $\mathcal{Q}$ fully specifies the behavior of all subroutines. Since security analysis makes sense only for a fully specified protocol, we will (implicitly) consider this to be the default in the following.

E. Realization Relation and Composition Theorems

As already mentioned, by virtue of the mapping from iUC to the IITM model definitions and theorems from the IITM model immediately carry over to iUC. In particular, this is true for the realization relation and the composition theorems. Note that the mapping implies a natural semantics to what it means to run a system consisting of an environment, a protocol (specified in iUC), and possibly an adversary/simulator.

Now, the realization relation in iUC corresponding to the one in the IITM model is defined in the obvious way as follows.

Definition 1 (Realization relation in iUC)**.** Let $\mathcal{P}$ and $\mathcal{F}$ be two R-environmentally bounded complete protocols

with identical sets of public roles.[10] The protocol $\mathcal{P}$ *realizes* $\mathcal{F}$ (denoted by $\mathcal{P} \leq \mathcal{F}$) iff there exists $\mathcal{S} \in \mathsf{Adv}_R(\mathcal{F})$ such that for all $\mathcal{E} \in \mathsf{Env}_R(\mathcal{P})$ it holds true that

$$\{\mathcal{E}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\} .$$

Note that, in iUC terms, $\mathcal{E}$ in $\{\mathcal{E}, \mathcal{P}\}$ connects to the I/O interfaces of public roles as well as the network interfaces of all roles of $\mathcal{P}$. In contrast, $\mathcal{E}$ in the system $\{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$ connects to the I/O interfaces of public roles of $\mathcal{F}$ and the network interface of $\mathcal{S}$. The simulator $\mathcal{S}$ connects to $\mathcal{E}$ (simulating the network interface of $\mathcal{P}$) and the network interface of $\mathcal{F}$.

In iUC, Theorem 1 from the IITM model immediately translates to:

Corollary 2 (Concurrent composition in iUC)**.** Let $\mathcal{P}$ and $\mathcal{F}$ be two protocols such that $\mathcal{P} \leq \mathcal{F}$. Let $\mathcal{Q}$ be another protocol such that $\mathcal{Q}$ and $\mathcal{F}$ are connectable. Let $\mathcal{R} \in \mathsf{Comb}(\mathcal{Q}, \mathcal{P})$ and let $\mathcal{I} \in \mathsf{Comb}(\mathcal{Q}, \mathcal{F})$ such that $\mathcal{R}$ and $\mathcal{I}$ have the same sets of public roles. If $\mathcal{R}$ is R-environmentally bounded and complete, then $\mathcal{R} \leq \mathcal{I}$.

Similarly to the IITM model, we emphasize that this corollary also covers the special cases of protocols with joint-state and global state. Furthermore, a second composition theorem for secure composition of an unbounded number of sessions of a protocol is also available. We discuss various types of composition, including composition for protocols with joint andand/or global state, in more detail in §V.

IV. Case Study

In this section, we illustrate flexibility and usability of our framework by means of a concrete example. More specifically, we model and analyze a key exchange protocol of the ISO/IEC 9798-3 standard [24], depicted in Figure 5.

We emphasize that our specifications of the ISO protocol given in the figures below are formally complete. No unnecessary modeling artifacts, such as artificially padding messages, flow bounds and such need to be taken care of. In particular, security of the ISO protocol can be proven in iUC in a formally sound way. Nothing needs to be swept under the rug (see also §I).

The ISO protocol is an authenticated version of the Diffie-Hellman key exchange protocol. While this protocol has already been analyzed previously in universal composability models (e.g., in [25], [26]), these analyses are either for modified versions of the protocol (as the protocol could not be modeled precisely as deployed in practice) or had to manually define many recurrent modeling related aspects (such as a general corruption model and an interpretation of machine instances), which is not only cumbersome but also hides the core logic of the protocol. Thus, this example is well suited to illustrate that our framework manages to combine both flexibility and ease of use for modeling a real-world protocol precisely as deployed in practice. We

[10] Intuitively, the role names are used to determine which parts of $\mathcal{F}$ are realized by which parts of $\mathcal{P}$.

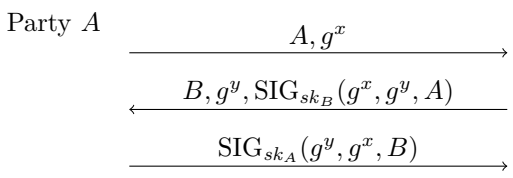


Fig. 5: ISO 9798-3 key exchange protocol for mutual authentication. A and B are the names of two parties that, at the end of the protocol, share a session key g^{xy}.

start by giving a high-level overview of how we model this protocol in §IV-A, then state our security result in §IV-B, and finally discuss features of our modeling in §IV-C.

A. Overview of our Modeling

We model the ISO protocol in a modular way using several smaller protocols. The static structure of all protocols, including their I/O connections for direct communication, is given in Figure 3, which was partly explained already in §III-A. We provide a formal specification of $\mathcal{F}_{\text{CA}}$ using our template in Figure 6. The remaining protocols are specified in Figures 7, 8, 9, and 11 in the appendix. The syntax is mostly self-explanatory, with a formal definition provided in Appendix A. In the following, we give a high-level overview of each protocol.

The ISO key exchange (Figure 5) is modeled as a real protocol $\mathcal{P}_{\text{KE}}$ that uses two ideal functionalities as subroutines: an ideal functionality $\mathcal{F}_{\text{sig-CA}}$ for creating and verifying ideal digital signatures and an ideal functionality $\mathcal{F}_{\text{CA}}$ modeling a certificate authority (CA) that is used to distribute public verification keys generated by $\mathcal{F}_{\text{sig-CA}}$. The real protocol $\mathcal{P}_{\text{KE}}$, as already mentioned in §III-A, consists of three roles, `initiator`, `responder`, and `setup`. The `setup` role models secure generation and distribution of a system parameter, namely, a description of a cyclic group (G, n, g). As this parameter must be shared between all runs of a key exchange protocol, `setup` is implemented by a single machine which spawns a single instance that manages all entities and always outputs the same parameter. The roles `initiator` and `responder` implement parties A and B, respectively, from Figure 5. Each role is implemented by a separate machine and every instance of those machines manages exactly one entity. Thus, these instances directly correspond to an actual implementation where each run of a key exchange protocol spawns a new program instance. We emphasize that two entities can perform a key exchange together even if they do not share the same SID, which models so-called local SIDs (cf. [20]) and is the expected behavior for many real-world protocols; we discuss this feature in more detail below.

During a run of $\mathcal{P}_{\text{KE}}$, entities use the ideal signature functionality $\mathcal{F}_{\text{sig-CA}}$ to sign messages. The ideal functionality $\mathcal{F}_{\text{sig-CA}}$ consists of two roles, `signer` and `verifier`, that allow for the corresponding operations. Both roles are implemented by the same machine and instances of that machine manage entities that share the same SID. The SID

sid of an entity is structured as a tuple (pid_{owner}, sid'), modeling a specific key pair of the party pid_{owner}. More specifically, in protocol $\mathcal{P}_{\text{KE}}$, every party *pid* owns a single key pair, represented by SID (pid, ϵ), and uses this single key pair to sign messages throughout all sessions of the key exchange. Again, this is precisely what is done in reality, where the same signing key is re-used several times. The behavior of $\mathcal{F}_{\text{sig-CA}}$ is closely related to the standard ideal signature functionalities found in the literature (such as [18]), except that public keys are additionally registered with $\mathcal{F}_{\text{CA}}$ when being generated.

As also mentioned in §III-A, the ideal CA functionality $\mathcal{F}_{\text{CA}}$ allows for storing and retrieving public keys. Both roles, `registration` and `retrieval`, are implemented by one machine and a single instance of that machine accepts all entities, as $\mathcal{F}_{\text{CA}}$ has to output the same keys for all sessions and parties. Keys are stored for arbitrary pairs of PIDs and SIDs, where the SID allows for storing different keys for a single party. In our protocol, keys can only be registered by $\mathcal{F}_{\text{sig-CA}}$, and the SID is chosen in a matter that it always has the form (pid, ϵ), denoting the single public key of party *pid*. We emphasize again that arbitrary other protocols and the environment are able to retrieve public keys from $\mathcal{F}_{\text{CA}}$, which models so-called global state.

We model the security properties of a key exchange via an ideal key exchange functionality $\mathcal{F}_{\text{KE}}$. This functionality consists of two roles, `initiator` and `responder`, and uses $\mathcal{F}_{\text{CA}}$ as a subroutine, thus providing the same interfaces (including the public role `retrieval` of $\mathcal{F}_{\text{CA}}$) as $\mathcal{P}_{\text{KE}}$ in the real world. Both `initiator` and `responder` roles are implemented via a single machine, and one instance of this machine manages all entities. This is due to the fact that, at the start of a run, it is not yet clear which entities will interact with each other to form a "session" and perform a key exchange (recall that entities need not share the same SID to do so, i.e., they use locally chosen SIDs, see also §IV-C). Thus, a single instance of $\mathcal{F}_{\text{KE}}$ must manage all entities such that it can internally group entities into appropriate sessions that then obtain the same session key. Formally, the adversary/simulator is allowed to decide which entities are grouped into a session, subject to certain restrictions that ensure the expected security guarantees of a key exchange, including authentication. If two honest entities finish a key exchange in the same session, then $\mathcal{F}_{\text{KE}}$ ensures that they obtain an ideal session key that is unknown to the adversary. The adversary may also use $\mathcal{F}_{\text{KE}}$ to register arbitrary keys in the subroutine $\mathcal{F}_{\text{CA}}$, also for honest parties, i.e., no security guarantees for public keys in $\mathcal{F}_{\text{CA}}$ are provided (see the remark in Figure 11).

B. Security Result

For the above modeling, we obtain the following result, with a proof provided in Appendix B.

Theorem 3. Let $\mathsf{groupGen}(1^\eta)$ be an algorithm that outputs descriptions (G, n, g) of cyclical groups such that n grows exponentially in η and the DDH assumption holds true. Then we have:

$$(\mathcal{P}_{\text{KE}}, \mathcal{F}_{\text{CA}} : \texttt{retrieval} \mid \mathcal{F}_{\text{sig-CA}}, \mathcal{F}_{\text{CA}} : \texttt{registration}) \leq (\mathcal{F}_{\text{KE}}, \mathcal{F}_{\text{CA}} : \texttt{retrieval} \mid \mathcal{F}_{\text{CA}} : \texttt{registration}) .$$

Note that we can realize $\mathcal{F}_{\text{sig-CA}}$ via a generic implementation $\mathcal{P}_{\text{sig-CA}}$ of a digital signature scheme (see Figure 10 in the Appendix):

Lemma 4. If the digital signature scheme used in $\mathcal{P}_{\text{sig-CA}}$ is EUF-CMA-secure, then

$$(\mathcal{P}_{\text{sig-CA}}, \mathcal{F}_{\text{CA}} : \texttt{retrieval} \mid \mathcal{F}_{\text{CA}} : \texttt{registration}) \leq (\mathcal{F}_{\text{sig-CA}}, \mathcal{F}_{\text{CA}} : \texttt{retrieval} \mid \mathcal{F}_{\text{CA}} : \texttt{registration}) .$$

Proof. Analogous to the proof in [18]. □

By Corollary 2, we can thus immediately replace the subroutine $\mathcal{F}_{\text{sig-CA}}$ of $\mathcal{P}_{\text{KE}}$ with its realization $\mathcal{P}_{\text{sig-CA}}$ to obtain an actual implementation of Figure 3 based on an ideal trusted CA:

Corollary 5. If the conditions of Theorem 3 and Lemma 4 are fulfilled, then

$$(\mathcal{P}_{\text{KE}}, \mathcal{F}_{\text{CA}} : \texttt{retrieval} \mid \mathcal{P}_{\text{sig-CA}}, \mathcal{F}_{\text{CA}} : \texttt{registration}) \leq (\mathcal{F}_{\text{KE}}, \mathcal{F}_{\text{CA}} : \texttt{retrieval} \mid \mathcal{F}_{\text{CA}} : \texttt{registration}) .$$

C. Discussion

In the following, we highlight some of the key details of our protocol model where we are able to model reality much more precisely than is possible in other, less flexible universal composability frameworks, including UC and GNUC.

Local SIDs: A special feature of $\mathcal{P}_{\text{KE}}$ (and $\mathcal{F}_{\text{KE}}$) is the use of so-called local session IDs (cf. [20]). That is, the SID of an entity $(pid, sid, role)$ models a value that is locally chosen and managed by each party *pid* and used only for locally addressing a specific instance of a protocol run of that party. In particular, two entities can perform a key exchange with each other (and thus form a "protocol session") even if they use different SIDs. This is in contrast to using so-called pre-established SIDs (or global SIDs), where entities in the same "protocol session" are assumed to already share some globally unique SID that was created prior to the actual protocol run, e.g., by exchanging nonces. Note that establishing such a global SID incurs an additional round trip of communication and thus many real-world protocols, including the key exchange in our case study, in practice use local SIDs instead. While we can easily and faithfully model such real-world behavior, this is not possible at all in most other universal composability models as they require global SIDs. In those models, protocol designers have to resort to analyzing variants of the actual protocols instead. This is not just a minor technicality or a cosmetic difference: as argued by Küsters et al. [20], there are protocols that are insecure when using locally chosen SIDs but become secure if a global SID for all participants in a session has already been established, i.e., security results for protocols with global SIDs do not necessarily carry over to actual implementations using local SIDs.

Description of the protocol $\mathcal{F}_{\text{CA}} = (\texttt{registration}, \texttt{retrieval})$:

Participating roles: $\{\texttt{registration}, \texttt{retrieval}\}$
Corruption model: incorruptible

Description of $M_{\texttt{registration},\texttt{retrieval}}$:

Implemented role(s): $\{\texttt{registration}, \texttt{retrieval}\}$
Internal state:
– $\mathsf{keys} : (\{0,1\}^*)^2 \to \{0,1\}^* \cup \{\bot\}$ {*Mapping from a tuple (PID,SID) to stored keys; initially* $\bot$.

CheckID$(pid, sid, role)$**:** Accept all entities.
Main:

recv $(\texttt{Register}, key)$ **from** $\texttt{I/O}$ **to** $(\_, \_, \texttt{registration})$: { *Allows every higher level protocol that connects to the* `Register` *role to register a key. The key is stored for the PID and SID of the caller of* $\mathcal{F}_{\text{CA}}$.
  if $\mathsf{keys}[\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}] \neq \bot$:
    reply $(\texttt{Register}, \texttt{failed})$.
  else:
    $\mathsf{keys}[\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}] = key$
    reply $(\texttt{Register}, \texttt{success})$.

recv $(\texttt{Retrieve}, (pid, sid))$ **from** $\_$ **to** $(\_, \_, \texttt{retrieval})$: { *Everyone, both* `NET` *and* `I/O`*, can retrieve keys registered by someone with PID pid and SID sid.*
  reply $(\texttt{Retrieve}, \mathsf{keys}[pid, sid])$.

Fig. 6: The ideal CA functionality $\mathcal{F}_{\text{CA}}$ models a public key infrastructure based on a trusted certificate authority.

Shared State: Recall that in $\mathcal{P}_{\text{KE}}$ there is just a single signing key for every party *pid* that is used by every entity $(pid, sid, role)$ in any role of that party. We emphasize that not only is it trivial to model such shared state in our framework, but most other UC-like frameworks cannot model this exact setting at all. Those frameworks assume disjoint protocol sessions by default and thus require different protocol sessions to use different keys. Alternatively, one can resort to a so-called joint-state realization, which has to modify the signed messages by adding a prefix consisting of a globally unique identifier for each key exchange session. Clearly, neither is done in an actual implementation of Figure 3. Thus, one cannot faithfully assess the security of this and many other protocols in those frameworks.

Global State: Our concept of public and private roles allows us to easily model global state, and even in a more general way than existing formulations as we support machines that are only partially global. In particular, our functionality $\mathcal{F}_{\text{CA}}$ is a much more realistic modeling than the existing global functionality $\mathcal{G}_{bb}$ for certificate authorities by Canetti et al. [22]. Since the latter is formulated in an extension of the UC framework, which does not support making parts of a global functionality "private", everyone has full access to all parts of the functionality, including key registration. Thus the environment can register keys for arbitrary parties (recall that the environment in Canetti's UC model can send messages in the name of arbitrary parties and arbitrary sessions, except for the so-called challenge session, in order to simulate arbitrary other protocols). This makes proving security of protocols relying on $\mathcal{G}_{bb}$ hard if not impossible as the environment can impersonate also honest parties.

V. Applying iUC

This section explains how iUC can be used to capture various kinds of protocols and settings, illustrating its flexibility and expressiveness. As part of this, we compare our framework with the UC and GNUC frameworks, showing that iUC is more flexible and can capture a larger variety of protocols and settings.

As part of the section, we also discuss various types of composition, including unbounded self composition, composition with joint-state and composition with global state (recall that concurrent composition has already been presented in §III-E). We emphasize that all of these cases, except for unbounded self composition, are covered directly by the concurrent composition theorem (cf. Corollary 2). Thus *a single theorem* covers most use cases, which illustrates the power and generality of the composition theorem in our framework.

A. Single Session Composition

In addition to concurrent composition (cf. §III-E and Corollary 2), our framework also supports so-called single session composition (also called unbounded self composition). Single session composition works on protocols that are built in such a way that instances of those protocols can be grouped into several disjoint sessions such that instances from different sessions do not share any state and do not interact with each other. For such protocols, the single session composition theorem, intuitively, states that if one session of a protocol $\mathcal{P}$ realizes one session of a protocol $\mathcal{F}$, then an unlimited number of sessions of $\mathcal{P}$ realizes an unlimited number of sessions of $\mathcal{F}$.

By default, our framework does not enforce a protocol structure with disjoint sessions (unlike many other universal composability models that assume disjoint sessions). In the contrary, we do not restrict the protocol designer and instead allow for, e.g., sharing arbitrary state between several protocol sessions by being able to manage several entities in the same instance of a machine. However, if so desired, one can easily define a protocol with disjoint sessions. On a high level, such a protocol has to ensure that (i) no instance accepts entities from different sessions (which makes the state disjoint) and (ii) entities send

messages to other entities only in the same session (which prevents interactions between different sessions). These properties are easy to obtain via appropriate definitions of the algorithms in our template:

For (i), one uses the **CheckID** algorithm to specify machine instances that do not accept entities from two or more different "sessions", where a "session" can be defined in many different ways. For example, if one considers a "session" to be defined by a single shared SID, as is common in the UC and GNUC models, one can define **CheckID** such that all entities using the same SID are accepted by one instance. This essentially creates what is called an ideal functionality in the UC and GNUC models, which uses a single instance to manage all parties in a single session and fulfills (i). Alternatively, one can also, e.g., use the default definition of **CheckID** that accepts a single unique entity per instance and which creates what is called a real protocol in the UC and GNUC models. For (ii), one uses appropriate definitions of the **Initialization**, **MessagePreprocessing**, **Main**, and **AllowAdvMessage** algorithms that send messages to other entities only if they are in the same session.

For protocols defined in such a way, our framework offers the following single session composition theorem:

Corollary 6 (Unbounded self composition (informal))**.** Let $\mathcal{P}$ and $\mathcal{F}$ be two complete protocols such that they are R-environmentally bounded and $\mathcal{P}$ realizes $\mathcal{F}$ in a single session. That is, the environment may send messages only to a single session of $\mathcal{P}$ or $\mathcal{F}$, respectively.
Then $\mathcal{P} \leq \mathcal{F}$, i.e., $\mathcal{P}$ realizes $\mathcal{F}$ in an unbounded number of sessions.

Proof. This is a direct implication of the unbounded self composition theorem of the IITM model. See also the formalized version of this theorem in Appendix C, which includes a more detailed argument. □

We provide a more detailed description of single session composition, including a formal version of the above theorem and an example of a protocol with disjoint sessions, in Appendix C. Note that we, just as for concurrent composition, we do not require a specific internal structure of $\mathcal{P}$ and $\mathcal{F}$ besides session disjointness. In addition, the definition of a "session" is not fixed and can be determined depending on the type of protocol. For example, sometimes it can be useful to define a "session" to be comprised of all entites that share the same prefix of their SIDs, or share the same (prefix of the) PID. This is in contrast to the UC and GNUC models, which fix in their theorems how a "session" is defined.

B. Joint-State Composition for Multiple Sessions

Modeling protocols with disjoint sessions (cf. §V-A) is not always realistic. In many cases, some kind of state should be shared between instances of the same machine in different sessions. For example, cryptographic keys for signing and verifying messages are generally supposed to be re-used across multiple sessions of a protocol. In order to be able to also capture these settings in universal composability models that assume disjoint sessions, the concept of joint-state composition was introduced [8], [11], [18]. A joint-state composition theorem intuitively states that a protocol $\mathcal{F}$ with disjoint sessions can be replaced by another protocol $\mathcal{P}$ that shares state between multiple sessions if no environment can distinguish both cases.

