iUC: Flexible Universal Composability Made Simple*

Jan Camenisch[1], Stephan Krenn[2],
Ralf Küsters[3], and Daniel Rausch[3]

[1] Dfinity, Zurich, Switzerland
jan@dfinity.org
[2] AIT Austrian Institute of Technology GmbH, Vienna, Austria
stephan.krenn@ait.ac.at
[3] University of Stuttgart, Stuttgart, Germany
{ralf.kuesters,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. A perfect model for universal composability should provide a sound basis for formal proofs and be very flexible in order to allow for modeling a multitude of different protocols. It should also be easy to use, including useful design conventions for repetitive modeling aspects, such as corruption, parties, sessions, and subroutine relationships, such that protocol designers can focus on the core logic of their protocols.
While many models for universal composability exist, including the UC, GNUC, and IITM models, none of them has achieved this ideal goal yet. As a result, protocols cannot be modeled faithfully and/or using these models is a burden rather than a help, often even leading to underspecified protocols and formally incorrect proofs.
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 in an unmatched way. Developing such a security framework is a very difficult and delicate task, as the long history of frameworks for universal composability shows.
We build our framework, called iUC, on top of the IITM model, which already provides soundness and flexibility while lacking sufficient usability. At the core of iUC is a single simple template for specifying essentially arbitrary protocols in a convenient, formally precise, and flexible way. We illustrate the main features of our framework with example functionalities and realizations.

Keywords: Universal Composability, Foundations

* This work 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. We would like to thank Robert Enderlein for helpful discussions.

Table of Contents

1 Introduction

Universal composability [4, 25] 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 [16], network time protocols [11], OAuth v2.0 [14], the integrity of file systems [8], as well as privacy in email ecosystems [13].

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

Several models for universal composability have been proposed in the literature [4, 5, 7, 9, 10, 15, 18, 23–25]. 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.

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: It should be easy to precisely and fully formalize protocols; this is also an important prerequisite for carrying out formally/mathematically correct proofs. There should exist (easy to use) modeling conventions that allow a protocol designer to focus on the core logic of protocols 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, existing models and frameworks are still unsatisfying in these respects as none combines all of these requirements simultaneously (we discuss this in more detail below). Thus, 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. Developing such a security framework is a difficult and very delicate task that takes multiple years if not decades as the history on models for universal composability shows. Indeed, this paper is the result of many years of iterations, refinements, and discussions.

Contributions: To achieve the above described goal, we here propose a new universal composability framework called iUC ("IITM based Universal Composability"). This framework builds on top of the IITM model with its extension to so-called responsive environments [1]. The IITM model was originally proposed in [18], with a full and revised version – containing a simpler and more general runtime notion – presented in [22].

The IITM model already meets our goals of *soundness* and *flexibility*. That is, the IITM model offers a very general and at the same time simple runtime notion so that protocol designers do not have to care much about runtime issues, making sound proofs easier to carry out. Also, protocols are defined in a very general way, i.e., they are essentially just arbitrary sets of Interactive Turing Machines (ITMs), which may be connected in some way. In addition, the model offers a general addressing mechanism for machine instances. This gives great flexibility as arbitrary protocols can be specified; all theorems, such as composition theorems, are proven for this very general class of protocols. Unfortunately, this generality hampers *usability*. The model does not provide design conventions, for example, to deal with party IDs, sessions, subroutine relationships, shared state, or (different forms of) corruption; all of this is left to the protocol designer to manually specify for every design and analysis task, distracting from modeling the actual core logic of a protocol.

In essence, iUC is an instantiation of the IITM model that provides a convenient and powerful framework for specifying protocols. In particular, iUC greatly improves upon *usability* of the IITM model by adding missing conventions for many of the above mentioned repetitive aspects of modeling a protocol, while also

abstracting from some of the (few) technical aspects of the underlying model (in particular the concept of named tapes, explained in §2); see below for the comparison of iUC with other frameworks.

At the core of iUC is *one* convenient template that supports protocol designers in specifying arbitrary types of protocols 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. The template comes with a clear and intuitive syntax which further facilitates specifications and allows others to quickly pick up protocol specifications and use them as subroutines in their higher-level protocols.