Supporting joint-state composition in models that assume disjoint sessions to be the default, such as UC and GNUC, entails introducing a new set of syntax constructs, defining a new realization relation, and defining (and proving) an entirely new composition theorem. In contrast, our framework supports joint-state seamlessly and out of the box, without needing any modifications or new theorems: as already mentioned in §V-A, protocols are able to share state by default, so no new syntax is necessary. To give an example, consider some protocol $\mathcal{F}$ with disjoint sessions where an instance of $\mathcal{F}$ manages all entities sharing a single SID. Such a protocol can re-use, e.g., some cryptographic key across multiple entities with the same SID, however, since different SIDs are handled by different instances, every session will use a different key. Now, one can define a joint-state realization $\mathcal{P}_{js}$ of $\mathcal{F}$ (with the same public roles) where one instance of $\mathcal{P}_{js}$ accepts *all* entities, also in different sessions. In $\mathcal{P}_{js}$, one can then use *the same* key for entities in multiple sessions as all entities are handled by the same instance. Once we have shown that $\mathcal{P}_{js}$ realizes $\mathcal{F}$ (as per Definition 1), we can use the following concurrent composition theorem for joint-state:

Corollary 7 (Concurrent composition with joint-state)**.** Let $\mathcal{P}_{js}$ be a R-environmentally bounded protocol with joint-state and $\mathcal{F}$ be a R-environmentally bounded protocol with disjoint sessions such that $\mathcal{P} \leq \mathcal{F}$. Let $\mathcal{Q}$ be another protocol such that $\mathcal{Q}$ and $\mathcal{F}$ are connectable. Let $\mathcal{R} \in \mathsf{Comb}(\mathcal{Q}, \mathcal{P}_{\mathsf{js}})$ and let $\mathcal{I} \in \mathsf{Comb}(\mathcal{Q}, \mathcal{F})$ such that $\mathcal{R}$ and $\mathcal{I}$ have the same sets of public roles.
If $\mathcal{R}$ is R-environmentally bounded, then

$$\mathcal{R} \leq \mathcal{I} \quad .$$

Proof. This is a mere special case of Corollary 2 and as such trivially implied. □

Note that, because both disjoint sessions and joint-state are mere special cases in our framework, both the realization relation and the concurrent composition theorem remain unchanged. In particular, we do not have to change any of the definitions, syntax, theorems, or introduce additional requirements. Overall, this drastically simplifies handling of joint-state for protocol designers as they are able to work with the same syntax, definitions, and theorems.

We want to highlight that the corruption model of our framework is fully compatible with joint-state in the sense that one can actually prove realizations. To understand why this is a non-trivial feature, consider an ideal signature

functionality $\mathcal{F}_{\text{sig}}$ with disjoint sessions and a potential joint-state realization $\mathcal{P}^{js}_{\text{sig}}$ that re-uses the same signature key in all sessions. Now, if an adversary corrupts the single signing key in $\mathcal{P}^{js}_{\text{sig}}$ and can thus forge messages for all sessions, this corresponds to infinitely many corrupted sessions in $\mathcal{F}_{\text{sig}}$. A simulator must be able to perform all of those corruptions, even though he has only polynomial runtime. Our framework deals with this issue by asking for the corruption state of newly created entities before any other operations are performed, i.e., the simulator does not have to pro-actively corrupt non-existing entities but can instead act re-actively when the environment triggers a new entity for the first time. We note that the corruption model proposed in the current version of the UC model actually prevents many joint-state realizations, such as $\mathcal{P}^{js}_{\text{sig}}$ for $\mathcal{F}_{\text{sig}}$ (we show this claim in Appendix D).

C. Joint-State Composition for Multiple Protocols

Most universal composability models (including the UC and GNUC models) consider only the above type of joint-state where a single instance of *one* machine realizes multiple instances of *one* machine.

Our framework, however, also allows for various other types of joint-state composition. It for instance allows one to use one instance of *one* machine to realize instances of *multiple* different machines. For example, one can use a protocol $\mathcal{P}$ to realize the combination of two protocols $\mathcal{F} \,||\, \mathcal{F}'$, where one machine instance of $\mathcal{P}$ realizes both an instance of $\mathcal{F}$ *and* instance of $\mathcal{F}'$. The concurrent composition theorem then implies that $\mathcal{Q} \,||\, \mathcal{P}$ realizes $\mathcal{Q} \,||\, \mathcal{F} \,||\, \mathcal{F}'$ (for an arbitrary protocol $\mathcal{Q}$), i.e., we can replace multiple independent protocols by a single one that is able to re-use some state across different invocations.

To illustrate when and why this type of joint state is useful, consider the following example. Let $\mathcal{R}$ be some higher level protocol, such as a key exchange, using an ideal signature protocol $\mathcal{F}_{\text{sig}} = (\texttt{signer}, \texttt{verifier})$ as a subroutine. Analogously, let $\mathcal{R}'$ be a different higher level protocol also using an ideal signature protocol $\mathcal{F}'_{\text{sig}}$ as a subroutine (where $\mathcal{F}_{\text{sig}}$ and $\mathcal{F}'_{\text{sig}}$ use the same machine code). The combined protocols $\mathcal{R} \,||\, \mathcal{F}_{\text{sig}}$ and $\mathcal{R}' \,||\, \mathcal{F}'_{\text{sig}}$ can be analyzed in isolation and proven to be secure. Now, the composition theorem implies that these protocols running concurrently, i.e., the combined protocol $\mathcal{R} \,||\, \mathcal{R}' \,||\, \mathcal{F}_{\text{sig}} \,||\, \mathcal{F}'_{\text{sig}}$, are still secure. This combination contains two separate subroutines $\mathcal{F}_{\text{sig}}$ and $\mathcal{F}'_{\text{sig}}$ that do not share any state between each other, i.e., $\mathcal{R}$ and $\mathcal{R}'$ use different signature keys. Ideally, one would like to obtain security even if $\mathcal{R}$ and $\mathcal{R}'$ use the same signature key; intuitively, this should be possible if $\mathcal{R}$ and $\mathcal{R}'$ sign messages from disjoint messages spaces such that signatures by $\mathcal{R}$ do not impact $\mathcal{R}'$ and vice-versa.

Our framework allows for showing this expected security result via an appropriate joint-state realization and the composition theorem: one defines a joint-state realization $\mathcal{P}^{js}_{\text{sig}}$ that has the same public roles as $\mathcal{F}_{\text{sig}} \,||\, \mathcal{F}'_{\text{sig}}$, i.e., two `signer` and two `verifier` roles. Internally, $\mathcal{P}^{js}_{\text{sig}}$ accepts all entities in one machine instance which then acts as a multiplexer for a single subroutine $\mathcal{F}''_{\text{sig}}$ (that, again, uses the same code as $\mathcal{F}_{\text{sig}}$). In particular, signing requests arriving for any of the signer roles of $\mathcal{P}^{js}_{\text{sig}}$ are prefixed with a unique ID, depending on the role where they arrived, and then forwarded to the `signer` role of $\mathcal{F}''_{\text{sig}}$. In other words, $\mathcal{P}^{js}_{\text{sig}}$ uses the same subroutine and thus the same signing key for signing requests arriving for both public `signer` roles. Once we have shown that $\mathcal{P}^{js}_{\text{sig}} \,||\, \mathcal{F}''_{\text{sig}} \leq \mathcal{F}_{\text{sig}} \,||\, \mathcal{F}'_{\text{sig}}$, we can use the concurrent composition theorem (cf. Corollary 2) to conclude that $\mathcal{R} \,||\, \mathcal{R}' \,||\, \mathcal{P}^{js}_{\text{sig}} \,||\, \mathcal{F}''_{\text{sig}} \leq \mathcal{R} \,||\, \mathcal{R}' \,||\, \mathcal{F}_{\text{sig}} \,||\, \mathcal{F}'_{\text{sig}}$. Thus we can use the same signing key for both protocols and still retain security by adding unique prefixes to keep the message spaces of each protocol disjoint (as is common in many real-world protocols).

D. Global Functionalities and Global State

Sometimes it is desirable to define a protocol in such a way that it exposes some of its subroutines to other protocols, the idea being that some state can or should be shared with other arbitrary (and unknown) protocols. For example, a common reference string (CRS) is generally considered to be a globally available resource, so it seems natural to make it globally available instead of modeling it as an internal subroutine that no other protocols can see or access. Another example is a subroutine modeling a public key infrastructure based on certificate authorities which should make public keys accessible for every protocol, not just one specific protocol (we use this example in our case study in §IV).

Just as for joint-state, most universal composability models, including the UC and GNUC models, had to introduce additional extensions to model so-called global state [9], [11], [22]. This entails additional syntax, changes to the definition of the realization relation, and introducing (potentially multiple) new composition theorems. In contrast, again, our framework seamlessly supports global state out of the box without any modifications to syntax or definitions. This is due to the built-in concept of public and private roles: having a globally available subroutine is as simple as making a role public. For example, consider a protocol $\mathcal{F}_{\mathsf{CRS}} = (\texttt{retrieveCRS})$ with a single role modeling a CRS, and a higher level protocol $\mathcal{P} = (\texttt{somePublicRole} \mid \texttt{somePrivateRole})$ using $\mathcal{F}_{\mathsf{CRS}}$ as a subroutine. If one wants to model a CRS that is only locally available to $\mathcal{P}$, then one considers the combined protocol $(\texttt{somePublicRole} \mid \texttt{somePrivateRole}, \texttt{retrieveCRS})$ (which can also be written as $(\mathcal{P} \mid \mathcal{F}_{\mathsf{CRS}})$ using the shorter notation from §III-D); to model a global CRS, one considers the combined protocol $(\texttt{somePublicRole}, \texttt{retrieveCRS} \mid \texttt{somePrivateRole})$ instead (which can also be written as $(\mathcal{P}, \mathcal{F}_{\mathsf{CRS}})$). Once we have shown that some protocol $\mathcal{P}$ with global state realizes some other protocol $\mathcal{F}$, we can use the following concurrent composition theorem:

Corollary 8 (Concurrent composition with global-state)**.** Let $\mathcal{P}$ and $\mathcal{F}$ be two R-environmentally bounded protocols with global state such that $\mathcal{P} \leq \mathcal{F}$. Let $\mathcal{Q}$ be another

protocol such that $\mathcal{Q}$ and $\mathcal{F}$ are connectable. Let $\mathcal{R} \in \mathsf{Comb}(\mathcal{Q}, \mathcal{P})$ and let $\mathcal{I} \in \mathsf{Comb}(\mathcal{Q}, \mathcal{F})$ such that $\mathcal{R}$ and $\mathcal{I}$ have the same sets of public roles.
If $\mathcal{R}$ is R-environmentally bounded, then

$$\mathcal{R} \leq \mathcal{I} \quad .$$

Proof. This is a mere special case of Corollary 2 and as such trivially implied. □

Again, just as for joint-state, we emphasize that both the realization relation and the composition theorem remain unchanged. We do not have to change any of the definitions, syntax, theorems, or introduce additional requirements, which makes global state in our framework much more user friendly.

We want to highlight that global state in our framework is very flexible due to our concept of public and private roles. For example, it is possible to make only parts of a protocol publicly available, instead of the full protocol. Our case study in §IV makes use of this feature to define a globally available public key infrastructure where key registration is protected. Furthermore, since public roles can be changed to be private when combined with another protocol, one can actually change global subroutines to be only locally available to a single protocol while retaining all security results and while still being able to use the composition theorem to replace that subroutine with its realization.

E. Global and Local Session IDs

As already explained in §IV-C, most universal composability models, including the UC and GNUC models, assume that all instances in a protocol session have somehow agreed on a globally unique session identifier before the start of the protocol. While this assumption is fine for some protocols, it prevents a faithful analysis of protocols that do not establish a session ID prior to start of a run but rather do so during the actual protocol execution (perhaps even only implicitly). In fact, there are protocols that can be shown to be secure when using a pre-established global SID, but become insecure without it (see [20] for an in-depth discussion of local SIDs).

To facilitate the analysis of different kinds of protocols, our framework supports both global (pre-established) SIDs and local SIDs that are managed by each participant on their own and may very well differ between several participants in the same session. We provide examples for both types of SIDs in our case study in §IV. On a high level, one of the main differences is how the **CheckID** algorithm is specified: for global SIDs which are shared by all participants in the same session, an instance generally accepts entities with the same SID only. For local SIDs that need not be the same for different participants in the same session, an instance might accept entities with varying SIDs. Then, such an instance might internally group entities into new "sessions", which models that, e.g., several entities with different local SIDs are executing a key exchange together and end up with a shared secret that determines who is part of the same "session". We emphasize that most other models, including the UC and GNUC models, do not directly support local SIDs.

F. Separating Entities and Instances

Historically, universal composability models generally did not distinguish between instances of a machine and a specific person executing a protocol in a specific session. Instead, they defined different types of protocols depending on what an instance is supposed to model: An instance of a real protocol stands for one party in one session, an instance of an ideal protocol stands for all parties in one session, an instance of a joint-state protocol stands for one party in all sessions, and so on. For each different modeling choice, a new type of protocol including corresponding notation had to be introduced.

Our framework introduces the concept of entities to clearly separate the modeling of a party in a session from the actual machine instance in the protocol run. This feature enables us to use *a single template* with a single set of notation to express multiple different types of protocols, including all of the classical protocol types mentioned above. A protocol designer can easily create his own mapping between instances and entities using the **CheckID** algorithm (as also illustrated in §V-A). For example, to model an ideal functionality as defined in the UC and GNUC models, one defines an instance that accepts all entities with a specific SID; to model a real protocol, instances accept only a single entity; to model a joint-state protocol, instances accept all entites belonging to the same party. One can also, e.g., design a single (potentially global) instance that manages all possible entities; see also $\mathcal{F}_{\text{CA}}$ in our case study in §IV for an example. This flexibility is not just limited to classical protocol types but one is rather able to express many other variations that have not been explored in the literature so far, such as instances that accept entities for fixed ranges of SIDs, which might be useful to model, e.g., using the same cryptographic key for a certain number of sessions before a new one is chosen.

G. Corruption Model

The corruption models proposed by the GNUC and UC models are quite limited in terms of expressiveness and/or lacking in terms of usability. For example, in GNUC protocols can only be corrupted top-down, i.e., the highest layers must be corrupted before subroutines can be corrupted, making it impossible to model situations where, e.g., a single subroutine gets corrupted but security for the overall protocol can still be guaranteed. The UC model defines a corruption model only for real protocols, whereas the specification of corruption of ideal functionalities is left to the protocol designer. This is quite tedious as there are several parts that are identical across most if not all ideal functionalities as they have to match the (observable) behavior of real protocols. Perhaps for this reason, many protocol designers do not explicitly specify

corruption behavior for their ideal protocols, thus leaving them underspecified.

In contrast, our framework provides a very flexible corruption model that, at the same time, is easy to use. One can easily adapt the corruption model to various different situations by tweaking one of the four corruption related algorithms To give just a few examples, one can use the **DetermineCorrStatus** algorithm to consider a higher level protocol to be corrupted if one/a certain percentage/all of its subroutines are corrupted, modeling situations where security guarantees can still be obtained until too many subroutines are corrupted. The **AllowCorruption** algorithm can be used to define incorruptible machines modeling, e.g., setup assumptions, or it can be used to force the adversary to corrupt all subroutines first, modeling that he can only take full control of a party/computer. The **AllowAdvMessage** algorithm can be used to restrict access of corrupted entities to other, potentially honest entities. This can be useful to restrict, e.g., access to an uncorrupted signature key stored on a secure hardware token that is not directly accessible by corrupted software running on a PC. In addition to these algorithms, our framework also provides mechanisms to model both static and dynamic corruption which can be freely combined with arbitrary definitions of the above algorithms (see also §III-B). Since all of these algorithms come with sensible defaults, we do not overburden a protocol designer. Instead, one is able to omit most or all of these algorithms and still obtain a fully specified protocol with reasonable behavior. Last but not least, our corruption model is defined for and compatible with various different types of protocols such as real, ideal, joint-state, and global state protocols as all of these protocols use the same underlying template.

H. Capturing SUC

To further illustrate the flexibility and epxressiveness of our framework, we show that we can easily capture the SUC model by Canetti et al. [10] as a mere special case in our framework. The SUC model was created as a simpler version of the UC model that is tailored towards the setting of secure multiparty computation. The most important changes are the following:

(i) A run consists of a fixed number of parties, each of them corresponding to one machine instance, instead of an unbounded number.
(ii) The runtime definition is simplified, i.e., machine instances are only required to run in polynomial time.
(iii) All communication is assumed to occur over authenticated channels.
(iv) The corruption model is also defined for ideal protocols.
(v) Machines do not have any subroutines except for possibly a single ideal functionality (i.e., all program logic is contained in a single machine).

Since some of those changes have to break out of the UC model, the authors of the SUC model had to create and prove a new composition theorem which takes up a major part of their paper.

It is easy to obtain all of the above properties within our framework by choosing appropriate definitions for all fields in our template:

(i) The number of parties can be fixed by defining the **CheckID** algorithm such that PIDs are accepted only if they are in the range of $[1, \ldots, n]$. If desired, one can also limit the number of sessions in the same way.
(ii) The runtime definition in our framework is already simpler than in SUC, as we (intuitively) require only the whole protocol to run in polynomial time instead of individual machines.
(iii) To model authenticated channels, one uses a subroutine $\mathcal{F}_{\text{ach}}$ for authenticated channels that forwards messages while leaking their content.
(iv) Corruption for ideal protocols is already defined in our framework and can be further customized, if desired.
(v) It is straightforward to encode the whole protocol logic into a single **Main** algorithm, if so desired, while connecting to at most a single (ideal) subroutine.

Importantly, we do not have to re-prove a composition theorem for creating what is just a mere instantiation. Furthermore, since we do not have to break out of our framework, such an instantiation can be combined and/or realized with arbitrary other protocols defined in our framework, including those that use joint-state and global state. This is in contrast to SUC, which supports only disjoint sessions with local state and cannot directly be combined with other UC protocols as they use different computational models (only a mapping from SUC protocols to artifical UC protocols exists).

VI. Conclusion

We have introduced the iUC framework for universal composability.

As illustrated by our case study, iUC is highly *flexible*, compared, for example, to the UC and GNUC models. It supports, among others, all of the following at once:

- Modeling different types of addressing mechanisms for protocol instances via global and local SIDs.
- Arbitrary protocol structures that are not necessarily hierarchical trees of machines/roles. Also, our structures may have several highest-level protocol machines/roles.
- Ideal and global functionalities with arbitrary structure, rather than monolithic single machines. This is crucial for a modular and verifiable design and analysis of complex cryptographic protocols.
- Machine instances which can model many different settings (one party, one session, one party in one session, ...) by accepting different sets of entities.
- Various forms of shared state, including, for example, global state and shared state across sessions by one or more parties in a form which goes beyond classical joint-state constructions.

- A more general version of global state where arbitrary parts of protocols can be made globally available via the concept of public roles, while other parts remain local subroutines.

iUC also supports many other features that were discussed in §V, including the classical multi-session composition theorems as well as the classical joint state theorem (which follows directly from Corollary 2), and a broad range of different corruption models. Our framework is even able to directly capture specialized frameworks, such as the SUC model for secure multi party computation [10].

The iUC framework combines this flexibility with greatly improved *usability* compared to the IITM model, but also compared to other models, such as UC and GNUC, due to the following:

- A general concept of roles and subroutines for modeling the internal structure of (systems of) machines and their connections, abstracting away from tapes and interfaces in the IITM.
- Many recurrent patterns of protocols are already defined by the framework, including the corruption model and the interpretation of machine instances.
- There is only a single template with a single set of conventions that can be used to specify and analyze many types protocols, including real, ideal, joint-state, and global protocols.
- The template includes many optional parts with sensible defaults such that protocol designers can concentrate on the core logic of their protocol.

Finally, iUC allows protocol designers to easily create *sound* and *complete* protocol specifications and security proofs:

- All theorems, including both composition theorems, directly carry over from the IITM model to iUC.
- There are no unnecessary modeling artifacts, such as artificially padding messages, machine exhaustion, dynamic instantiations with new machine code, or flow bounds, that a protocol designer has to manually take care of, facilitating formally correct proofs.
- The template itself fully specifies the exact behavior of a protocol, even if optional parts are omitted.

Altogether, the iUC framework is a well-founded framework for universal composability which combines soundness, flexibility, and usability in an unmatched way. As such it is an important and convenient tool for the precise modular design and analysis of security protocols and applications.

References

[1] R. Canetti, "Universally Composable Security: A New Paradigm for Cryptographic Protocols," in *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science (FOCS 2001)*. IEEE Computer Society, 2001, pp. 136–145.

[2] B. Pfitzmann and M. Waidner, "A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2001, pp. 184–201.

[3] K. Hogan, H. Maleki, R. Rahaeimehr, R. Canetti, M. van Dijk, J. Hennessey, M. Varia, and H. Zhang, "On the Universally Composable Security of OpenStack," *IACR Cryptology ePrint Archive*, vol. 2018, p. 602, 2018.

[4] R. Canetti, K. Hogan, A. Malhotra, and M. Varia, "A Universally Composable Treatment of Network Time," in *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. IEEE Computer Society, 2017, pp. 360–375.

[5] S. Chari, C. S. Jutla, and A. Roy, "Universally Composable Security Analysis of OAuth v2.0," *IACR Cryptology ePrint Archive*, vol. 2011, p. 526, 2011.

[6] R. Canetti, S. Chari, S. Halevi, B. Pfitzmann, A. Roy, M. Steiner, and W. Z. Venema, "Composable Security Analysis of OS Services," in *Applied Cryptography and Network Security - 9th International Conference, ACNS 2011, Nerja, Spain, June 7-10, 2011. Proceedings*, ser. Lecture Notes in Computer Science, vol. 6715, 2011, pp. 431–448.

[7] P. Chaidos, O. Fourtounelli, A. Kiayias, and T. Zacharias, "A Universally Composable Framework for the Privacy of Email Ecosystems," in *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III*, ser. Lecture Notes in Computer Science, vol. 11274. Springer, 2018, pp. 191–221.

[8] R. Canetti and T. Rabin, "Universal Composition with Joint State," in *Advances in Cryptology, 23rd Annual International Cryptology Conference (CRYPTO 2003), Proceedings*, ser. Lecture Notes in Computer Science, vol. 2729. Springer, 2003, pp. 265–281.

[9] R. Canetti, Y. Dodis, R. Pass, and S. Walfish, "Universally Composable Security with Global Setup," in *Theory of Cryptography, Proceedings of TCC 2007*, ser. Lecture Notes in Computer Science, S. P. Vadhan, Ed., vol. 4392. Springer, 2007, pp. 61–85.

[10] R. Canetti, A. Cohen, and Y. Lindell, "A Simpler Variant of Universally Composable Security for Standard Multiparty Computation," in *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 9216. Springer, 2015, pp. 3–22.

[11] D. Hofheinz and V. Shoup, "GNUC: A New Universal Composability Framework," *J. Cryptology*, vol. 28, no. 3, pp. 423–508, 2015.

[12] U. Maurer and R. Renner, "Abstract Cryptography," in *Innovations in Computer Science - ICS 2010. Proceedings*, B. Chazelle, Ed. Tsinghua University Press, 2011, pp. 1–21.

[13] U. Maurer, "Constructive Cryptography - A New Paradigm for Security Definitions and Proofs," in *TOSCA 2011*, ser. LNCS, vol. 6993, 2011, pp. 33–56.

[14] R. Küsters, "Simulation-Based Security with Inexhaustible Interactive Turing Machines," in *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW-19 2006)*. IEEE Computer Society, 2006, pp. 309–320, see http://eprint.iacr.org/2013/025/ for a full and revised version.

[15] R. Canetti, L. Cheung, D. K. Kaynar, M. Liskov, N. A. Lynch, O. Pereira, and R. Segala, "Analyzing Security Protocols Using Time-Bounded Task-PIOAs," *Discrete Event Dynamic Systems*, vol. 18, no. 1, pp. 111–159, 2008.

[16] R. Canetti, "Universally Composable Security: A New Paradigm for Cryptographic Protocols," Cryptology ePrint Archive, Tech. Rep. 2000/067, 2000, available at http://eprint.iacr.org/2000/067 with new versions from December 2005 and July 2013.

[17] R. Küsters and M. Tuengerthal, "The IITM Model: a Simple and Expressive Model for Universal Composability," Cryptology ePrint Archive, Tech. Rep. 2013/025, 2013, available at http://eprint.iacr.org/2013/025.

[18] ——, "Joint State Theorems for Public-Key Encryption and Digital Signature Functionalities with Local Computation," in *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008)*. IEEE Computer Society, 2008, pp. 270–284, the full version is available at https://eprint.iacr.org/2008/006.

[19] ——, "Universally Composable Symmetric Encryption," in *CSF '09*, 2009, pp. 293–307.

[20] ——, "Composition Theorems Without Pre-Established Session Identifiers," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)*, Y. Chen, G. Danezis, and V. Shmatikov, Eds. ACM, 2011, pp. 41–50.

[21] J. Camenisch, R. R. Enderlein, S. Krenn, R. Küsters, and D. Rausch, "Universal Composition with Responsive Environments," in *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security*, ser. Lecture Notes in Computer Science, J. H. Cheon and T. Takagi, Eds., vol. 10032. Springer, 2016, pp. 807–840.

[22] R. Canetti, D. Shahaf, and M. Vald, "Universally Composable Authentication and Key-Exchange with Global PKI," in *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 9615. Springer, 2016, pp. 265–296.

[23] Anonymized for Review, "iUC: Flexible Universal Composability Made Simple (Full Version)," Technical Report, 2018, available at http://51.75.151.144/iuc-full.pdf .

[24] "ISO/IEC IS 9798-3, Entity authentication mechanisms — Part 3: Entity authentication using assymetric techniques," 1993.

[25] R. Canetti and H. Krawczyk, "Universally Composable Notions of Key Exchange and Secure Channels," in *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Proceedings*, ser. Lecture Notes in Computer Science, vol. 2332. Springer, 2002, pp. 337–351.

[26] R. Küsters and D. Rausch, "A Framework for Universally Composable Diffie-Hellman Key Exchange," in *IEEE 38th Symposium on Security and Privacy (S&P 2017)*. IEEE Computer Society, 2017, pp. 881–900.

Description of the protocol $\mathcal{P}_{\text{KE}} = (\texttt{initiator}, \texttt{responder} \mid \texttt{setup})$:

Participating roles: $\texttt{initiator}, \texttt{responder}, \texttt{setup}$
Corruption model: static
Protocol parameters:
- $\mathsf{groupGen}(1^\eta)$. {*Algorithm for generating tuples (G, n, g) describing cyclic groups G of size n with generator g.*

Description of $M_{\texttt{initiator}}$:

Implemented role(s): $\texttt{initiator}$
Subroutines: $\texttt{setup}$, $\mathcal{F}_{\text{sig-CA}}$, $\mathcal{F}_{\text{CA}} : \texttt{retrieval}$
Internal state:
- $(\mathsf{G}, \mathsf{n}, \mathsf{g}) \in (\{0,1\}^* \cup \{\bot\})^3 = (\bot, \bot, \bot)$ {*Global group parameters.*
- $\mathsf{state} \in \{\bot, \texttt{started}, \texttt{inSession}, \texttt{finished}\}$ {*Current state in the key exchange.*
- $\mathsf{caller} \in \{0,1\}^* \cup \{\bot\} = \bot$ {*Stores the initial caller of this entity/instance.*
- $\mathsf{intendedPID} \in \{0,1\}^* \cup \{\bot\} = \bot$ {*Stores the intended partner PID.*
- $\mathsf{k} \in \{0,1\}^* \cup \{\bot\} = \bot$ {*Stores the session key.*
- $\mathsf{e}_{\mathsf{init}} \in \mathbb{Z}_n \cup \{\bot\} = \bot$ {*Secret DH exponent of initiator.*
- $\mathsf{h}_{\mathsf{resp}} \in \mathsf{G} \cup \{\bot\} = \bot$ {*Public DH key share of responder.*

Corruption behavior:
- **DetermineCorrStatus**$(pid, sid, role)$: {*Consider entity corrupted if one of the signature keys is corrupted.*
  if $\mathsf{intendedPID} = \bot$: return $\mathbf{corr}(pid, (pid, \epsilon), \mathcal{F}_{\text{sig}} : \texttt{signer})$. {**corr** *sends* CorruptionStatus*? to the specified entity, waits for the response, and then outputs that response.*
  else: return $\mathbf{corr}(\mathsf{intendedPID}, (\mathsf{intendedPID}, \epsilon), \mathcal{F}_{\text{sig}} : \texttt{signer}) \vee \mathbf{corr}(pid, (pid, \epsilon), \mathcal{F}_{\text{sig}} : \texttt{signer})$.
- **AllowAdvMessage**$(pid, sid, role, pid_{\text{receiver}}, sid_{\text{receiver}}, role_{receiver}, m)$: {*The adversary may not use an uncorrupted signer entity. If the signer entity is corrupted, then the adversary already knows the key, so there is no need to give access.*
  If $role_{receiver} = \mathcal{F}_{\text{sig-CA}} : \texttt{signer}$, return $\texttt{false}$.
  Otherwise output $\texttt{true}$ iff $pid = pid_{\text{receiver}}$.

Initialization:

send $\texttt{GetParameters}$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \epsilon, \texttt{setup})$; {*Get DH parameters*
wait for $(\texttt{GetParameters}, (G, n, g))$.
$(\mathsf{G}, \mathsf{n}, \mathsf{g}) \leftarrow (G, n, g)$.
send $\texttt{InitSign}$ **to** $(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{pid}_{\mathsf{cur}}, \epsilon), \mathcal{F}_{\text{sig-CA}} : \texttt{signer})$;
wait for $(\texttt{InitSign}, \texttt{success}, \_)$.

Main:

recv $(\texttt{InitKE}, intendedPartner)$ **from** $\texttt{I/O}$ **s.t.** $\mathsf{state} = \bot$: {*Start KE and send first message*
$(\mathsf{state}, \mathsf{caller}, \mathsf{intendedPID}) \leftarrow (\texttt{started}, \mathsf{entity}_{\mathsf{call}}, intendedPartner)$.
Choose $\mathsf{e}_{\mathsf{init}} \leftarrow \mathbb{Z}_n$ uniformly at random, compute $h_{init} = \mathsf{g}^{\mathsf{e}_{\mathsf{init}}}$.
send $(\mathsf{pid}_{\mathsf{cur}}, h_{init})$ **to** $\texttt{NET}$.

recv $(\mathsf{intendedPID}, h_{resp}, \sigma)$ **from** $\texttt{NET}$ **s.t.** $\mathsf{state} = \texttt{started}$: {*Receive second message and output key*
send $(\texttt{Retrieve}, (\mathsf{intendedPID}, (\mathsf{intendedPID}, \epsilon)))$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \epsilon, \mathcal{F}_{\text{CA}} : \texttt{retrieval})$;
wait for $(\texttt{Retrieve}, pk)$. {*Get public verification key of intended partner*
if $pk = \bot$:
  abort.
$msg = (\mathsf{g}^{\mathsf{e}_{\mathsf{init}}}, h_{resp}, \mathsf{pid}_{\mathsf{cur}})$. {*Check signature*
send $(\texttt{Verify}, msg, \sigma, pk)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{intendedPID}, \epsilon), \mathcal{F}_{\text{sig-CA}} : \texttt{verifier})$;
wait for $(\texttt{VerResult}, b)$.
if $b = \texttt{false}$:
  abort.
$\mathsf{h}_{\mathsf{resp}} \leftarrow h_{resp}$; $\mathsf{k} \leftarrow \mathsf{h}_{\mathsf{resp}}{}^{\mathsf{e}_{\mathsf{init}}}$; $\mathsf{state} \leftarrow \texttt{finished}$.
send $(\texttt{FinishKE}, \mathsf{k})$ **to** caller.

recv $\texttt{GetLastMessage}$ **from** $\texttt{NET}$ **s.t.** $\mathsf{state} = \texttt{finished}$: {*Allow the adversary to retrieve the third protocol message. This works around the fact that machines may send only one message at a time, but the last protocol step outputs two (one for the network and one for the user with the key).*
$m = (\mathsf{h}_{\mathsf{resp}}, \mathsf{g}^{\mathsf{e}_{\mathsf{init}}}, \mathsf{intendedPID})$.
send $(\texttt{Sign}, m)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{pid}_{\mathsf{cur}}, \epsilon), \mathcal{F}_{\text{sig-CA}} : \texttt{signer})$;
wait for $(\texttt{Signature}, \sigma)$.
send σ **to** $\texttt{NET}$.

Description of $M_{\texttt{setup}}$:

Implemented role(s): $\texttt{setup}$
Internal state:
- $(\mathsf{G}, \mathsf{n}, \mathsf{g}) \in (\{0,1\}^* \cup \{\bot\})^3$ {*Global group parameters, initially $\bot$.*

CheckID$(pid, sid, role)$**:** Accept all entities.
Corruption behavior:
- **AllowCorruption**$(pid, sid, role)$: return $\texttt{false}$. {*The adversary may not corrupt the (honestly generated) setup parameters.*

Initialization:

$(\mathsf{G}, \mathsf{n}, \mathsf{g}) \leftarrow \mathsf{groupGen}(1^\eta)$.

Main:

recv $\texttt{GetParameters}$ **from** $\_$: {*Everyone may retrieve the group parameters, including the adversary on the network.*
reply $(\texttt{GetParameters}, (\mathsf{G}, \mathsf{n}, \mathsf{g}))$.

Fig. 7: A real key exchange protocol $\mathcal{P}_{\text{KE}}$ for realizing $\mathcal{F}_{\text{KE}}$ (part 1).

Description of $M_{\texttt{responder}}$:

Implemented role(s): `responder`
Subroutines: `setup`, $\mathcal{F}_{\text{sig-CA}}$, $\mathcal{F}_{\text{CA}}$: `retrieval`
Internal state:

- $(\mathsf{G}, \mathsf{n}, \mathsf{g}) \in (\{0,1\}^* \cup \{\bot\})^3 = (\bot, \bot, \bot)$ {*Global group parameters.*
- $\mathsf{state} \in \{\bot, \texttt{started}, \texttt{inSession}, \texttt{finished}\}$ {*Current state in the key exchange.*
- $\mathsf{caller} \in \{0,1\}^* \cup \{\bot\} = \bot$ {*Stores the initial caller of this entity/instance.*
- $\mathsf{intendedPID} \in \{0,1\}^* \cup \{\bot\} = \bot$ {*Stores the intended partner PID.*
- $\mathsf{k} \in \{0,1\}^* \cup \{\bot\} = \bot$ {*Stores the session key.*
- $\mathsf{e}_{\mathsf{resp}} \in \mathbb{Z}_n \cup \{\bot\} = \bot$ {*Secret DH exponent of responder.*
- $\mathsf{h}_{\mathsf{init}} \in \mathsf{G} \cup \{\bot\} = \bot$ {*Public DH key share of initiator.*

Corruption behavior:

- **DetermineCorrStatus**$(pid, sid, role)$: {*Consider entity corrupted if one of the signature keys is corrupted.*
  if $\mathsf{intendedPID} = \bot$: return **corr**$(pid, (pid, \epsilon), \mathcal{F}_{\text{sig}} : \texttt{signer})$.
  else: return **corr**$(\mathsf{intendedPID}, (\mathsf{intendedPID}, \epsilon), \mathcal{F}_{\text{sig}} : \texttt{signer}) \vee$ **corr**$(pid, (pid, \epsilon), \mathcal{F}_{\text{sig}} : \texttt{signer})$.
- **AllowAdvMessage**$(pid, sid, role, pid_{\text{receiver}}, sid_{\text{receiver}}, role_{receiver}, m)$: {*The adversary may not use an uncorrupted signer entity. If the signer entity is corrupted, then the adversary already knows the key.*
  If $role_{receiver} = \mathcal{F}_{\text{sig-CA}}$: `signer`, return `false`.
  Otherwise output `true` iff $pid = pid_{\text{receiver}}$.

Initialization:

send `GetParameters` **to** $(\mathsf{pid}_{\mathsf{cur}}, \epsilon, \texttt{setup})$; {*Get DH parameters.*
wait for $(\texttt{GetParameters}, (G, n, g))$.
$(\mathsf{G}, \mathsf{n}, \mathsf{g}) \leftarrow (G, n, g)$.
send `InitSign` **to** $(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{pid}_{\mathsf{cur}}, \epsilon), \mathcal{F}_{\text{sig-CA}} : \texttt{signer})$;
wait for $(\texttt{InitSign}, \texttt{success}, \_)$.

Main:

recv $(\texttt{InitKE}, intendedPartner)$ **from** `I/O` **s.t.** $\mathsf{state} = \bot$: {*Start KE.*
$(\mathsf{state}, \mathsf{caller}, \mathsf{intendedPID}) \leftarrow (\texttt{started}, \mathsf{entity}_{\mathsf{call}}, intendedPartner)$. {*Notify network that the key exchange has started and the responder is ready to receive the first message.*
send `InitKE` **to** `NET`.

recv $(\mathsf{intendedPID}, h_{init})$ **from** `NET` **s.t.** $\mathsf{state} = \texttt{started}$: {*Receive first message, send second message.*
$\mathsf{h}_{\mathsf{resp}} \leftarrow h_{init}$
Choose $\mathsf{e}_{\mathsf{resp}} \leftarrow \mathbb{Z}_n$ uniformly at random, compute $h_{resp} = \mathsf{g}^{\mathsf{e}_{\mathsf{resp}}}$.
$m = (\mathsf{h}_{\mathsf{init}}, \mathsf{g}^{\mathsf{e}_{\mathsf{resp}}}, \mathsf{intendedPID})$.
send $(\texttt{Sign}, m)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{pid}_{\mathsf{cur}}, \epsilon), \mathcal{F}_{\text{sig-CA}} : \texttt{signer})$;
wait for $(\texttt{Signature}, \sigma)$.
$\mathsf{state} \leftarrow \texttt{inSession}$
send $(\mathsf{pid}_{\mathsf{cur}}, h_{resp}, \sigma)$ **to** `NET`.

recv σ **from** `NET` **s.t.** $\mathsf{state} = \texttt{inSession}$: {*Receive third message, output key.*
send $(\texttt{Retrieve}, (\mathsf{intendedPID}, (\mathsf{intendedPID}, \epsilon)))$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \epsilon, \mathcal{F}_{\text{CA}} : \texttt{retrieval})$;
wait for $(\texttt{Retrieve}, pk)$. {*Get public verification key of intended partner.*
if $pk = \bot$:
  abort.
$msg = (\mathsf{g}^{\mathsf{e}_{\mathsf{resp}}}, \mathsf{h}_{\mathsf{init}}, \mathsf{pid}_{\mathsf{cur}})$. {*Check signature.*
send $(\texttt{Verify}, msg, \sigma, pk)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{intendedPID}, \epsilon), \mathcal{F}_{\text{sig-CA}} : \texttt{verifier})$;
wait for $(\texttt{VerResult}, b)$.
if $b = \texttt{false}$:
  abort.
$\mathsf{k} \leftarrow \mathsf{h}_{\mathsf{init}}{}^{\mathsf{e}_{\mathsf{resp}}}$; $\mathsf{state} \leftarrow \texttt{finished}$.
send $(\texttt{FinishKE}, \mathsf{k})$ **to** caller.

Fig. 8: A real key exchange protocol $\mathcal{P}_{\text{KE}}$ for realizing $\mathcal{F}_{\text{KE}}$ (part 2). Note that each instance of $M_{\texttt{initiator}}$ and $M_{\texttt{responder}}$ corresponds to a single entity.

Description of the protocol $\mathcal{F}_{\text{sig-CA}} = (\texttt{signer}, \texttt{verifier})$:

Participating roles: $\{\texttt{signer}, \texttt{verifier}\}$
Corruption model: dynamic with secure erasures
Protocol parameters:
- $\mathsf{p} \in \mathbb{Z}[x]$. {*Polynomial that bounds the runtime of the algorithms provided by the adversary.*

Description of $M_{\texttt{signer},\texttt{verifier}}$:

Implemented role(s): $\{\texttt{signer}, \texttt{verifier}\}$
Subroutines: $\mathcal{F}_{\text{CA}} : \texttt{registration}$
Internal state:
- $(\mathsf{sig}, \mathsf{ver}, \mathsf{pk}, \mathsf{sk}) \in (\{0,1\}^* \cup \{\bot\})^4 = (\bot, \bot, \bot, \bot)$. {*Algorithms and key pair.*
- $\mathsf{pidowner} \in \{0,1\}^* \cup \{\bot\} = \bot$. {*Party ID of the key owner.*
- $\mathsf{msglist} \subseteq \{0,1\}^* = \emptyset$. {*Set of recorded messages.*
- $\mathsf{KeysGenerated} \in \{\texttt{ready}, \bot\} = \bot$. {*Has signer initialized his key?*

CheckID$(pid, sid, role)$**:**
Check that $sid = (pid', sid')$; otherwise output `reject`.
Accept all entities with the same SID. {*An instance manages all parties and roles in a single session.*

Corruption behavior:
- **LeakedData**$(pid, sid, role)$: If called while $(pid, sid, role)$ determines its initial corruption status, use the default behavior of **LeakedData**. That is, output the initially received message and the sender of that message.
  Otherwise, if $role = \texttt{signer}$ and $pid = \mathsf{pidowner}$, return $\mathsf{KeysGenerated}$. In all other cases return $\bot$.
- **AllowAdvMessage**$(pid, sid, role, pid_{\text{receiver}}, sid_{\text{receiver}}, role_{receiver}, m)$:
  Check that $pid = pid_{\text{receiver}}$.
  If $role_{\text{receiver}} = \mathcal{F}_{\text{CA}} : \texttt{registration}$, also check that $role = \texttt{signer}$ and $sid = (pid, sid')$ (for some sid').
  {*Only the owner of a key may register it, modeling that $\mathcal{F}_{\text{CA}}$ authenticates the owner upon registration.*
  If all checks succeed, output `true`, otherwise output `false`.