A key difficulty in designing iUC was to preserve the *flexibility* of the original IITM model in expressing (and composing) arbitrary protocols while still improving *usability* by fixing modeling conventions for certain repetitive aspects. We solve this tension between flexibility and usability by, on the one hand, allowing for a high degree of customization and, on the other hand, by providing sensible defaults for repetitive and standard specifications. Indeed, as further explained and discussed in §4 and also illustrated by our case study (cf. §5 and Appendix E), iUC preserves flexibility and supports a wide range of protocol types, protocol features, and composition operations, such as: ideal and global functionalities with arbitrary protocol structures, i.e., rather than being just monolithic machines, they may, for example, contain subroutines; protocols with joint-state and/or global state; shared state between multiple protocol sessions (without resorting to joint-state realizations); subroutines that are partially globally available while other parts are only locally available; realizing global functionalities with other protocols (including joint-state realizations that combine multiple global functionalities); different types of addressing mechanisms via globally unique and/or locally chosen session IDs; global functionalities that can be changed to be local when used as a subroutine; many different highly customizable corruption types (including incorruptability, static corruption, dynamic corruption, corruption only under certain conditions, automatic corruption upon subroutine corruptions); a corruption model that is fully compatible with joint-state realizations; arbitrary protocol structures that are not necessarily hierarchical trees and which allow for, e.g., multiple highest-level protocols that are accessible to the environment.

Importantly, all of the above is supported by just *a single template* and *two* composition theorems (one for parallel composition of multiple protocols and one for unbounded self composition of the same protocol). This makes iUC quite user friendly as protocol designers can leverage the full flexibility with just the basic framework; there are no extensions or special cases required to support a wide range of protocol types.

We emphasize that we do not claim specifications done in iUC to be shorter than the informal descriptions commonly found in the universal composability literature. A full, non-ambiguous specification cannot compete with such informal descriptions in terms of brevity, as these descriptions are often underspecified and ignore details, including model specific details and the precise corruption behavior. iUC is rather meant as a *powerful and sound tool for protocol designers that desire to specify protocols fully, without sweeping or having to sweep anything under the rug, and at the same time without being overburdened with modeling details and technical artifacts.* Such specifications are crucial for being able to understand, reuse, and compose results and to carry out sound proofs.

Related work: The currently most relevant universal composability models are the UC model [4] (see [3] for the latest version), the GNUC model [15], the IITM model [18] (see [22] for the full and revised version), and the CC model [23]. The former three models are closely related in that they are based on polynomial runtime machines that can be instantiated during a run. In contrast, the CC model follows a more abstract approach that does not fix a machine model or runtime notion, and is thus not directly comparable to the other models (including iUC). Indeed, it is still an open research question if and how typical UC-style specifications, proofs, and arguments can be modeled in the CC model. In what follows, we therefore relate iUC with the UC and GNUC models; as already explained and further detailed in the rest of the paper, iUC is an instantiation of the IITM model.

While both the UC and GNUC models also enjoy the benefits of established protocol modeling conventions, those are, however, less flexible and less expressive than iUC. Let us give several concrete examples: conventions in UC and GNUC are built around the assumption of having globally unique SIDs that are shared between all participants of a protocol session, and thus locally managed SIDs cannot directly be expressed (cf. §4, §5, and §5.3 for details including a discussion of local SIDs). Both models also assume protocols to have disjoint

sessions and thus their conventions do not support expressing protocols that directly share state between sessions, such as signature keys (while both models support joint-state realizations to somewhat remedy this drawback, those realizations have to modify the protocols at hand, which is not always desirable; cf. §5.3). Furthermore, in both models there is only a single highest-level protocol machine with potentially multiple instances, whereas iUC supports arbitrarily many highest-level protocol machines. This is very useful as it, for example, allows for seamlessly modeling global state without needing any extensions or modifications to our framework or protocol template (as illustrated in §5). In the case of GNUC, there are also several additional restrictions imposed on protocols, such as a hierarchical tree structure where all subroutines have a single uniquely defined caller (unless they are globally available also to the environment) and a fixed top-down corruption mechanism; none of which is required in iUC.

There are also some major differences between UC/GNUC and iUC on a technical level which further affect overall usability as well as expressiveness. Firstly, both UC and GNUC had to introduce various extensions of the basic computational model to support new types of protocols and composition, including new syntax and new composition theorems for joint-state, global state, and realizations of global functionalities [5, 7, 12, 15]. This not only forces protocol designers to learn new protocol syntax and conventions for different types of composition, but also indicates a lack of flexibility in supporting new types of composition (say, for example, a joint-state realization that combines several separate global functionalities, cf. §5.3). In contrast, both composition theorems in iUC as well as our single template for protocols seamlessly support all of those types of protocols and composition, including some not considered in the literature so far (cf. Appendix E.3). Secondly, there are several technical aspects in the UC model a protocol designer has to take care of in order to perform sound proofs: a runtime notion that allows for exhaustion of machines, even ideal functionalities, and that forces protocols to manually send runtime tokens between individual machine instances; a directory machine where protocols have to register all instances when they are created; "subroutine respecting" protocols that keep sessions disjoint. Technical requirements of the GNUC model mainly consist of several restrictions imposed on protocol structures (as mentioned above) which in particular keep protocol sessions disjoint. Unlike UC, the runtime notion of GNUC supports modeling protocols that cannot be exhausted, however, GNUC introduces additional flow-bounds to limit the number of bits sent between certain machines. In contrast, as also illustrated by our case study, iUC does not require directory machines, iUC's notion for protocols with disjoint sessions is completely optional and can be avoided entirely, and iUC's runtime notion allows for modeling protocols without exhaustion, without manual runtime transfers, and without requiring flow bounds (exhaustion and runtime transfers can of course be modeled as special cases, if desired).

The difference in flexibility and expressiveness of iUC compared to UC and GNUC is further explained in §4 and illustrated by our case study in §5, where we model a real world key exchange protocol exactly as it would be deployed in practice. This case study is not directly supported by the UC and GNUC models (as further discussed in §5.3). A second illustrative example is given in Appendix E.9, where we show that iUC can capture the SUC model [10] as a mere special case. The SUC model was proposed as a simpler version of the UC model specifically designed for secure multi party computation (MPC), but has to break out of (some technical aspects of) the UC model.

Structure of this paper: In §2, we briefly recall the IITM model along with the extension to responsive environments as far as relevant for iUC. We describe the iUC framework in §3, with a discussion of the main concepts and features in §4. A case study further illustrates and highlights some features of iUC in §5. We conclude in §6. Further details are provided in the appendix.

This paper is the full version of [2].

2 Relevant Parts of the IITM Model

The IITM model was first introduced in [18] and revised in [22] with a more general and simpler runtime notion. In [1], the IITM model was extended to handle *responsive environments*, a general concept (see below) that can also be applied to other models. 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 H.

Inexhaustible interactive Turing machines: An inexhaustible interactive Turing machine (IITM or simply ITM) is a probabilistic Turing machine with a number of named 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 $\mathcal{Q} = \{M_1, \ldots, M_k\}$[4] 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 name n, it is required that at most two of these ITMs have a tape named n; we say that both ITMs respectively their tapes with name n are connected. Tapes in $\mathcal{Q}$ that are already connected to (a tape of) another machine of $\mathcal{Q}$ 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 for secure direct communication with other protocols[5] (or the environment, see below) 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, where the external tapes of $\mathcal{Q}_1$ and $\mathcal{Q}_2$ that are now connected are internal tapes of the system $\{\mathcal{Q}_1, \mathcal{Q}_2\}$. 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;[6] if no master ITM exists, the run of $\mathcal{Q}$ terminates immediately. If a message m is written by some instance of a machine, say M', on one of its named tapes, say on a tape named n, and there is a different machine, say M, in $\mathcal{Q}$ with another tape also named n, 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 with name n. 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 with name n 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

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

[5] I/O tapes are generally used to model a subroutine relationship between two protocols, representing that a higher-level protocol is able to directly send inputs and receive outputs from its subroutines without interference of a network attacker.

[6] 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.

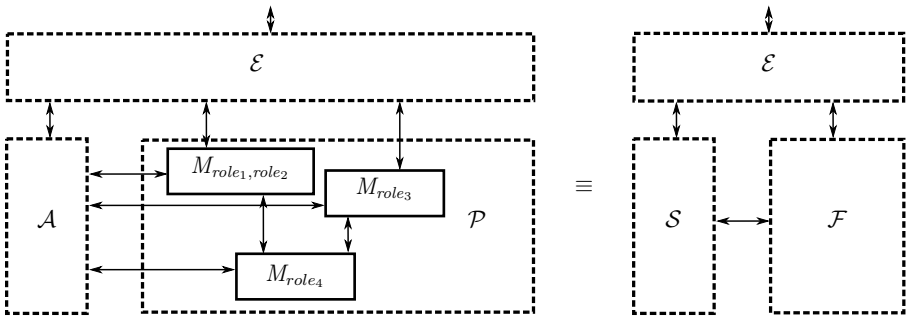


Fig. 1: The setup for the universal composability experiment ($\mathcal{P} \leq \mathcal{F}$) and internal structure of protocols. Here $\mathcal{E}$ is an environmental system, $\mathcal{A}$ and $\mathcal{S}$ are adversarial systems, and $\mathcal{P}$ and $\mathcal{F}$ are protocol systems. Arrows between $\mathcal{E}$ 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 $\mathcal{P}$ with the machines depicted is just an example. Also, all other systems, including $\mathcal{E}$, $\mathcal{A}$, and $\mathcal{F}$, can also consist of several machines connected in some way.