Initialization:

send responsively `InitMe` **to** `NET`;
wait for $(\texttt{Init}, (sig, ver, pk, sk))$.
$(\mathsf{sig}, \mathsf{ver}, \mathsf{pk}, \mathsf{sk}) \leftarrow (sig, ver, pk, sk)$.
Parse $\mathsf{sid}_{\mathsf{cur}}$ as (pid, sid).
$\mathsf{pidowner} \leftarrow pid$.

Main:

recv `InitSign` **from** `I/O` **to** $(\mathsf{pidowner}, \_, \texttt{signer})$: {*Only the owner of the key can create (and use) his signing key.*
  send $(\texttt{Register}, \mathsf{pk})$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \epsilon, \mathcal{F}_{\text{CA}} : \texttt{registration})$;
  wait for $(\texttt{Register}, \_)$.
  $\mathsf{KeysGenerated} \leftarrow \texttt{ready}$. {*Successful initialization. Note that* `signer` *can submit* `InitSign` *multiple times, always with the same effect.*
  reply $(\texttt{InitSign}, \texttt{success}, \mathsf{pk})$.

recv $(\texttt{Sign}, msg)$ **from** `I/O` **to** $(\mathsf{pidowner}, \_, \texttt{signer})$ **s.t.** $\mathsf{KeysGenerated} = \texttt{ready}$:
  $\sigma \leftarrow \mathsf{sig}^{(\mathsf{p})}(msg, \mathsf{sk})$.
  $b \leftarrow \mathsf{ver}^{(\mathsf{p})}(msg, \sigma, \mathsf{pk})$. {*Sign and check that verification succeeds.*
  if $\sigma = \bot \vee b \neq \texttt{true}$:
    reply $(\texttt{Signature}, \bot)$. {*Signing or verification test failed.*
  else:
    add msg **to** $\mathsf{msglist}$.
    reply $(\texttt{Signature}, \sigma)$. {*Record msg for verification and return signature.*

recv $(\texttt{Verify}, msg, \sigma, pk)$ **from** `I/O` **to** $(\_, \_, \texttt{verifier})$:
  $b \leftarrow \mathsf{ver}^{(\mathsf{p})}(msg, \sigma, pk)$. {*Verify signature.*
  if $pk = \mathsf{pk} \wedge b = \texttt{true} \wedge msg \notin \mathsf{msglist} \wedge (\mathsf{pidowner}, \mathsf{sid}_{\mathsf{cur}}, \texttt{signer}) \notin \mathsf{CorruptionSet}$: {*cf. §III-C for* CorruptionSet.
    reply $(\texttt{VerResult}, \texttt{false})$. {*Prevent forgery.*
  else:
    reply $(\texttt{VerResult}, b)$. {*Return verification result.*

Fig. 9: The ideal signature functionality $\mathcal{F}_{\text{sig-CA}}$.

Description of the protocol $\mathcal{P}_{\text{sig-CA}} = (\texttt{signer}, \texttt{verifier})$:

Participating roles: `signer`, `verifier`
Corruption model: dynamic with secure erasures
Protocol parameters:
- an EUF-CMA signature scheme $\Sigma = (\mathsf{gen}, \mathsf{sig}, \mathsf{ver})$.

Description of $M_{\texttt{signer}}$:

Implemented role(s): `signer`
Subroutines: $\mathcal{F}_{\text{CA}}$: `registration`
Internal state:
- $(\mathsf{sk}, \mathsf{pk}) \in (\{0,1\}^* \cup \{\bot\})^2 = (\bot, \bot)$. {*key pair.*
- $\mathsf{pidowner} \in \{0,1\}^* \cup \{\bot\} = \bot$. {*Party ID of the key owner.*
- $\mathsf{KeysGenerated} \in \{\texttt{ready}, \bot\} = \bot$. {*Has signer initialized his key?*

CheckID$(pid, sid, role)$**:**
Check that $sid = (pid', sid')$; otherwise output `reject`.
Accept a single entity. {*An instance manages exactly one entity.*

Corruption behavior:
- **AllowAdvMessage**$(pid, sid, role, pid_{\text{receiver}}, sid_{\text{receiver}}, role_{receiver}, m)$:
  Check that $pid = pid_{\text{receiver}}$.
  If $role_{\text{receiver}} = \mathcal{F}_{\text{CA}}$: `registration`, also check that $sid = (pid, sid')$ (for some sid').
  If all checks succeed, output `true`, otherwise output `false`.

Initialization:
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{gen}(1^\eta)$.
Parse $\mathsf{sid}_{\mathsf{cur}}$ as (pid, sid).
$\mathsf{pidowner} \leftarrow pid$.

Main:

recv `InitSign` **from** `I/O` **to** $(\mathsf{pidowner}, \_, \_)$:
  send $(\texttt{Register}, \mathsf{pk})$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{CA}}$: `registration`$)$;
  wait for $(\texttt{Register}, \_)$.
  $\mathsf{KeysGenerated} \leftarrow \texttt{ready}$.
  reply $(\texttt{InitSign}, \texttt{success}, \mathsf{pk})$.

recv $(\texttt{Sign}, msg)$ **from** `I/O` **to** $(\mathsf{pidowner}, \_, \_)$ **s.t.** $\mathsf{KeysGenerated} = \texttt{ready}$:
  $\sigma \leftarrow \mathsf{sig}(msg, \mathsf{sk})$. {*Sign msg, return signature.*
  reply $(\texttt{Signature}, \sigma)$.

Description of $M_{\texttt{verifier}}$:

Implemented role(s): `verifier`
CheckID$(pid, sid, role)$**:**
Check that $sid = (pid', sid')$; otherwise output `reject`.
Accept a single entity. {*An instance manages exactly one entity.*

Main:

recv $(\texttt{Verify}, msg, \sigma, pk)$ **from** `I/O`:
  $b \leftarrow \mathsf{ver}(msg, \sigma, pk)$.
  reply $(\texttt{VerResult}, b)$. {*Verify, return result.*

Fig. 10: The real signature protocol $\mathcal{P}_{\text{sig-CA}}$.

Description of the protocol $\mathcal{F}_{\text{KE}} = (\texttt{initiator}, \texttt{responder})$:

Participating roles: $\{\texttt{initiator}, \texttt{responder}\}$
Corruption model: dynamic with secure erasures
Protocol parameters:
- $\mathsf{groupGen}(1^\eta)$. {*Algorithm for generating tuples (G, n, g) describing cyclic groups G of size n with generator g.*

Description of $M_{\texttt{initiator},\texttt{responder}}$:

Implemented role(s): $\{\texttt{initiator}, \texttt{responder}\}$
Subroutines: $\mathcal{F}_{\text{CA}} : \texttt{registration}$
Internal state:
- $(\mathsf{G}, \mathsf{n}, \mathsf{g}) \in (\{0,1\}^* \cup \{\bot\})^3 = (\bot, \bot, \bot)$ {*Global group parameters.*
- $\mathsf{state} : \{0,1\}^* \to \{\bot, \texttt{started}, \texttt{inSession}, \texttt{finished}\}$ {*Stores the current state of entities in key exchange; initially $\bot$.*
- $\mathsf{caller} : \{0,1\}^* \to \{0,1\}^* \cup \{\bot\}$ {*Stores the calling entity for each entity in key exchange; initially $\bot$.*
- $\mathsf{intendedPID} : \{0,1\}^* \to \{0,1\}^* \cup \{\bot\}$ {*Mapping from entity to intended partner PID; initially $\bot$.*
- $\mathsf{sessions} \subseteq \{0,1\}^* \times \{0,1\}^* = \emptyset$ {*Pairs of entities in the same global key exchange session.*
- $\mathsf{sessionKeys} : \{0,1\}^* \to \{0,1\}^* \cup \{\bot\}$ {*Mapping from entity to session key; initially $\bot$.*

CheckID$(pid, sid, role)$**:** Accept all entities.
Corruption behavior:
- **LeakedData**$(pid, sid, role)$: If called while $(pid, sid, role)$ determines its initial corruption status, use the default behavior of **LeakedData**. That is, output the initially received message and the sender of that message.
  Otherwise, return $(\mathsf{caller}[pid, sid, role], \mathsf{sessionKeys}[pid, sid, role])$.

Initialization:

recv m **from** *sender*:
  $(\mathsf{G}, \mathsf{n}, \mathsf{g}) \leftarrow \mathsf{groupGen}(1^\eta)$.
  if $sender = \texttt{NET} \wedge m = \texttt{InitGroup}$:
    send $(\texttt{LeakGroup}, (\mathsf{G}, \mathsf{n}, \mathsf{g}))$ **to** NET. {*Allow adversary to start initialization and then return the generated group. No other actions are performed in this case.*
  else:
    send responsively $(\texttt{LeakGroup}, (\mathsf{G}, \mathsf{n}, \mathsf{g}))$ **to** NET; {*Leak the group parameters to the adversary. Note that this command forces the adversary to respond such that the run can continue as expected.*
    wait for _.

Main: {*Note that* **Main** *continues processing the message m that* **Initialization** *has already parsed if* **Initialization** *does not end the run.*

recv $(\texttt{InitKE}, intendedPartner)$ **from** I/O **s.t.** $\mathsf{state}[\mathsf{entity}_{\mathsf{cur}}] = \bot$:
  $\mathsf{state}[\mathsf{entity}_{\mathsf{cur}}] \leftarrow \texttt{started}$.
  $\mathsf{caller}[\mathsf{entity}_{\mathsf{cur}}] \leftarrow \mathsf{entity}_{\mathsf{call}}$.
  $\mathsf{intendedPID}[\mathsf{entity}_{\mathsf{cur}}] \leftarrow intendedPartner$.
  send $(\texttt{InitKE}, intendedPartner)$ **to** NET.

recv $(\texttt{GroupSession}, entity_I, entity_R)$ **from** NET:
  Parse and check the following:
    $entity_I = (pid_I, sid_I, \texttt{initiator}) \wedge entity_R = (pid_R, sid_R, \texttt{responder})$
  $\wedge$ $(\mathsf{intendedPID}[entity_I] = pid_R \vee entity_I \in \mathsf{CorruptionSet})$ {*If an entity is not corrupted: ensure that its partner is correct.*
  $\wedge$ $(\mathsf{intendedPID}[entity_R] = pid_I \vee entity_R \in \mathsf{CorruptionSet})$
  $\wedge$ $(\mathsf{state}[entity_I] = \texttt{started} \vee entity_I \in \mathsf{CorruptionSet})$ {*If an entity is not corrupted: ensure that it has started the exchange and is not already in another session.*
  $\wedge$ $(\mathsf{state}[entity_R] = \texttt{started} \vee entity_R \in \mathsf{CorruptionSet})$.
  if all checks succeed:
    Choose $k \leftarrow \mathsf{G}$ uniformly at random.
    $\mathsf{sessionKeys}[entity_I] \leftarrow k$.
    $\mathsf{sessionKeys}[entity_R] \leftarrow k$.
    Add $(entity_I, entity_R)$ to sessions.
    $\mathsf{state}[entity_I] \leftarrow \texttt{inSession}$.
    $\mathsf{state}[entity_R] \leftarrow \texttt{inSession}$. {*By this, honest entities cannot be grouped into a second group; however, corrupted ones can be part of many groups.*
    reply $(\texttt{GroupSession}, \texttt{success})$.
  else:
    reply $(\texttt{GroupSession}, \texttt{failed})$.

recv $(\texttt{FinishKE}, k)$ **from** NET **s.t.** $\mathsf{state}[\mathsf{entity}_{\mathsf{cur}}] = \texttt{inSession}$:
  Find the partner $entity_p$ of $\mathsf{entity}_{\mathsf{cur}}$ in sessions. {*Note that $entity_p$ is uniquely defined as $\mathsf{entity}_{\mathsf{cur}}$ is honest.*
  if $entity_p \in \mathsf{CorruptionSet}$:
    $\mathsf{sessionKeys}[\mathsf{entity}_{\mathsf{cur}}] \leftarrow k$. {*Adversary may choose the key if he has corrupted the partner.*
  $\mathsf{state}[\mathsf{entity}_{\mathsf{cur}}] \leftarrow \texttt{finished}$.
  send $(\texttt{FinishKE}, \mathsf{sessionKeys}[\mathsf{entity}_{\mathsf{cur}}])$ **to** $\mathsf{caller}[\mathsf{entity}_{\mathsf{cur}}]$.

recv $(\texttt{Register}, pk)$ **from** NET:
  send $(\texttt{Register}, pk)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \epsilon, \mathcal{F}_{\text{CA}} : \texttt{registration})$; {*Allow the adversary to register arbitrary keys in $\mathcal{F}_{\text{CA}}$ for honest entities. Note that $\mathcal{F}_{\text{KE}}$ provides security for session keys independently of any keys stored in $\mathcal{F}_{\text{CA}}$, so there is no harm in giving the adversary full access also for honest entities.*
  wait for $(\texttt{Register}, response)$.
  reply $(\texttt{Register}, response)$.

Fig. 11: The ideal key exchange functionality $\mathcal{F}_{\text{KE}}$.

Appendix A
Notation

In the following we present a compact yet formally complete and unambiguous syntax for detailing the different blocks of the protocol specification template presented in Figure 4.

A. General notation

We recommend to use `typewriter font` for strings, sans serif font for global variables (i.e., variables that are persistent across multiple activations of the same instance of a machine), *italic font* for local (i.e., ephemeral) variables, and **bold font** for keywords (e.g., for sending or receiving).

B. Special variables

For notational convenience, each instance maintains two framework-specific global variables, namely $\mathsf{entity}_{\mathsf{cur}} = (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ and $\mathsf{entity}_{\mathsf{call}} = (\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})$, both consisting of a PID, an SID, and a role. The former triplet, $\mathsf{entity}_{\mathsf{cur}}$, specifies the entity that was accepted in the **CheckAddress** mode and thus the entity that is currently being processed as the receiver of an incoming message. If the current activation is due to a message received on the I/O interface, then $\mathsf{entity}_{\mathsf{call}}$ specifies the entity that called the current entity by sending that message.

C. CheckID

To specify that an instance of a machine should manage all entities for a single party, one can use the informal statement "accept all entities with the same PID". This is short for the following formal algorithm: Let $(pid, sid, role)$ be the input of **CheckID**. If no entity has been accepted yet, then output `accept`. Otherwise, let $(pid_1, sid_1, role_1)$ be the first entity that was accepted at some point in the past. Output `accept` iff $pid = pid_1$; output `reject` otherwise.

The informal statements "accept all entities with the same SID", "accept all entities with the same SID and role", ... are interpreted in an analogous way.

D. Receiving inputs

The algorithms **Initialization**, **EntityInitialization**, **MessagePreprocessing**, and **Main** generally have to process incoming messages for various entities. We structure these algorithms as a sequence of blocks, where each block is of the generic form:

recv ⟨message pattern⟩ **from** ⟨sender⟩ **to** ⟨receiver⟩ **s.t.** ⟨condition⟩ : ⟨code⟩

Upon receiving an input, the instance sequentially checks whether the input matches one of these specifications and executes first matching block. In particular, the ordering of blocks influences the behaviour of an instance if a message fits multiple blocks as only the first one is executed.

- The ⟨message pattern⟩ describes the format of the message accepted by this code block. It is built from local variables, global variables, strings, and special characters such as "(", ")", ",", "_", and "⊥". To compare a message m to a pattern, first the values of all global and, if already defined, local variables are inserted into the pattern. The resulting pattern p is then compared to m, where undefined local variables match to arbitrary bit strings. If a block is entered after a successful match, then undefined local variables are initialized with the bit strings they matched with. We use the special symbol "_" in patterns to match with arbitrary bit strings, just as for undefined local variables, but without storing the results for later use.
- The ⟨sender⟩ is either the constant bit string `NET` if a message is to be received on the network interface, the constant bit string `I/O` if a message is to be received on the I/O interface, or of the form $(pid_{snd}, sid_{snd}, role_{snd})$ if a message is to be received from a specific sender entity on the I/O interface. In the latter case, *pid* and *sid* can be constructed just as a message pattern, whereas *role* is either a fixed bit string of a (known) subroutine or a variable denoting a connection on the I/O interface to some other (higher-level) protocol where the exact role name is not known. If a concrete bit string is given, then, for better readability, authors are encouraged to prefix *role* with the protocol it belongs to, e.g., "$\mathcal{F}_{\text{sig}}$: `signer`".
  Again, "_" can be used as a wildcard symbol.
- The ⟨receiver⟩ is an entity $(pid_{rcv}, sid_{rcv}, role_{rcv})$ that denotes the intended receiving entity of a message and is built analogous to $(pid_{snd}, sid_{snd}, role_{snd})$ from above.
- In ⟨condition⟩ one can define arbitrary further conditions, e.g., depending on the current state of the instance, that need to be satisfied in order to enter the subsequent code block.
- Finally, ⟨code⟩ specifies arbitrary PPT code that will be executed.

To omit unnecessary syntax in the above generic pattern of blocks, the ⟨condition⟩ and ⟨receiver⟩ parts can be omitted, if no additional conditions need to be fulfilled to accept a message or if the message shall be accepted by this block for any receiver entity, respectively. If there is only one block that is *always* entered for all incoming messages, then we omit the header altogether and just give ⟨code⟩.

E. Sending outputs

The activation of an instance ends when it sends outputs on one of its interfaces, or aborts with a special **abort** command in which case the environment gets activated by definition of the IITM model.

Send-commands are part of the ⟨code⟩ block introduced above and follow the general format:

send ⟨message pattern⟩ **from** ⟨sender⟩ **to** ⟨receiver⟩.

where all parts are as for receiving messages, but with swapped semantics for sending and receiving entities (e.g., `NET` can now only occur as ⟨receiver⟩).

Analogous to before, ⟨sender⟩ can be omitted in the regular case where the message is sent in the name of the currently active entity, i.e., $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$. If in addition the ⟨receiver⟩ is the same as the original callee, i.e., $(\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})$, then one can use the following shorthand notation:

reply ⟨message pattern⟩.

Waiting for Immediate Replies: Often, one would like to obtain some information from, e.g., a subroutine and then continue the computation where it left of. To accommodate this need, we complement the above generic commands with two additional constructions:

- The following command allows an instance to send output to a receiver and wait for a response of a specific format:

  send ⟨message pattern out⟩ **from** ⟨*sender*⟩ **to** ⟨*receiver*⟩;
  wait for ⟨message pattern in⟩ **s.t.** ⟨*condition*⟩.

  Upon receiving the correct response from ⟨*receiver*⟩, the computation continues where it stopped, even preserving local variables. All other incoming messages will be dropped by this instance, ending the activation immediately (except for some framework specific meta messages related to corruption and initialization which are still processed to allow for, e.g., corruption of entities. See Appendix F-B6 for more details). In other words, the instance is "stuck" until it receives the expected response. As this can easily disrupt the protocol execution if the receiver does not answer immediately, this command should be used only sparsely and with special care.
  As before, ⟨sender⟩ can be omitted and will by default be set to $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ in this case.
- In line with Camenisch et al. [21], we support responsive environments and adversaries where one can send a restricting messages on the network interface that force the adversary to immediately return an answer (i.e., before any other interaction with the protocol can occur). One can do so via the following command:

  send responsively ⟨message pattern out⟩ **from** ⟨*sender*⟩;
  wait for ⟨message pattern in⟩ **s.t.** ⟨*condition*⟩.

  where the ⟨*sender*⟩ can again be omitted. This command sends such a restricting message and receives the response; if the response does not match the expected criteria, then the initial message is repeated until the response is accepted. This is a useful contruction in many cases where meta messages need to be exchanged with the adversary, e.g., during initialization steps of an instance, cf. [21] for details.

F. Macros

We provide the following macros that can be used in algorithms:

- One can obtain the corruption status of a subroutine entity $(pid_{sub}, sid_{sub}, role_{sub})$ via the following macro:

  corr$(pid_{sub}, sid_{sub}, role_{sub})$

  Formally, this macro sends the special message `CorruptionStatus`? to $entity_{sub}$, which will respond with the corruption status of the entity (note that, by default, the response to this request is immediate. In particular, control is returned to the caller of **corr** even if $entity_{sub}$ is corrupted). The macro then outputs this response.
- One can initialize a subroutine entity $(pid_{sub}, sid_{sub}, role_{sub})$ via the following macro:

  init$(pid_{sub}, sid_{sub}, role_{sub})$

  More specifically, this sends a special message `InitEntity` to $entity_{sub}$, which will then run **Initialization**, **EntityInitialization**, and determine its initial corruption status (steps that have already been executed before are skipped). Importantly, $entity_{sub}$ will always return control to the caller of **init**, even if it gets corrupted, such that the computation can continue as expected.

We provide a formal definition of these macros in Appendix F-B6.

Appendix B
Security Proof of our Case Study

In this section we provide a proof sketch of Theorem 3, i.e., we show that the real key exchange $\mathcal{R} := (\mathcal{P}_{\text{KE}}, \mathcal{F}_{\text{CA}} : \texttt{retrieval} \mid \mathcal{F}_{\text{sig-CA}}, \mathcal{F}_{\text{CA}} : \texttt{registration})$ realizes the ideal key exchange $\mathcal{I} := (\mathcal{F}_{\text{KE}}, \mathcal{F}_{\text{CA}} : \texttt{retrieval} \mid \mathcal{F}_{\text{CA}} : \texttt{registration})$. As part of this, we define a responsive simulator $\mathcal{S}$ such that the real world running $\mathcal{R}$ is indistinguishable from the ideal world running $\{\mathcal{S}, \mathcal{I}\}$ for every ppt environment $\mathcal{E}$.

First note that it is easy to see that that both $\mathcal{R}$ and $\mathcal{I}$ are R-environmentally bounded and complete. Now, the simulator $\mathcal{S}$ is defined as follows: the simulator is a single machine that is connected to $\mathcal{I}$ and the environment via their network interfaces. In a run, there is only a single instance of the machine $\mathcal{S}$ that accepts all incoming messages. The simulator $\mathcal{S}$ internally simulates $\mathcal{R}$, including its behavior on the network interface to the environment. More precisely, the simulation runs as follows:

- At the start of a run, $\mathcal{S}$ obtains the group parameters used by $\mathcal{F}_{\text{KE}}$: if the simulator is activated for the first time via the $(\texttt{LeakGroup}, (\mathsf{G}, \mathsf{n}, \mathsf{g}))$ message, then he simply saves this message and returns `ok`. Otherwise, $\mathcal{S}$ sends an `InitGroup` message to (an arbitrary entity of) $\mathcal{F}_{\text{KE}}$ to trigger **Initialization** and obtain the group parameters. Note that the environment cannot observe whether the simulator has manually triggered **Initialization** of $\mathcal{F}_{\text{KE}}$. The group parameters are used by $\mathcal{S}$ as output of the internally simulated $\mathcal{P}_{\text{KE}} : \texttt{setup}$ role.
- Upon receiving a $(\texttt{InitKE}, entity_{partner})$ from an honest[11] entity $entity_{sender}$, $\mathcal{S}$ forwards this message in

[11] We consider an entity to be honest if it outputs `false` upon `CorruptionStatus`? requests. Conversely, a we call an entity corrupted if it outputs `true`, even if it was not explicitly corrupted.

the name of a higher-level protocol to the simulated entity $entity_{sender}$ in $\mathcal{R}$. This triggers the start of a key exchange in the simulation.

- As soon as an honest entity $entity_1 = (pid_1, sid_1, role_1)$ in a public role of $\mathcal{R}$ outputs $(\texttt{FinishKE}, k)$, the following happens:
  - If $role_1$ is `initiator`, then $\mathcal{S}$ looks for a simulated entity $entity_2 = (pid_2, sid_2, role_2)$ such that $role_2$ is `responder` and both entities agree on the public Diffie-Hellman key shares of each other. Then $\mathcal{S}$ sends $(\texttt{GroupSession}, entity_1, entity_2)$ to some honest entity[12] in $\mathcal{I}$, waits for the response, and then sends $(\texttt{FinishKE}, k)$ to $entity_1$ in $\mathcal{I}$. This causes the initiator to output a session key.
  - If $role_1$ is `responder`, then $\mathcal{S}$ sends $(\texttt{FinishKE}, k)$ to $entity_1$ in $\mathcal{I}$. This causes the responder to output a session key.
- Every time a public key is registered for some PID and SID in $\mathcal{F}_{\text{CA}}$ in the simulated $\mathcal{R}$, $\mathcal{S}$ registers the same key for the same PID and SID in $\mathcal{I}$ by sending a suitable `Register` request to an honest entity of $\mathcal{F}_{\text{KE}}$.
- All network communication from the environment is forwarded to corresponding entities in the internally simulated $\mathcal{R}$, and vice versa.
- $\mathcal{S}$ keeps the corruption states of entities in public roles of $\mathcal{I}$ and the same entities in the internal simulation of $\mathcal{R}$ in sync. In particular, if such an entity of $\mathcal{I}$ asks for its initial corruption status, then the same entity in $\mathcal{R}$ is simulated to also do so. Furthermore, as soon as a simulated entity in a public role of $\mathcal{R}$ considers itself to be corrupted (either explicitly or due to a corrupted subroutine), $\mathcal{S}$ corrupts the same entity in $\mathcal{I}$. Note that we do not have to care about entities in private roles as those cannot be accessed by the environment anyway.
- If a corrupted entity *entity* in a public role of $\mathcal{R}$ outputs a message on its I/O interface to a higher-level protocol, then $\mathcal{S}$ instructs the (explicitly) corrupted entity *entity* in $\mathcal{I}$ to forward the same message on its I/O interface. The same is also done in the other direction.
- If any error occurs while running the above steps, $\mathcal{S}$ aborts (and thus fails the simulation).

This concludes the description of the simulator. It is easy to see that $\{\mathcal{S}, \mathcal{I}\}$ is R-environmentally bounded and $\mathcal{S}$ is responsive for $\mathcal{I}$ as long as $\mathcal{S}$ aborts only with negligible probability while running with $\mathcal{I}$ and a responsive environment; we show that this is indeed the case as part of the following proof as this property is also necessary for showing indistinguishability. Now, let $\mathcal{E}$ be an arbitrary responsive environment. We argue in two steps that $\mathcal{E}$ cannot distinguish $\mathcal{R}$ and $\{\mathcal{S}, \mathcal{I}\}$.

Step 1: We start by considering a protocol $\mathcal{I}'$ that behaves as $\mathcal{I}$ but always allows outputs the session key that is provided by the simulator, even if two honest entities are combined into a session. Then $\mathcal{R}$ and $\{\mathcal{S}, \mathcal{I}'\}$ are indistinguishable for $\mathcal{E}$ due to the following:

Observe that $\mathcal{S}$ already simulates $\mathcal{R}$ perfectly on the network interface. In particular, upon corruption of a simulated public entity, $\mathcal{S}$ obtains sufficient information from corrupting the corresponding entity in $\mathcal{I}'$ to compute the leakage of the simulated entity. Furthermore, corrupted entities are also simulated perfectly on the I/O interface: firstly, $\mathcal{S}$ is indeed able to keep the corruption states of public entities in sync as $\mathcal{I}'$ does not impose any limitations on when corruption can occur. Secondly, once a public entity has been corrupted, $\mathcal{S}$ has full control over the I/O interface. The only case where the simulation might fail (and potentially trigger an abort) is while handling `FinishKE` responses from honest entities in the internal simulation: (i) in the case of an `initiator` entity, there might not be a suitable `responder` entity that can be grouped into a session, and (ii) in the case of a `responder` entity, it might not have been grouped into a session yet. We now argue that both cases do not occur with more than negligible probability.

Case (i): let $entity_{init}$ be an `initiator` entity in the simulated $\mathcal{R}$ that outputs a `FinishKE` message while being honest. Let $pid_{intended}$ be the PID of the intended partner. We have to show that $\mathcal{S}$ can indeed find a `responder` entity that can be partnered with $entity_{init}$, in which case the simulation succeeds. Since $entity_{init}$ is honest, we have that the signing keys belonging to pid_{init} and $pid_{intended}$ must be uncorrupted. As `FinishKE` is output only after a valid signature from $pid_{intended}$ on $(\mathsf{h}_{\mathsf{init}}, \mathsf{h}_{\mathsf{resp}}, pid_{init})$ is received (and $\mathcal{F}_{\text{CA}}$ outputs the correct public key of $pid_{intended}$), this implies that there is at least one honest entity $entity_{intended}$ of $pid_{intended}$ that has signed this share and has the intended partner pid_{init}. We still need the following properties of $entity_{intended}$ for the pairing to succeed:

1. $entity_{intended}$ is a `responder`.
2. $entity_{intended}$ has not been grouped already (and thus is still in the state `started`).

Both properties hold true with overwhelming probability:

1. Suppose by contradiction that $entity_{intended}$ was an `initiator`. Then, the signature was created during the third protocol step. Since $entity_{intended}$ is honest, this implies that it has previously received a signature on the message $(\mathsf{h}_{\mathsf{resp}}, \mathsf{h}_{\mathsf{init}}, pid_{intended})$ signed by an honest entity of pid_{init} that has the public DH key share $\mathsf{h}_{\mathsf{init}}$. As honestly generated DH key shares collide with negligible probability only, there is only one such entity, namely, $entity_{init}$ with overwhelming probability. However, $entity_{init}$ has not signed any messages yet. Thus, $entity'$ is a `responder` with overwhelming probability.
2. By definition of $\mathcal{S}$, honest `responder` entities are only paired if there is an honest `initiator` entity that has accepted the signature on the second protocol message

[12] The exact entity does not matter for this request, however, it must be honest such that **Main** in $\mathcal{I}$ is actually executed.

$(\mathsf{h}_{\mathsf{init}}, \mathsf{h}_{\mathsf{resp}}, pid_{init})$. This message is only accepted by honest entities that have the public DH key share $\mathsf{h}_{\mathsf{init}}$. By the same argument as above, with overwhelming probability $entity_{init}$ is the only entity that fits this description and thus $entity_{intended}$ has not already been grouped with a different entity.

As there are only polynomially many entities in every run (as the environment has only polynomial runtime), this implies that $\mathcal{S}$ succeeds with its simulation in case (i) with overwhelming probability.

Case (ii): let $entity_{resp}$ be a `responder` entity in the simulated $\mathcal{R}$ that outputs a `FinishKE` message while being honest. Let $pid_{intended}$ be the PID of the intended partner. We have to show that this entity has already been grouped with an (honest) `initiator` entity, in which case the simulation succeeds. Since $entity_{resp}$ is honest, we have that the signing keys belonging to pid_{resp} and $pid_{intended}$ must be uncorrupted. As `FinishKE` is output only after a valid signature from $pid_{intended}$ on $(\mathsf{h}_{\mathsf{resp}}, \mathsf{h}_{\mathsf{init}}, pid_{resp})$ is received, this implies that there is at least one honest entity $entity_{intended}$ of $pid_{intended}$ that has signed this share and has the intended partner pid_{resp}. We still have to argue the following properties of $entity_{intended}$:

1. $entity_{intended}$ is an `initiator`.
2. $entity_{intended}$ has been grouped with $entity_{resp}$.

Both properties hold true with overwhelming probability:

1. Suppose by contradiction that $entity_{intended}$ was a `responder`. Then, the signature was created after receiving the public DH key share $\mathsf{h}_{\mathsf{resp}}$ in the first message and subsequently generating a fresh DH key share $\mathsf{h}_{\mathsf{init}}$. As DH key shares of honest instances collide with other (previously existing) DH shares only with negligible probability, this implies that $entity_{resp}$ has generated its public DH key share before $entity_{intended}$ has received it (with overwhelming probability). Thus, $entity_{resp}$ has already received the first message containing $\mathsf{h}_{\mathsf{init}}$ before $entity_{intended}$ has honestly generated $\mathsf{h}_{\mathsf{init}}$, which is a contradiction with overwhelming probability. So $entity_{intended}$ is a `initiator` with overwhelming probability.
2. As $entity_{intended}$ is an `initiator`, it must have already output a session key after accepting a signature on the message $(\mathsf{h}_{\mathsf{init}}, \mathsf{h}_{\mathsf{resp}}, pid_{init})$ signed by pid_{resp}. Thus, by definition of $\mathcal{S}$, it is indeed already grouped. Furthermore, as the signing key of pid_{resp} is uncorrupted and honestly generated DH key shares collide with negligible probability, we have that there is only one entity that signs such a message, namely, $entity_{resp}$ (with overwhelming probability). In other words, $entity_{intended}$ and $entity_{resp}$ are already partnered.

As there are only polynomially many entities in every run, this implies that $\mathcal{S}$ also succeeds with its simulation in case (ii) with overwhelming probability.

Overall, we have that the systems $\mathcal{R}$ and $\{\mathcal{S}, \mathcal{I}'\}$ are indistinguishable for every responsive environment as the simulation of $\mathcal{S}$ is perfect with overwhelming probability.

Step 2: We now show that the systems $\{\mathcal{S}, \mathcal{I}'\}$ and $\{\mathcal{S}, \mathcal{I}\}$ are indistinguishable for $\mathcal{E}$, which concludes the proof. Observe that the only difference between both systems is how the session keys are chosen for sessions (groups) consisting of two *honest* entities. In this case, $\mathcal{I}$ chooses a uniformly random key g^c, whereas $\mathcal{I}'$ uses the key from the simulator $\mathcal{S}$, which is g^{ab}.

To show indistinguishability, one uses a standard hybrid argument that replaces the keys g^{ab} of honest sessions with an ideal key g^c in the order of their occurrence. The distinguishing advantage between each of the hybrid steps can be upper bounded via a reduction to the DDH assumption. Note that this reduction indeed works: keys are replaced only for sessions consisting of honest entities. Since we consider static corruption of the real protocol, honest entities will not get corrupted later on and thus will never reveal their secret exponents. Furthermore, the whole system can be simulated in polynomial time. Thus, the system can be simulated by a ppt distinguisher on the DDH game without knowing the secret exponents. Overall, as there are only a polynomial number of hybrid steps (as the environment can only create a polynomial number of sessions), each of which is upper bounded by the same negligible function, this hybrid argument implies that $\{\mathcal{S}, \mathcal{I}'\}$ and $\{\mathcal{S}, \mathcal{I}\}$ are indistinguishable for $\mathcal{E}$. □

Appendix C
Example: Single Session Analysis

This section illustrates how a single-session security analysis, which was explained on a high-level in §V-A, can be performed in iUC. As part of this, we provide a concrete example of a protocol that has disjoint sessions and can thus be analyzed for just a single-session.

We start by defining (disjoint) protocol sessions and of a protocol $\mathcal{P}$ in iUC. For this purpose, we introduce a function σ, called a *protocol session ID (PSID) function*, that groups entities of $\mathcal{P}$ into protocol sessions. On a high level, the function σ takes as input an entity and assigns it a PSID. A session is then defined via a single PSID *psid* and encompasses all entities with that *psid*. Intuitively, the sessions of a protocol $\mathcal{P}$ are disjoint (according to σ) if instances accept entities only for a single PSID and send messages only to other entities with the same PSID. Thus, entities cannot share state directly or indirectly with entities from other protocol sessions. We note that the concepts of SIDs and PSIDs are disjoint from each other: an SID is used to denote multiple runs of some party *pid* in some role *role*, whereas a PSID denotes the set of all entities that form a global protocol session. While it is possible to define a protocol session *psid* to contain exactly those entities that share some fixed SID *sid*, a protocol session can also, e.g., contain entities with multiple different SIDs. We provide an example for the latter case at the end of this section.

More formally, protocol session functions and protocols with disjoint sessions are defined as follows.

Definition 2 (PSID function)**.** A function $\sigma : (\{0,1\}^*)^3 \to \{0,1\}^* \cup \{\bot\}$ is called *PSID function* if it is computable in polynomial time (in the length of its input).

Definition 3 (Protocols with disjoint sessions)**.** Let σ be a PSID function and let $\mathcal{P}$ be a complete protocol. We say that $\mathcal{P}$ has *disjoint sessions (according to σ)*, or $\mathcal{P}$ is a *σ-session protocol*, if in all runs of the system $\{\mathcal{E}, \mathcal{P}\}$ (for an arbitrary environment $\mathcal{E} \in \mathsf{Env}_R(\mathcal{P})$ that interacts with the I/O interfaces of public roles of $\mathcal{P}$ and all network interfaces) the following holds true for every machine M in $\mathcal{P}$:

1. M will never accept (via the **CheckID** algorithm) an entity $(pid, sid, role)$ with $\sigma(pid, sid, role) = \bot$.
2. If $(pid, sid, role)$ is the first entity that an instance of M accepted, then this instance rejects all following entities $(pid', sid', role')$ where $\sigma(pid, sid, role) \neq \sigma(pid', sid', role')$.
3. Let $(pid, sid, role)$ be the first entity that an instance of M accepted. If this instance sends a message m, then the message is sent in the name of an entity $(pid', sid', role')$ such that $\sigma(pid, sid, role) = \sigma(pid', sid', role')$. Furthermore, if m is sent on a connection to some role $role''$ in $\mathcal{P}$, then this message is sent to an entity $(pid'', sid'', role'')$ such that $\sigma(pid, sid, role) = \sigma(pid'', sid'', role'')$.

We can analyze a single session of a σ-session protocol $\mathcal{P}$ in isolation to obtain security for an unbounded number of sessions of $\mathcal{P}$. We use a special type of environment to define such a single session security analysis. A so-called *single-session environment* (for a PSID function σ) may send messages to entities with the same PSID only (according to σ) only. More specifically, the environment may not send a message to an entity *entity* with $\sigma(entity) = \bot$ and, if *entity* is the first entity that the environment sends a message to, then all following messages are sent to entities *entity*$'$ such that $\sigma(entity) = \sigma(entity')$. Hence, such an environment may invoke one session of the protocol $\mathcal{P}$ only. We denote the set of all single-session (responsive and universally bounded) environments for a Protocol $\mathcal{P}$ by $\mathsf{Env}_{R,\sigma\text{-single}}(\mathcal{P})$.

We can now define the single session realization relation and state the unbounded self-composition theorem in iUC (an informal version of this theorem was given as Corollary 6 in §V). Both the relation relation and the composition theorem are natural translations of the corresponding statements in the IITM model to the iUC framework.

Definition 4 (Single session realization relation)**.** Let σ be a PSID function, and let $\mathcal{P}$ and $\mathcal{F}$ be two R-environmentally bounded complete σ-session protocols with identical sets of public roles. We say that $\mathcal{P}$ *single-session realizes* $\mathcal{F}$ ($\mathcal{P} \leq_{\sigma\text{-single}} \mathcal{F}$) if there exists a simulator $\mathcal{S} \in \mathsf{Adv}_R(\mathcal{F})$ such that $\{\mathcal{E}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$ for all $\mathcal{E} \in \mathsf{Env}_{R,\sigma\text{-single}}(\mathcal{P})$.[13]

Corollary 9 (Unbounded self-composition theorem)**.** Let σ be a PSID function, and let $\mathcal{P}$ and $\mathcal{F}$ be two protocols such that $\mathcal{P} \leq_{\sigma\text{-single}} \mathcal{F}$. Then $\mathcal{P} \leq \mathcal{F}$.

Proof. This follows from the unbounded self-composition theorem in the IITM model [21]. However, unlike Corollary 2, it is not a trivial instantiation but rather requires a short argument. This is because the definitions of PSID functions and σ-session protocols in iUC are slightly different from how they are defined in the IITM model, so we have to relate the notions from iUC to those in the IITM model. We provide a full proof in Appendix G. □

We now illustrate how protocols with disjoint sessions can be modeled in iUC by giving an example. More specifically, we model the standard case that is considered in the UC and GNUC models where a single session of a single highest-level protocol is analyzed in isolation. This single session can use potentially several instances of arbitrary subroutines, as long as no instance is accessible by two or more different sessions. In our framework, we model this setting by considering a combined protocol $\mathcal{R} := \mathcal{P} \,||\, \mathcal{S}_1 \,||\, \ldots \,||\, \mathcal{S}_n$ consisting of a highest-level protocol $\mathcal{P}$ with several subroutine protocols $\mathcal{S}_i$.

We define a protocol session of $\mathcal{R}$ via the SID used by entities in the highest-level protocol $\mathcal{P}$. That is, an entity $(pid, sid, role)$ of $\mathcal{P}$ runs in the protocol session $psid := sid$. The SIDs of entities in subroutines consist of two parts, a prefix and a suffix, where the prefix is the actual protocol session that they run in and the suffix allows for arbitrarily many copies of a subroutine within the same session. That is, an entity of $\mathcal{S}_i$ has the form $(pid, (sid_{pre}, sid_{suf}), role)$ and runs in session $psid := sid_{pre}$. This directly implies a definition of a PSID function σ which, in particular, is computable in polynomial time: $\sigma(pid, sid, role)$ checks whether *role* is in $\mathcal{P}$ or $\mathcal{S}_i$ and then either outputs *sid* or the prefix of *sid*. In all other cases (e.g., if there is no prefix in the SID in case of a subroutine) σ outputs $\bot$.

Now, (instances of) machines in $\mathcal{R}$ have to meet the three properties of Definition 3 in order to have disjoint sessions: (i) they may not accept entities that belong to no protocol session (i.e., where σ outputs $\bot$), (ii) they never accept entities from two different protocol sessions, and (iii) senders of messages are in the correct session and receivers, if they are part of $\mathcal{R}$, are in the same session. We ensure these properties as follows:

(i) The **CheckID** algorithm is used to ensure that the SIDs of entities in subroutines have the expected format. That is, subroutine machines accept only entities that have a prefix in their SID. Thus, no

[13] Note that both $\mathcal{F}$ and $\mathcal{P}$ use the same PSID function σ to define disjoint sessions. Thus, they agree in their "behavior" for the shared public roles, i.e., entities of those roles are grouped into protocol sessions in the same way.

machine in $\mathcal{R}$ accepts an entity that does not have a PSID.

(ii) This can also be enforced via the **CheckID** algorithm. More specifically, a machine saves the first entity that it has accepted and then accepts following entities only if they have the same PSID as the first one. That is, they either have the same SID (in the case of entities in $\mathcal{P}$) or the same SID prefix (in the case of entities in $\mathcal{S}_i$).

(iii) This property is straightforward to ensure via suitable definitions of the various algorithms in our template. More specifically, every **send** command in each of those algorithms must be defined such that senders and receivers of this message meet this condition (note that this includes the macros from Appendix A, which internally send messages). Furthermore, we use the **AllowAdvMessage** algorithm to prevent the adversary from breaking this condition for corrupted entities.

We provide formal definitions of the components of $\mathcal{R}$ following these guidelines in Figures 12 and 13.

We directly obtain that a protocol $\mathcal{R}$ that is constructed as described above (cf. Figures 12, 13) is a σ-session protocol. Thus we can use Corollary 9 to obtain the following: Let $\mathcal{R}$ and $\mathcal{I}$ be two σ-session protocols that are constructed as described above (for the same σ). If $\mathcal{R} \leq_{\sigma\text{-single}} \mathcal{I}$ then $\mathcal{R} \leq \mathcal{I}$. That is, it is sufficient to analyze and compare a single protocol session of $\mathcal{R}$ with a single protocol session of $\mathcal{I}$ (and a simulator) to obtain security for arbitrarily many sessions running concurrently. We note that we did not fix which roles are public and private in $\mathcal{R}/\mathcal{I}$, i.e., this argument also works even if some of the subroutine protocols have public roles that the environment can access.

Appendix D
Joint-State and Corruption in the UC Model

In this section, we validate the claim from §V-B that the corruption model proposed by the current version of the UC model prevents many types of joint-state realizations. This illustrates how difficult it is to define a corruption model that on the one hand is flexible and reasonable, but on the other hand also supports all expected features of universal composability models such as joint-state composition.

We start by recalling the corruption model proposed by the UC model: The UC model defines several types of corruption for so-called real protocols (as well as the real parts of so-called hybrid protocols that contain both real and ideal protocols), all of them following a similar structure. The adversary on the network may send a special corruption request to any instance of a real protocol. Upon receiving such a request and validating that corruption is allowed at this point in time, the real protocol sends a corruption notification that includes its own identity (PID and SID) to all currently known higher-level protocols. If a protocol receives such a notification from a subroutine, it forwards the notification to its own higher-level protocols, and so on until the notification reaches the environment. Afterwards, for most corruption types (except honest-but-curious corruption) the corrupted protocol acts as a pure message forwarder to/from the adversary from/to other protocols, just as in our framework. While the UC model does not propose any corruption model for ideal protocols, we assume in the following that ideal protocols are defined such that the simulator also has some way to corrupt a party in a session, which in turn triggers corruption notifications and subsequently allows forwarding messages to/from the environment; ideal protocols that do not provide this behavior cannot be realized using the UC corruption model for real protocols anyway (independently of joint-state).

Now, consider the standard case in the UC model of an ideal protocol $\mathcal{I}$ that uses one instance per session (i.e., per SID) such that instances do not interact with each other and do not share any state. Let $\mathcal{P}_{js}$ be some potential joint-state realization that uses a single instance per party (i.e., per PID) to realize *all* sessions (i.e., SIDs) of that party such that this single instance can re-use some shared state (or a single subroutine) across those sessions. Assume that $\mathcal{P}_{js}$ uses the corruption model proposed by the UC model. In this common and very general setting, there is no simulator such that $\mathcal{P}_{js}$ and $\mathcal{I}$ running with the simulator are indistinguishable: consider a run where the adversary in the real world corrupts a single instance of $\mathcal{P}_{js}$, say for party Alice. Since he can now forward messages in *arbitrary* sessions to the environment, the simulator in the ideal world must corrupt the corresponding instances of $\mathcal{I}$ (at least for party Alice) to be able to do the same. Note that, upon corruption of $\mathcal{P}_{js}$, the simulator cannot at the same time corrupt *all* corresponding instances of $\mathcal{I}$ as there are infinitely many (recall that one instance of $\mathcal{P}_{js}$ handles all sessions, but one instance of $\mathcal{I}$ handles only a single one). So the simulator can corrupt instances of $\mathcal{I}$ only in a delayed fashion, i.e., as soon as he learns a new SID that is used by the environment. However, if he does so, he will trigger a notification to the environment, which does not occur in the real world as the single instance of Alice in $\mathcal{P}_{js}$ was already corrupted earlier. Overall, this implies that there is no suitable simulator and one cannot actually show that $\mathcal{P}_{js}$ realizes $\mathcal{I}$.

In contrast to the UC model, our corruption model is built with joint-state in mind. While the simulator in our framework also has the problem that he has to corrupt infinitely many sessions in the ideal world, he actually *can do so in a delayed manner*: first, note that as soon as a single instance of $\mathcal{P}_{js}$ of Alice gets corrupted, the simulator can corrupt Alice in $\mathcal{I}$ for all already existing sessions (of which there are only polynomially many and which the simulator knows as every new session has already asked for its initial corruption status at some point). Thus, the environment cannot distinguish real and ideal world by requesting the corruption status of sessions that already existed prior to corrupted. If the environment requests the corruption status of a session that did not exist and has not

Structure of a highest-level protocol $\mathcal{P}$ (used in a combined protocol $\mathcal{P} \,||\, \mathcal{S}_1 \,||\, \ldots \,||\, \mathcal{S}_n$) with disjoint sessions:

Participating roles: *arbitrary*
Corruption model: *arbitrary*

For each of the machines M of $\mathcal{P}$:

Implemented role(s): *arbitrary*
CheckID$(pid, sid, role)$**:**
Perform *arbitrary* checks and, potentially, output `reject` based on these checks.
If not other entity has been accepted yet, output `accept`.
Otherwise, let $sid^{accepted}$ be the (full) SID of the first entity that was accepted.
Output `accept` if and only if $sid^{accepted} = sid$.
Corruption behavior:
– **AllowAdvMessage**$(pid, sid, role, pid_{receiver}, sid_{receiver}, role_{receiver}, m)$:
If $role_{receiver}$ is part of $\mathcal{P}$ or a higher-level protocol/the environment, then check that $sid = sid_{receiver}$.
Otherwise, try to parse as $sid_{receiver}$ as (sid_{prefix}, sid') and check that $sid = sid_{prefix}$. {*i.e., the role is specified in* $\mathcal{S}_i$.
If any of the previous steps/checks fails, output `false`. {*Ensure that messages are sent only to the same "session".*
Perform *arbitrary* other checks and output `true` or `false` based on these checks.

Other Corruption behavior algorithms, Initialization, EntityInitialization, MessagePreprocessing, Main:
These algorithms are *arbitrary*, but subject to the restriction that they may only send messages from entities managed by the current instance[a] to entities that are part of the same "session".
In particular, messages must have a correct header (cf. §F-B3). Furthermore, if a message is sent to a higher-level protocol or a role in $\mathcal{P}$, then $sid_{sender} = sid_{receiver}$, and if it is sent to a subroutine role in one of the subroutine protocols $\mathcal{S}_i$, then $sid_{receiver} = (sid_{sender}, sid')$ (for messages sent to the network there is not restriction imposed in the receiver).

[a] i.e., from entities that get accepted by **CheckID**.

Fig. 12: Example structure of a highest-level protocol with disjoint-sessions. Fields/algorithms that are marked as *arbitrary* or that are omitted can be specified freely by the protocol designer without breaking disjoint sessions. See Figure 13 for how subroutine protocols $\mathcal{S}_i$ are defined.

Structure of subroutine protocols $\mathcal{S}_i$ (used in a combined protocol $\mathcal{P} \,||\, \mathcal{S}_1 \,||\, \ldots \,||\, \mathcal{S}_n$) with disjoint sessions:

Participating roles: *arbitrary*
Corruption model: *arbitrary*

For each of the machines M of $\mathcal{S}_i$:

Implemented role(s): *arbitrary*
CheckID$(pid, sid, role)$**:**
Check that $sid = (sid_{prefix}, sid')$; otherwise output `reject`.
Perform *arbitrary* additional checks and, potentially, output `reject` based on these checks.
If not other entity has been accepted yet, output `accept`.
Otherwise, let $sid_{prefix}^{accepted}$ be the prefix of the SID of the first entity that was accepted.
Output `accept` if and only if $sid_{prefix}^{accepted} = sid_{prefix}$.
Corruption behavior:
– **AllowAdvMessage**$(pid, sid, role, pid_{receiver}, sid_{receiver}, role_{receiver}, m)$:
Parse sid as (sid_{prefix}, sid').
If $role_{receiver}$ is part of $\mathcal{P}$, then check that $sid_{prefix} = sid_{receiver}$.
Otherwise, try to parse as $sid_{receiver}$ as (sid_{prefix}, sid''). {*i.e., the role is specified in some* $\mathcal{S}_j$.
If any of the previous steps/checks fails, output `false`. {*Ensure that messages are sent only to the same "session".*
Perform *arbitrary* other checks and output `true` or `false` based on these checks.

Other Corruption behavior algorithms, Initialization, EntityInitialization, MessagePreprocessing, Main:
These algorithms are *arbitrary*, but subject to the restriction that they may only send messages from entities managed by the current instance[a] to entities that are part of the same "session".
In particular, messages must have a correct header (cf. §F-B3). Furthermore, if a message is sent to a role in $\mathcal{P}$, then $sid_{sender} = (sid_{receiver}, sid')$, and if it is sent to a subroutine role in one of the subroutine protocols $\mathcal{S}_i$, then the prefixes of sid_{sender} and $sid_{receiver}$ are identical (for messages sent to the network there is not restriction imposed in the receiver).

[a] i.e., from entities that get accepted by **CheckID**.

Fig. 13: Example structure of subroutine protocols with disjoint-sessions. Fields/algorithms that are marked as *arbitrary* or that are omitted can be specified freely by the protocol designer without breaking disjoint sessions. See Figure 12 for how the highest-level protocol $\mathcal{P}$ is defined.

been initialized yet, then both real and ideal world return `false` by definition. If the environment first initializes a new session (by sending some arbitrary message that is not `CorruptionStatus`?), then the simulator is asked to provide the initial corruption status and is thus able to corrupt that session as soon as it is created. Overall, this allows the simulator to take control of all necessary entities reactively and without providing a way for the environment to check whether corruption occurs in a delayed fashion.

Appendix E
More Details about the IITM Model

In this section, we provide further details about the IITM model with responsive environments which extend the description given in §II. We note that the level of detail given in §II is fully sufficient in order to understand and use the iUC framework; it is not necessary to also read this section. This section is used only to provide additional technical information for defining the technical mapping in Appendix F and proving the unbounded self-composition theorem in Appendix G.

Responsiveness of environments and adversaries.: Formally, restricting messages and responsive environments/adversaries are defined as follows. A so-called *restriction* R defines both restricting messages and possible answers to them; R is a subset of $\{0,1\}^+ \times \{0,1\}^+$ which contains message pairs (m, m') and must be efficiently decidable (see [21] for details). We define $R[0] := \{m \mid (m, m') \in R\}$. A message $m \in R[0]$ is called a *restricting message* and if $(m, m') \in R$, then m' is a possible answer to m. Now, an environmental system $\mathcal{E}$ is *responsive* for a system $\mathcal{Q}$ if for all but a negligible set of runs of $\{\mathcal{E}, \mathcal{Q}\}$ the following is true: If in a run an instance of $\mathcal{Q}$ sends a restricting messages m on the network interface to $\mathcal{E}$, then $\mathcal{E}$ has to answer "immediately" in the following sense: After having received m, the first message m' sent from $\mathcal{E}$ to $\mathcal{Q}$ (if any) has to be sent to the same instance of $\mathcal{Q}$ which sent m and it must hold true that $(m, m') \in R$. Analogously, an adversarial system is *responsive* for a system $\mathcal{Q}$ if the same property holds in runs of $\{\mathcal{E}, \mathcal{A}, \mathcal{Q}\}$ for any environment $\mathcal{E}$ that is responsive for $\{\mathcal{A}, \mathcal{Q}\}$. In other words, the adversary has to answer restricting messages from $\mathcal{Q}$ immediately in the above sense as well. Note that it may, however, contact the (responsive) environment before answering $\mathcal{Q}$. (The adversary should send only restricting messages to $\mathcal{E}$ in this case as otherwise $\mathcal{E}$ would be free to contact $\mathcal{Q}$, violating the responsiveness property.) As shown in [21], an environment which is responsive for a system $\mathcal{Q}$ is also responsive for all systems $\mathcal{Q}'$ indistinguishable from $\mathcal{Q}$.

Unbounded self-composition theorem.: We have already presented the concurrent composition theorem in §II (on a slightly informal level). The IITM model also supports a second composition theorem for the secure composition of an unbounded number of sessions of the same protocol system, given that one session of the protocol system is secure. To state this theorem, following [17], [21], we have to consider protocol systems where instances of machines in these systems have protocol session IDs (PSIDs);[14] instances with the same PSID form a session. We also have to introduce environments which invoke a single session of a protocol only. For this purpose, PSID functions σ and σ-session protocols are introduced.

A *PSID function* σ assigns a PSID (or $\bot$) to every message sent or received on a tape of an IITM. Typically, messages are prefixed with PSIDs, and the PSID function σ simply extracts these PSIDs.

An (instance of an) IITM M is called a *σ-session machine* if, while running in an arbitrary context, it does not accept messages (in mode **CheckAddress**) for which σ outputs $\bot$. Also, if M accepted a message on some tape at some point for which σ returned the PSID $sid \neq \bot$, then later M may only accept messages with the same PSID sid, i.e., for which σ returns sid. Also, M may only output messages with this sid. So, altogether an instance of M can only be addressed by one PSID (the first one M accepts) and this instance only outputs messages with that PSID. A protocol system is called a σ-session protocol if all IITMs in that system are σ-session versions.

A *single-session environment* (for an PSID function σ) may invoke machines with the same protocol session ID (according to σ) only. That is, such an environment may output messages for which σ yields only the same PSID, and hence, it may invoke one session of the protocol only. We denote the set of single-session (responsive and universally bounded) environments for a system $\mathcal{Q}$ by $\mathsf{Env}_{R,\sigma\text{-single}}(\mathcal{Q})$.

We say that $\mathcal{P}$ *single-session realizes* $\mathcal{F}$ ($\mathcal{P} \leq_{\sigma\text{-single}} \mathcal{F}$) if there exists a simulator $\mathcal{S} \in \mathsf{Adv}_R(\mathcal{F})$ such that $\{\mathcal{E}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$ for all $\mathcal{E} \in \mathsf{Env}_{R,\sigma\text{-single}}(\mathcal{P})$. Now, the composition theorem states that if a single session of a real protocol $\mathcal{P}$ realizes a single session of an ideal protocol $\mathcal{F}$, then multiple sessions of $\mathcal{P}$ realize multiple sessions of $\mathcal{F}$.

Theorem 10 (Unbounded self-composition [21])**.** Let R be a restriction, σ be an PSID function, and let the protocol systems $\mathcal{P}$ and $\mathcal{F}$ be σ-session protocols. Then, $\mathcal{P} \leq_{\sigma\text{-single}} \mathcal{F}$ implies $\mathcal{P} \leq \mathcal{F}$.

Appendix F
Formal Mapping of Protocols to ITMs

In this section, we explain how the templates specified in §III are mapped to actual systems in the sense of the IITM model with responsive environments. The resulting systems are instantiations of the protocol systems in the IITM model with responsive environment (see §II, §E, and [21]). Hence, all theorems of the responsive IITM model, including composition theorems, hold true for these systems. This, in particular, shows that iUC inherits the soundness properties of the IITM model.

This section is structured as follows: First, we introduce a low-level syntax for describing ITMs in Appendix F-A.

[14] In the original IITM model, these IDs were called session IDs (SIDs). To avoid confusion with the concept of SIDs used in entities in iUC, we have renamed them to protocol session identifiers. This is in line with the terminology used in iUC for the same concept, see Appendix C.

Then, in Appendix F-B, we explain how a single instance of our template from §III-C is mapped to a system of ITMs, including the exact specification of those ITMs. Finally, in Appendix F-C we explain how a complete protocol, which is defined by one or more instances of our template, is interpreted as a system of ITMs (this mainly entails connecting the individual systems created from each template instance).

A. Notation for the Formal Specification of ITMs

Before being able to formally specify how protocol systems in the sense of the IITM model are obtained from the specifications/templates, we need to introduce some notation for specifying ITMs. This notation is used to formally define the ITMs that are obtained from our template, where by ITMs we mean the (plain) ITMs introduced in §II. We note that this is a rather low-level syntax that need not be used by a protocol designer but is rather only used for presenting the mapping of our template to ITMs. For specifying algorithms in our template, we provide a convenient high-level syntax in Appendix A. Note that the following low-level syntax borrows and adjusts several elements from the high-level syntax such as message patterns.

Message patterns: A message pattern mp is used to describe the format of a message $m \in \{0,1\}^*$. It is built from *local variables* (denoted in italic font) which only exist for a single activation of an algorithm, global variables (denoted in sans-serif font) which are part of the internal state of an instance of a machine and can be accessed across multiple activations of different algorithms of the same instance, `strings` (denoted in typewriter font), and special characters such as "(", ")", "," and "$\bot$".

Message patterns can be used to describe outgoing messages, in the following denoted by mp_{out}, and incoming messages, in the following denoted by mp_{in}. If a message pattern is used for sending, the current values of global and local variables are inserted, while the remainder of the pattern stays as is (in particular, strings and special signs are not altered). The resulting message is then sent. If a message pattern is used for receiving, a message m upon receipt is matched against the pattern: After inserting the values of global variables and, if already defined, those of local variables into mp_{in}, the resulting message must be the same as m except for undefined local variables, which match an arbitrary text. After a successful match, all local variables contain the value that they matched on. The special symbol _ can be used in mp_{in} instead of an undefined local variable if the value that is matched on is not needed afterwards, i.e., _ matches everything but does not store the result.

To illustrate message patterns, consider the case where an instance of an ITM has a global variable id storing the ID of the instance. Such an instance might at some point send a request on the network to the adversary which contains a unique request ID *qid*, which is stored in a local variable (as it is no longer needed once the algorithm has terminated). Such an ID is useful to match responses from the adversary to specific requests. Now, the message pattern $m_{\text{in}} = (\mathsf{id}, (\texttt{Response}, qid, m'))$ can be used to wait for a response from the adversary. This pattern will match any message m which contains the ID id of the instance, the fixed bit string `Response`, the value contained in the local variable *qid*, and an arbitrary bit string which will be stored in a new local variable m' after a successful match.

Sending raw messages: When we write **send** mp_{out} **on** t, we mean that the message m that is created from mp_{out} at runtime is sent on tape t.

Restricting messages: As explained in §II and formally defined in [21], we consider a restriction relation R and responsive environments such that if a real or ideal protocol outputs a restricting message $x \in R[0]$ on a network tape, the environment/adversary/simulator has to send a reply y on the corresponding tape with $(x,y) \in R$ immediately, i.e., without sending an incorrect message (wrong message according to R or wrong tape) to the protocol before. To be precise, we use the following definition of R:

$$
\begin{aligned}
R := \{(m,m') | m = (id, \texttt{CorruptMe?}) \wedge \\
\quad m' = (id, (\texttt{SetCorruptionStatus}, b)) \wedge \\
\quad id \in \{0,1\}^* \wedge b \in \{\texttt{false}, \texttt{true}\}\} \\
\cup \{(m,m') | m = (id, (\texttt{CorrStatusRestrict}, b, m'')) \wedge \\
\quad m' = (id, \texttt{OK}) \wedge \\
\quad id \in \{0,1\}^* \wedge b \in \{\texttt{false}, \texttt{true}\} \wedge \\
\quad m'' \in \{0,1\}^* \cup \{\bot\}\} \\
\cup \{(m,m') | m = (id, (\texttt{Respond}, m'')) \wedge \\
\quad m' = (id, m''') \wedge \\
\quad id \in \{0,1\}^* \wedge m'', m''' \in \{0,1\}^* \cup \{\bot\}\}
\end{aligned}
$$

Note that according to R defined above, a restricting message $m \in R[0]$ and a possible response m' with $(m,m') \in R$ always start with the same ID id. In our framework, instances of machines will use id to store the sending entity of a message. Thus, by the definition of the restriction the response will be sent back to the same entity and thus to the corresponding instance. In particular, the adversary/environment may not interact with any other protocol instances before sending this response (except for negligible probability, which can be ignored in security proofs).

We note that, as also discussed in [21], it would be sufficient to consider a restriction R which contains only the last type of message pairs. That is, one simply indicates a restricting message by adding `Respond` to the message and does not make any restrictions about the response. If one wants an answer that satisfies certain conditions, one can inspect the answer and if it does not satisfy the condition, one can send the restricting message again until one obtains a message that satisfies the condition; see also the command introduced next. However, we chose to consider the above version of R as it makes explicit which responses are permitted for framework specific messages and thus slightly simplifies runs (i.e., we do not have to re-send messages in those cases).

We write **send responsively** mp_{out} **on** $\mathsf{t}_{\mathsf{net}}$; **wait for** mp_{in} **on** $\mathsf{t}_{\mathsf{net}}$ **s.t.** $\langle condition \rangle$ to emphasize that the machine sends a *restricting* message on a network tape $\mathsf{t}_{\mathsf{net}}$ and then waits to receive a response on the same network tape that matches with mp_{in} and satisfies $\langle condition \rangle$. This command will only be used if a message $m \in R[0]$ is sent. We note that the message pattern mp_{in} is usually defined in such a way that it accepts (some of the) possible answers to the restricting message. If an incoming message m' (with $(m, m') \in R$) is not accepted by the command, then the machine repeats the command **send responsively** mp_{out} **on** $\mathsf{t}_{\mathsf{net}}$; **wait for** mp_{in} **on** $\mathsf{t}_{\mathsf{net}}$ **s.t.** $\langle condition \rangle$, i.e., it automatically sends the first message m on $\mathsf{t}_{\mathsf{net}}$ again and waits for an answer that matches mp_{in} and satisfies *condition*. This is repeated until the answer matches mp_{in} and satisfies *condition*. Because of the responsiveness requirement, it is guaranteed that the environment/adversary/simulator has to provide the expected answer to the correct instance if it wants the run to continue.

Abort: We use the special keyword **abort** to say that a machine stops its current activation at some point. More specifically, as soon as a machine in **Compute** mode reaches the **abort** command, it will produce empty output and thus stop its computation. Then, by definition, the master IITM is activated with empty input on the start tape.

B. Mapping Templates

To explain how protocols in our framework can be interpreted as (protocol) systems in the sense of the IITM model, we first have to explain how a single instance of the template in Figure 4 is mapped to a system of ITMs. We start by describing how such a system is structured and then detail the **CheckAddress** and **Compute** modes of the ITMs, including the behavior in case of corruption. Based on the mapping of templates, we then explain in §F-C how a full protocol, potentially defined via several different instances of the template interacting with each other, as well as public and private roles are interpreted as a system in the IITM model. While some of the following has already been sketched in §II and §III, this section provides full details.

1) System of machines: In the following, let $\mathcal{P}$ be a protocol defined by a single instance of the template in Figure 4. Recall from §III that protocols consist of several machines (i.e., ITMs) that implement the roles in this protocol. To be more precise, for each (set of) role(s) defined in the **Participating roles** field, there is one machine that implements this set. Thus, if there are n (sets of) roles in $\mathcal{P}$, then $\mathcal{P}$ is the system $\{M_1, \ldots, M_n\}$. Next, we explain how a machine $M \in \{M_1, \ldots, M_n\}$ given in the tempis defined, where M_i is defined by one part of the template. Let $\{role_1, \ldots, role_m\}$ be the set of roles that M implements, as specified in the **Implemented role(s)** field.

2) Tapes: Recall that each ITM uses a pair of tapes to communicate with another ITM. In the following, for simplicity, we call a pair of related unidirectional tapes with opposite directions a bidirectional tape (or just a tape). Also recall that tapes are grouped into network and I/O tapes.

The machine M (of $\mathcal{P}$) has m network tapes to communicate with the adversary and several I/O tapes to communicate with other protocol machines (both subroutines and high-level protocols) or the environment, each of them with a unique name. Every network tape corresponds to a single role $role_j$ and is used to let this role send messages to or receive messages from the adversary. Similarly, each role has a number of I/O tapes associated with it that are used when receiving or sending messages to other protocol machines. More specifically, for each role $role_j$ and each subroutine role $subrole_r$ specified in the **Subroutines** field there is an I/O tape that is used for direct communication between $role_j$ and $subrole_r$.[15] Conversely, those subroutine roles $subrole_r$ have a corresponding I/O tape. Note that $subrole_r$ might not specify $role_j$ as a subroutine itself. This means that each role, in addition to I/O tapes to subroutines, also has to provide I/O tapes for connections with arbitrary non-subroutine higher-level protocols. Thus every role $role_i$ is parameterized with a number of I/O tapes that other non-subroutine roles *role* can connect to. The specific number of tapes will be chosen appropriately depending on the number of other protocols that want to use $role_i$ as a subroutine and whether $role_i$ is public or private; see §F-C for details.

3) Message format: In order to uniquely determine both the sending entity and the intended receiving entity of a message, machines in our framework expect incoming messages m on some tape t to have a specific format (and likewise will encode all outgoing messages in this format). That is, a message that is received on an I/O tape must be of the form $m = ((pid_{snd}, sid_{snd}), (pid_{rcv}, sid_{rcv}), m')$ where m' is the message payload, pid_{snd} is the PID of the sender, sid_{snd} is the SID of the sender, pid_{rcv} is the PID of the intended receiver, and sid_{rcv} is the SID of the intended receiver. Now, the intended receiver of such a message is the entity $(pid_{rcv}, sid_{rcv}, role_{rcv})$ where $role_{rcv}$ is the role of M that the tape t belongs to. The sender $(pid_{snd}, sid_{snd}, role_{snd})$ is determined analogously, where $role_{snd}$ is also derived from the tape t and is either a string denoting the name of the subroutine role that this tape connects to, or a number $l \in \mathbb{N}$ denoting one of the I/O tapes connecting to arbitrary, non-subroutine higher level protocols.[16] Sending messages on the I/O interface is analogous; in particular, an instance is expected to prefix

[15] Except for the (uncommon) special case that $subrole_r$ is also a role of M, in which case there is no I/O tape as communication is handled internally.

[16] This implies in particular that machines do not learn the actual role name of the sending entities of non-subroutine higher level protocols, and thus cannot depend on this name in their code. Note that having some generic but consistent reference is usually sufficient, however, if so desired, protocol designers are free specify their algorithms such that they expect an additional identifier in message payloads.

messages with a correct header (if it does not, then the message will be dropped by the receiver in **CheckAddress** mode) and the tape is determined from both the sender role $role_{snd}$ (which is a role implemented by this machine) and the intended receiver role $role_{rcv}$ (which is either a subroutine role or a number $l \in \mathbb{N}$).

The expected format for messages m received and sent on network tapes is shortened to be $m = ((pid_{rcv}, sid_{rcv}), m')$ or $m = ((pid_{snd}, sid_{snd}), m')$, respectively, as one of the communication partners is always the adversary/simulator and thus does not need to be further determined. The role of the sending/receiving entity is uniquely defined by the network tape that is used for sending/receiving.

We note protocol designers using our syntax from Appendix A only have to deal with the actual message payload m' in their protocol specification; the syntax automatically takes care of adding the correct headers to the message payload and parsing headers of incoming messages.

4) Check address mode of protocol machines: As mentioned, every instance of a machine in our framework manages one or more entities of the form $(pid, sid, role)$ (where $role$ is one of the roles of M), and every entity is managed by a unique instance, i.e., there are no two instances that manage the same entity. On a high level, the **CheckAddress** mode is used to decide which instance manages which entity and route incoming messages accordingly.

First, recall that in the IITM model, whenever a message m is received on a tape of M, then all existing instances of M (in the order of their creation) are invoked in mode **CheckAddress** to check which instance accepts m. The first instance to accept m gets to process m in mode **Compute**. If no such instance exists, then a new one is created and run in mode **CheckAddress**. If this new instance accepts, it gets to process m, and otherwise, m is dropped and the new instance is removed from the run.

Now, upon being activated with a message m on tape t, an instance of M does the following in the **CheckAddress** mode, as specified in detail in Figure 14: the instance first checks that m contains a header that specifies the sender and intended receiver as described in the paragraph "message format". Note that the expected header format depends on whether the tape t is an I/O or network tape. If m does not contain a correct header, then `reject` is output; this ensures that instances accept a message and enter **Compute** only if they can determine both the sender and intended receiver of a message. Otherwise, the instance determines the intended receiving entity $(pid_{rcv}, sid_{rcv}, role_{rcv})$ and runs **CheckID**$(pid_{rcv}, sid_{rcv}, role_{rcv})$ to determine whether that entity is accepted or rejected. Thus, the protocol designer can freely define which receiving entities are accepted and thus managed by an instance of a machine.

Note that, since we require **CheckID** to produce consistent outputs (i.e., never output both `accept` and `reject` for the same entity during any run), the above definition of the **CheckAddress** mode implies that every receiving entity that has been accepted at some point will be accepted by the same instance again; in particular, no other instance gets to process messages for that entity during any point in the run. In other words, in every run for every (accepted) entity there is a uniquely determined instance that implements/manages that entity during mode **Compute**.

5) Compute mode of protocol machines: Recall that the **Compute** mode of an ITM specifies the actual computation performed by (an instance of) the ITM. Our framework fixes parts of the behavior of protocol machines in a specific way to guarantee the desired behavior in terms of corruption and addressing other machines. All other aspects can be customized by a protocol designer via specification of the various algorithms in the template from Figure 4.

We provide the formal specification of the **Compute** mode of protocol machines in Figures 15 and 16. On a high level, when an instance is activated with some message that includes the message body m from a sender $sender$ (either some entity connected via the I/O interface or the network) for some receiving entity $(pid_{rcv}, sid_{rcv}, role_{rcv})$, then the instance performs the following steps in order:

1. The instance processes requests on the I/O interface to obtain the current corruption status of $(pid_{rcv}, sid_{rcv}, role_{rcv})$. If such a request was processed, i.e., $m =$ `CorruptionStatus`?, then none of the following steps are performed but instead a response is returned directly to the sending entity $sender$.
2. If this is the first time that this instance reaches this step, then it runs **Initialization**.
3. If this is the first time that this instance reaches this step when receiving some message for the entity $(pid_{rcv}, sid_{rcv}, role_{rcv})$, then it runs **EntityInitialization**.
4. If this is the first time that this instance reaches this step when receiving some message for the entity $(pid_{rcv}, sid_{rcv}, role_{rcv})$, then it asks the adversary to determine the initial corruption status of that entity. This is done via a restricting message, i.e., the adversary is forced to respond such that the computation can continue from this point forward (except for negligible probability).
5. Corruption requests received from the network are processed.
6. If $(pid_{rcv}, sid_{rcv}, role_{rcv})$ is explicitly corrupted by the adversary, then messages are forwarded to/from the network.
7. Otherwise, if $(pid_{rcv}, sid_{rcv}, role_{rcv})$ is honest, then first **MessagePreprocessing** is run which might edit the message body m'. If **MessagePreprocessing** does not end the current activation, e.g., by sending a message, then **Main** is run afterwards on the modified message body m'.

Let us highlight and discuss a few aspects of the implementation in the following.

Upon receiving a message m on tape t in mode **CheckAddress**, do the following.

if t is an I/O tape:: {*Check that m contains the expected header, cf. §F-B3*

Check that $m = ((pid_{snd}, sid_{snd}), (pid_{rcv}, sid_{rcv}), m')$.

else:

Check that $m = ((pid_{rcv}, sid_{rcv}), m')$.

if the above check fails:

output `reject`.

Compute the receiving role $role_{rcv}$ from t. {*cf. §F-B3*

$decision \leftarrow$ **CheckID**$(pid_{rcv}, sid_{rcv}, role_{rcv})$.

Output *decision*.

Fig. 14: The **CheckAddress** mode of protocol machines in our framework.

User-defined algorithms: The **Compute** mode makes calls to all user-defined algorithms from the template given in Figure 4, except for **CheckID** which is used in the **CheckAddress** mode. We do not fix or restrict how these algorithms should be defined and we allow them full access not only to the internal state but also to all framework-specific variables such as $\mathsf{transcript}$, which is a log of all sent and received messages, and $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$, which stores the entity that has received the current message.[17] While this gives great flexibility for protocol designers, it also means that they must be careful in their definitions such that they do not accidently disrupt the intended protocol execution. In general, all algorithms should ensure that, when sending a message, that message has the expected format (cf. §F-B3) including a header that specifies sender and receiver. Note that this is automatically taken care of if our convenient syntax from §A is used, i.e., a protocol designer using this syntax only has to worry about the actual message payloads. The following messages payloads should be used with care as they are also used internally by our framework:

- `InitEntity`
- `InitEntityDone`
- `CorruptionStatus`?
- `CorruptionStatus`
- `CorruptMe`?
- `CorrStatusRestrict`
- `CorrMsgForward`

In principle, every algorithm can end the current activation either by sending a message or using the **abort** keyword. This has to be used with some care as it is quite easy to disrupt the intended protocol execution. In particular, a protocol generally should ensure that the initialization phase can run without interruption.

Core logic of (honest) entities: Recall that the algorithms **Initialization**, **EntityInitialization**, **MessagePreprocessing**, and **Main** define the core logic of a protocol machine. One important property of both initialization algorithms is that they are run before the initial corruption status of the current entity is determined and thus before the adversary can take control of the current entity. This means that initialization is performed even if the adversary intends to corrupt the current entity right at the start, allowing for performing a protected setup without interference of the adversary. In particular, one can first honestly create some internal state, and then later on decided based on this internal state whether the adversary may actually corrupt the current entity. In contrast, both **MessagePreprocessing** and **Main** are executed only for honest entities, so if an adversary corrupts an entity right at the beginning, they will never be run for that entity. Instead, all messages would always be forwarded to the adversary in this case.

Corruption handling: While the general corruption related behavior and the corresponding algorithms have already been explained in §III-B and §III-C, we now give more details about the technical aspects of corruption in this paragraph.

First, note that all corruption related behavior (Steps 1., 4., 5., and 6.) will be disabled and skipped entirely if **Corruption model** is set to `custom`. Thus, all framework-specific corruption related messages, such as `CorruptionStatus`?, are no longer handled automatically in such a case. Instead, a protocol designer is able to receive those messages in **MessagePreprocessing** and **Main** and manually specify how they are handled. For example, one can define an entirely different mechanism where corruption is not handled per entity but rather per machine instance, while `CorruptionStatus`? requests are still answered in an appropriate way to ensure interoperability with other protocols from our framework.

If **Corruption model** is not set to `custom`, then Step 1. handles `CorruptionStatus`? requests from other protocols/the environment. Observe that this is done at the very start of **Compute** mode and the current activation is ended directly after, without even performing any type of initialization. This ensures that `CorruptionStatus`? requests, by default and depending on the definition of the **DetermineCorrStatus** algorithm, are not visible to the network and do not affect the behavior of the protocol. As mentioned previously, this is because intuitively these requests are meta messages that are supposed to allow other protocols/the environment to obtain a snapshot of the current corruption state at any point in time, i.e., it should not matter for the protocol execution whether/when such a

[17] This access is kept implicit in Figures 15 and 16 as we do not want to clutter the calls to the algorithms with several additional parameters.

State variable initDone $\in \{\texttt{true}, \texttt{false}\} = \texttt{false}$. {*Has the instance been initialized?*

State variable entityInitDone $\subseteq (\{0,1\}^*)^3 = \emptyset$. {*Set of entities that have been initialized.*

State variable explicitCorr : $(\{0,1\}^*)^3 \to \{\texttt{true}, \texttt{false}, \bot\}$. {*Has an entity been explicitly corrupted? Initially* $\bot$.

State variable corrStatus : $(\{0,1\}^*)^3 \to \{\texttt{true}, \texttt{false}\}$. {*Consider entity to be corrupted? Initially* `false`.

State variable internalState. {*The internal state of the machine as defined in the template from Figure 4.*

State variable transcript. {*Log of all messages that were sent and received.*

State variable $\mathsf{entity}_{\mathsf{cur}} \in (\{0,1\}^* \cup \{\bot\})^3 = (\bot, \bot, \bot)$. {*Currently active entity (which was activated by receiving a message).*

State variable $\mathsf{entity}_{\mathsf{call}} \in (\{0,1\}^* \cup \{\bot\})^3 = (\bot, \bot, \bot)$. {*Sender of the last message that was received on the I/O interface.*

Upon receiving a message $m = ((pid_{snd}, sid_{snd}), (pid_{rcv}, sid_{rcv}), m')$ on an I/O tape t,
or a message $m = ((pid_{rcv}, sid_{rcv}), m')$ from a network tape t do:

Determine the receiver entity and store it in $\mathsf{entity}_{\mathsf{cur}} = (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$. If t is an I/O tape, also determine the sender entity
and store it in $\mathsf{entity}_{\mathsf{call}} = (\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})$. Let $\mathsf{t}_{\mathsf{net}}$ be the network tape corresponding to $\mathsf{role}_{\mathsf{cur}}$. {*cf. §F-B2 and §F-B3.*

if Corruption model $\neq \texttt{custom} \wedge m' = \texttt{CorruptionStatus}? \wedge t$ is an I/O tape: {*Step 1.: handle corruption status requests.*
  if $\mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}] = \bot$: {*Initial corruption status has not been determined yet.*
    send $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), (\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}), (\texttt{CorruptionStatus}, \texttt{false}))$ **on** t.
  else:
    $\mathsf{corrStatus}[\mathsf{entity}_{\mathsf{cur}}] \leftarrow \mathsf{corrStatus}[\mathsf{entity}_{\mathsf{cur}}] \vee \mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}]$.
    if $\mathsf{corrStatus}[\mathsf{entity}_{\mathsf{cur}}] = \texttt{false}$:
      $\mathsf{corrStatus}[\mathsf{entity}_{\mathsf{cur}}] \leftarrow$ **DetermineCorrStatus**$(\mathsf{entity}_{\mathsf{cur}})$.
    send $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), (\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}), (\texttt{CorruptionStatus}, \mathsf{corrStatus}[\mathsf{entity}_{\mathsf{cur}}]))$ **on** t.

if $\mathsf{initDone} = \texttt{false}$: {*Steps 2. and 3.: Initialization of internal state.*
  $\mathsf{initDone} \leftarrow \texttt{true}$.
  Run **Initialization**.
if $\mathsf{entity}_{\mathsf{cur}} \notin \mathsf{entityInitDone}$:
  Add $\mathsf{entity}_{\mathsf{cur}}$ to entityInitDone.
  Run **EntityInitialization**$(\mathsf{entity}_{\mathsf{cur}})$.

if Corruption model $\neq \texttt{custom}$:
  if $\mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}] = \bot$: {*Step 4.: Initialize corruption status of new entity.*
    if $t = \mathsf{t}_{\mathsf{net}} \wedge m' = (\texttt{SetCorruptionStatus}, b) \wedge b \in \{\texttt{true}, \texttt{false}\}$: {*First message already sets corruption status.*
      if $b = \texttt{true}$ and **Corruption model** allows corruption of the entity at this point:
        $\mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}] \leftarrow$ **AllowCorruption**$(\mathsf{entity}_{\mathsf{cur}})$. {*Use* **AllowCorruption** *to decide whether corruption succeeds.*
      else:
        $\mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}] \leftarrow \texttt{false}$.
      if $\mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}] = \texttt{true}$: {*Return a response to the adversary. Note that this stops the activation, none of the following steps are performed.*
        leakage $\leftarrow$ **LeakedData**().
        send $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), (\texttt{CorruptionStatus}, \texttt{true}, \mathit{leakage}))$ **on** $\mathsf{t}_{\mathsf{net}}$.
      else:
        send $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), (\texttt{CorruptionStatus}, \texttt{false}, \bot))$ **on** $\mathsf{t}_{\mathsf{net}}$.
    else: {*For all other first messages.*
      send responsively $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), \texttt{CorruptMe}?)$ **on** $\mathsf{t}_{\mathsf{net}}$;
      wait for $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), (\texttt{SetCorruptionStatus}, b))$ **on** $\mathsf{t}_{\mathsf{net}}$ **s.t.** $b \in \{\texttt{true}, \texttt{false}\}$.
      if $b = \texttt{true}$ and **Corruption model** allows corruption of the entity at this point:
        $\mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}] \leftarrow$ **AllowCorruption**$(\mathsf{entity}_{\mathsf{cur}})$. {*Use* **AllowCorruption** *to decide whether corruption succeeds.*
      else:
        $\mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}] \leftarrow \texttt{false}$.
      if $\mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}] = \texttt{true}$: {*Leak information and give control to either the adversary or, if this instance was triggered by an* `InitEntity` *command, to* $\mathsf{entity}_{\mathsf{call}}$.
        leakage $\leftarrow$ **LeakedData**().
        if t is an I/O tape and $m' = \texttt{InitEntity}$:
          send responsively $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), (\texttt{CorrStatusRestrict}, \texttt{true}, \mathit{leakage}))$ **on** $\mathsf{t}_{\mathsf{net}}$;
          wait for $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), \texttt{OK})$ **on** $\mathsf{t}_{\mathsf{net}}$.
          send $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), (\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}), \texttt{InitEntityDone})$ **on** t.
        else:
          send $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), (\texttt{CorruptionStatus}, \texttt{true}, \mathit{leakage}))$ **on** $\mathsf{t}_{\mathsf{net}}$.

  else if $\mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}] = \texttt{false}$: {*Step 5.: Process corruption requests for already existing entities.*
    if $t = \mathsf{t}_{\mathsf{net}} \wedge m' = (\texttt{SetCorruptionStatus}, b) \wedge b \in \{\texttt{true}, \texttt{false}\}$:
      if $b = \texttt{true}$ and **Corruption model** allows corruption of entities at this point: {*Corruption is allowed only if both the corruption model and* **AllowCorruption** *allow it.*
        $\mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}] \leftarrow$ **AllowCorruption**$(\mathsf{entity}_{\mathsf{cur}})$.
      else:
        $\mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}] \leftarrow \texttt{false}$.
      if $\mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}] = \texttt{true}$: {*Return a response to the adversary. Note that this stops the activation, none of the following steps are performed.*
        leakage $\leftarrow$ **LeakedData**().
        send $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), (\texttt{CorruptionStatus}, \texttt{true}, \mathit{leakage}))$ **on** $\mathsf{t}_{\mathsf{net}}$.
      else:
        send $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), (\texttt{CorruptionStatus}, \texttt{false}, \bot))$ **on** $\mathsf{t}_{\mathsf{net}}$.

Fig. 15: The **Compute** mode of protocol machines (part 1).

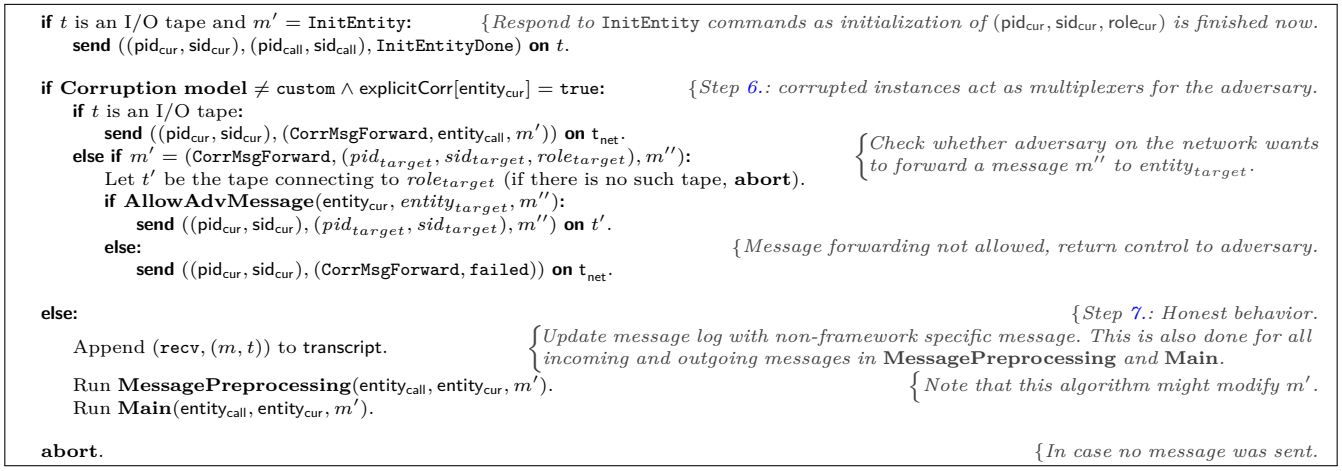


Fig. 16: The **Compute** mode of protocol machines (part 2).

snapshot was retrieved. In particular, if the behavior of the protocol were to depend on whether a `CorruptionStatus`? request was received previously, then this can create additional artificial attack vectors for the environment.

Except for `CorruptionStatus`? requests, all corruption related behavior is performed during Steps 4.-6., i.e., after (honest) initialization has been completed but before **MessagePreprocessing** and **Main**. The implementation mostly follows the behavior that has been described before and which we do not repeat here. We want to highlight some details though: firstly, observe that, when an entity asks for its initial corruption status in Step 4., it does so using a restricting message which the adversary must answer immediately. This feature ensures that the adversary must decide on the corruption status without interrupting the protocol execution by, e.g., first interacting with and changing the state of other instances. In particular, if the adversary decides not to corrupt the entity, then the protocol execution continues without interference. Secondly, there is a technical special case that needs to be handled differently, namely, receiving a `InitEntity` request from the I/O interface. This special message can be used by other protocols to initialize a subroutine entity, including its corruption status, but without losing control to the adversary in case the entity gets corrupted (cf. **init** macro defined in Appendix A). Thus, if the adversary corrupts an entity that initializes itself due to `InitEntity` request, he is notified via a restricting message (instead of a regular message) which includes the leakage of the entity. The adversary then has to return control to the corrupted entity immediately, which can then respond to the initial sender of the `InitEntity` request. Furthermore, if such a request is received, then the instance responds after Step 5. as then the entity has been initialized, i.e., Step 6. in particular is not executed. Overall, this yields the desired behavior.

6) Mapping the syntax from §A: Interpreting the send and receive commands presented in §A as commands in the sense of the IITM model is mostly straightforward. In particular, the tapes that are used and the headers that are added to the message payloads can be directly computed from the sending and receiving entities that are specified in the send and receive commands (as described in §F-B3). However, the **send responsively; wait for** and **send; wait for** commands need some additional explanation.

Recall that the **send responsively; wait for** command allows for sending messages to the network that will be answered immediately with a specific message. In particular, the adversary is not allowed to interact with other parts of the protocol or other entities before providing the expected response. Formally, this command sends the restricting message $(header, (\texttt{Respond}, m'))$ on a network tape, where $header$ is the message header as defined in §F-B3 and m' is the message payload as specified in the **send responsively** command. By definition of the restriction (cf. §F-A), the adversary has to respond to such a message with a message of the format $(header, m'')$, where $header$ is the same as in the restricting message and m'' is some arbitrary bit string. Because the headers are identical, the same instance of a machine will receive the response. That instance then continues where it left off, i.e., without repeating all previous steps and with all local variables still set to the same values, and tries to match m'' to the conditions and the message format imposed by the **wait for** part. If the match fails, it repeats the whole process by re-sending the original message $(header, (\texttt{Respond}, m'))$. Thus, an adversary must provide an expected answer if he wants the run to continue. We note that, formally, the adversary might not provide an immediate response (or a wrong response) with negligible probability, however, this negligible set of runs can be ignored in security proofs.

While the purpose of the **send; wait for** command is similar to the **send responsively; wait for** command, its implementation is actually more complex. Recall that this command is also supposed to allow a protocol designer to send some message and then wait for a response such that the run continues where it left off, including keeping all local variables. However, unlike the **send responsively; wait for** command, the **send; wait for** command can be

used for sending messages to both arbitrary protocols and the network; furthermore, it is not guaranteed that such a request will indeed be answered immediately. For example, if an instance I queries a subroutines via the **send; wait for** command, then this subroutine might be corrupted such that the request is forwarded to the adversary who can decide to, e.g., not answer but rather activate the instance I on a network tape or a different I/O tape again. Thus, we need to specify the behavior of I while it is waiting for a response but receives another message. In particular, it should still allow for certain meta actions, such as obtaining the current corruption status or getting corrupted, to be processed.

We use the following definition: If an instance I sends a messages via the **send; wait for** command, then it pushes the current values of all local variables and the current position in the program code on an internal stack. Now, if I receives some message m while this stack is non-empty, it proceeds as follows depending on the type of message:

- Check whether m is one of the following types of meta messages: `CorruptionStatus`?, `InitEntity` on the I/O interface or `SetCorruptionStatus`, `CorrMsgForward` on the network interface.
  If so, then proceed with the standard program logic while ignoring the state stored on the internal stack. In other words, these messages are processed separately even when waiting for a response to another request. Note that these messages might also include calls to a **send; wait for** command and thus push new states to the stack.
- For all other messages m, I takes the top most state stored on the stack and continue from the stored program position with the stored local variables. That is, the instance checks whether m matches all criteria of the **wait for** command based on the stored values of local variables and the current values of global variables (which might have changed since the state was stored on the stack!). If it does, then the computation continues where it left of; otherwise, the state is pushed back to the stack and the instance stops the current activation without output.

As should be obvious from this definition, it is quite easy to produce unintended behavior with the **send; wait for** command. We thus emphasize again that this construct must be used with special care and only sparely, e.g., to obtain a value from an incorruptible subroutine that responds immediately by its definition.

Besides the send and receive commands, we have also introduced two macros in §A: the **corr**$(pid, sid, role)$ and the **init**$(pid, sid, role)$ macros for retrieving the corruption status of and initializing an entity, respectively. Using the notation from §A, the **corr**$(pid, sid, role)$ macro is just a shorthand notation for **send** `CorruptionStatus`? **to** $(pid, sid, role)$; **wait for** $(\texttt{CorruptionStatus}, b)$ **s.t.** $b \in \{\texttt{true}, \texttt{false}\}$. Analogously, the **init**$(pid, sid, role)$ macro is just a shorthand notation for **send** `InitEntity` **to** $(pid, sid, role)$; **wait for** `InitEntityDone`. Both of these are fully defined by the above explanations.

C. Mapping Protocols

After we have explained how our template can be mapped to individual machines in the sense of the IITM model, we can explain how a complete protocol $\mathcal{P} = (role_1^{pub}, \ldots, role_n^{pub} \mid role_1^{priv}, \ldots, role_m^{priv})$ is mapped to a system of machines, where $\mathcal{P}$ might be built from several (sub-)protocols which are implemented by instance of the template each. More specifically, we mainly need to explain how the tapes of individual machines are connected and how public and private roles differ.

Let $\mathcal{P}$ be an arbitrary complete protocol as above. Let $M_1, \ldots, M_l$ be the machines of $\mathcal{P}$ that are specified using one or more templates and which implement one or more roles each. Now, the tapes corresponding to a role $role_i$ implemented by a machine M_j are connected as follows (in the context of $\mathcal{P}$):

- Recall that $role_i$ has an I/O tape t for each subroutine role $role_{sub}$ of M_j. Since $\mathcal{P}$ is complete, we have that $role_{sub}$ is implemented by one of the machines of $\mathcal{P}$. If $role_{sub}$ also uses $role_i$ as a subroutine, then t is connected to the corresponding subroutine I/O tape of $role_{sub}$. Otherwise, the tape t is connected to one of the parameterized many I/O tapes of $role_{sub}$.
- Recall that $role_i$ also has parameterized many I/O tapes for arbitrary higher level protocols to connect to.
  - If $role_i$ is a private role in $\mathcal{P}$, then the parameter is chosen such that there are exactly as many tapes as are needed for roles of $\mathcal{P}$ that want to use $role_i$ as a subroutine. In other words, every role of $\mathcal{P}$ can connect to $role_i$, however, there are no additional (unconnected) tapes that would allow for other protocols or the environment to connect to $role_i$.
  - If $role_i$ is a public role in $\mathcal{P}$, then the parameter is still arbitrary but sufficiently large such that all roles of $\mathcal{P}$ using $role_i$ as a subroutine can connect to it. In other words, not only can all roles of $\mathcal{P}$ connect to $role_i$, but there are also arbitrarily many unconnected tapes that can be used by other protocols or the environment to connect to $role_i$.
- The single network tape of $role_i$ is left unconnected such that the adversary/simulator in the definition of the realization relation can connect to it.

The protocol $\mathcal{P}$ is then implemented by the system of machines $\{M_1, \ldots, M_l\}$ that is connected as described above.

Appendix G
Proof of the Unbounded Self-Composition Theorem

In this section we provide a proof for the unbounded self-composition theorem in iUC (cf. Corollary 9 in Appendix C).

Let σ be an PSID function as defined for iUC in Definition 2. Suppose we have two (iUC) protocols $\mathcal{P}$ and $\mathcal{F}$ that are σ-session protocols such that $\mathcal{P} \leq_{\sigma\text{-single}} \mathcal{F}$. In the following, we want to use the unbounded self-composition theorem of the IITM model (cf. Theorem 10 in Appendix E) to conclude that $\mathcal{P} \leq \mathcal{F}$. For this purpose, we have to define a PSID function $\widetilde{\sigma}$ in the sense of the IITM model (cf. Appendix E), show that $\mathcal{P}$ and $\mathcal{F}$ are $\widetilde{\sigma}$-session protocols in the sense of the IITM model (cf. Appendix E), and show that $\mathcal{P} \leq^{\sim}_{\sigma\text{-single}} \mathcal{F}$ (again, in the sense of the IITM model). An important difficulty here is that a "σ-session protocol" is a property defined for runs of the whole protocol (in some arbitrary responsive environment), whereas "$\widetilde{\sigma}$-session protocol" is a property defined for runs of individual machines of the protocol (in some arbitrary context that might not even be responsive or runtime bounded). Thus, formally, some properties that hold true in the context of the whole protocol, might no longer be true if the protocol is broken apart. Dealing with this issue is the main obstacle of this proof.

Let us begin by summarizing the major differences between σ and $\widetilde{\sigma}$ as well as σ-session protocols and $\widetilde{\sigma}$-session protocols in iUC and the IITM model, respectively. Firstly, σ is defined on entities, whereas $\widetilde{\sigma}$ takes as input a message and a named tape and then determines the PSID of the machine instance that sent/received this message. Thus, we have to define $\widetilde{\sigma}$ such that it uses the header information contained in messages in iUC (cf. Appendix F-B3) and the tape to determine the entity that receives some message and then evaluate σ for that entity. Note that importantly, for internal tapes connecting two roles of $\mathcal{P}$, the output of $\widetilde{\sigma}$ is not only required to match the PSID of the receiving entity but it must also match the PSID of the sending entity (as otherwise the sender would send a message to another session, i.e., $\mathcal{P}$ would not be a $\widetilde{\sigma}$-session protocol).

We define $\widetilde{\sigma}(m,t)$ as follows:

- If t is an external output tape of $\mathcal{P}/\mathcal{F}$ (i.e., connecting to the environment or adversary), then compute the sending entity $(pid_{snd}, sid_{snd}, role_{snd})$. That is, parse m to obtain $pid[snd]$ and $sid[snd]$ of the receiving entity and compute $role[snd]$ (if parsing m does not work, e.g., due to an invalid header format, set $\widetilde{\sigma}(m,t) := \bot$). Output the PSID of the receiving entity, i.e., $\widetilde{\sigma} := \sigma(entity_{snd})$.
- If t is an external input tape of $\mathcal{P}/\mathcal{F}$ (i.e., connecting from the environment or adversary) or an internal tape of $\mathcal{P}/\mathcal{F}$ (i.e., connecting two machines of $\mathcal{P}/\mathcal{F}$), then compute the receiving entity $(pid_{rcv}, sid_{rcv}, role_{rcv})$. That is, parse m to obtain $pid[rcv]$ and $sid[rcv]$ of the receiving entity and compute $role[rcv]$ (if parsing m does not work, e.g., due to an invalid header format, set $\widetilde{\sigma}(m,t) := \bot$). Output the PSID of the receiving entity, i.e., $\widetilde{\sigma} := \sigma(entity_{rcv})$.
- For tapes t that are not part of $\mathcal{P}/\mathcal{F}$, set $\widetilde{\sigma}(m,t) := \bot$.

Observe that $\widetilde{\sigma}$ is indeed a session function as it is efficiently computable. The protocols $\mathcal{P}$ and $\mathcal{F}$ are *almost* $\widetilde{\sigma}$-session protocols for the above definition. Consider the behavior of individual machines in runs of the whole protocol with a responsive environment. Firstly, observe that no machine in $\mathcal{P}/\mathcal{F}$ accepts messages where $\widetilde{\sigma}$ is $\bot$ as then either the header is malformed or σ is also $\bot$. Secondly, if an instance has accepted a messages with some PSID according to $\widetilde{\sigma}$, then it will not accept messages for any other PSIDs, as this would imply that it accepts two entities with different PSIDs according to σ. Thirdly, messages sent by an instance have the same PSID according to $\widetilde{\sigma}$ as those that were previously accepted. For external tapes, this directly follows from the fact that the sending entity must have the correct PSID according to σ. For internal tapes, this is implied by the additional requirement that the receiving entity also has the correct PSID according to σ.

So overall, the properties of machines of $\widetilde{\sigma}$-session protocols hold true but only for the specific context of the whole protocol running with a responsive environment. However, we need those properties to also hold true when running individual machines in an arbitrary context, which is not the case in general. Thus we have to modify $\mathcal{P}$ and $\mathcal{F}$ slightly. The new protocols $\widetilde{\mathcal{P}}$ and $\widetilde{\mathcal{F}}$ are the same as before, except for the following changes made to each machine: Before accepting a message in mode **CheckAddress**, the machine first checks that the properties of $\widetilde{\sigma}$ session protocols are not violate and rejects the message otherwise. Furthermore, before sending a message, the machine again checks that the properties of $\widetilde{\sigma}$-session protocols are fulfilled and aborts the activation without sending the message otherwise. Thus, we have that $\widetilde{\mathcal{P}}$ and $\widetilde{\mathcal{F}}$ are $\widetilde{\sigma}$-session protocols and, by the above observations, they behave identical to $\mathcal{P}$ and $\mathcal{F}$ when running the whole protocol in a responsive environment. Note that both $\widetilde{\mathcal{P}}$ and $\widetilde{\mathcal{F}}$ are still R-environmentally-bounded as in particular $\widetilde{\sigma}$ is efficiently computable.

Now let $\mathcal{E} \in \mathsf{Env}_{R,\widetilde{\sigma}\text{-single}}(\mathcal{P})$ be a single-session environment according to $\widetilde{\sigma}$. We have that $\mathcal{E}$ sends only messages m on tape t such that $\widetilde{\sigma}(m,t)$ always outputs the same PSID, say, $psid$. By definition of $\widetilde{\sigma}$, those messages are always sent to entities who have PSID $psid$ according to σ. Thus we have $\mathcal{E} \in \mathsf{Env}_R(\mathcal{P})$. As $\mathcal{P} \leq_{\sigma\text{-single}} \mathcal{F}$ by assumption, we have that there exists a simulator $\mathcal{S}_{single}$ such that $\{\mathcal{E}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}_{single}, \mathcal{F}\}$. As $\mathcal{P}$ and $\widetilde{\mathcal{P}}$ as well as $\mathcal{F}$ and $\widetilde{\mathcal{F}}$ behave identical in runs with arbitrary responsive environments, we have that $\{\mathcal{E}, \widetilde{\mathcal{P}}\} \equiv \{\mathcal{E}, \mathcal{S}_{single}, \widetilde{\mathcal{F}}\}$. In summary, this implies $\widetilde{\mathcal{P}} \leq^{\sim}_{\sigma\text{-single}} \widetilde{\mathcal{F}}$.

We can now apply the unbounded self-composition theorem of the IITM model (cf. Theorem 10) to conclude that $\widetilde{\mathcal{P}} \leq \widetilde{\mathcal{F}}$. By the same argument as above, using that $\widetilde{\mathcal{P}}$ and $\mathcal{P}$ as well as $\widetilde{\mathcal{F}}$ and $\mathcal{F}$ behave identical in the context of arbitrary responsive environments, this implies $\mathcal{P} \leq \mathcal{F}$. □