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 named tapes, say a tape named n', and then stops. If there is another ITM with a tape also named n' in the system, the message m' is delivered to one instance of that ITM on the tape with name n' as described above. If the instance of M stops without outputting a message or there is no other ITM with a tape named n', then (an instance of) the master ITM is activated in the same way as described above. 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.

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 an example of how $\mathcal{P}$ 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 detail in [1], it is often natural to expect the adversary to send an immediate response to such requests, as otherwise one has to deal with additional and often hard to resolve artificial complications in protocol specifications and security proofs. For this reason, [1] introduces

the concept of responsive environments and proposes an extension of the IITM model (as well as the UC and GNUC models). 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 H. This concept is also used and illustrated, for example, in §3.2 and in the case study (see, e.g., Figure 7). See also Appendix E.8 for more discussion of this concept.

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 (or environment) $\mathcal{E}$ 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 $\mathcal{Q}$, $\mathsf{Env}(\mathcal{Q})$ 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 (or protocol) $\mathcal{P}$ has to be *environmentally bounded*, i.e., for every $\mathcal{E} \in \mathsf{Env}(\mathcal{P})$ 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 (or adversary) $\mathcal{A}$ for a protocol system $\mathcal{P}$ must satisfy that the combined system $\{\mathcal{A}, \mathcal{P}\}$ is environmentally bounded. We define the set $\mathsf{Adv}(\mathcal{P})$ to contain all such adversarial systems for $\mathcal{P}$ where, in addition, we require that these adversaries 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. These runtime notions are introduced, discussed, and compared with notions of other frameworks in detail in [22], with the treatment of responsiveness presented in [1]. In particular, as discussed in [22], we point out that the environmentally bounded property is typically easy to check and should cover all practical protocols. In fact, we are not aware of any real-world protocol that is not environmentally bounded.

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 $\mathcal{A} \in \mathsf{Adv}(\mathcal{P})$ there exists $\mathcal{S} \in \mathsf{Adv}(\mathcal{F})$ such that for all $\mathcal{E} \in \mathsf{Env}(\{\mathcal{A}, \mathcal{P}\})$ 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.

The above simulation notion is often called "universal simulatability" or "universal composability". 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 "universal simulatability" [1, 22], which is an important sanity check for a UC-like model. In particular, *strong simulatability* is a conceptually simpler notion that allows for omitting the adversary $\mathcal{A}$ and which 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 $\mathcal{S} \in \mathsf{Adv}(\mathcal{F})$ such that for all $\mathcal{E} \in \mathsf{Env}(\mathcal{P})$ 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. *[1] 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 environmentally bounded and $\mathcal{Q}$ connects only to external I/O tapes of $\mathcal{P}/\mathcal{F}$, 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 [20, 22].

The second 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 [1]. We state this theorem more formally in Appendix H. 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 $\mathcal{F}$.

3 The iUC Framework

In this section, we present the iUC framework which is built on top of the IITM model. As explained in §1, the main shortcoming of the IITM model is a lack of usability due to missing conventions for protocol specifications. Thus, protocol designers have to manually define many repetitive modeling related aspects such as a corruption model, connections between machines, specifying the desired machine instances (e.g., does an instance model a single party, a protocol session consisting of multiple parties, a globally available resource), the application specific addressing of individual instances, etc. The iUC framework solves this shortcoming by adding convenient and powerful conventions for protocol specifications to the IITM model. A key difficulty in crafting these conventions is preserving the flexibility of the original IITM model in terms of expressing a multitude of various protocols in natural ways, while at the same time not overburdening a protocol designer with too many details. We solve this tension by providing *a single template* for specifying arbitrary types of protocols, including real, ideal, joint-state, global state protocols, which needed several sets of conventions and syntax in other frameworks, and sometimes even new theorems. Our template includes many optional parts with sensible defaults such that a protocol designer has to define only those parts relevant to her specific protocol. As the iUC framework is an instantiation of the IITM model, all composition theorems and properties of the IITM model carry over.

We start by explaining the general structure of protocols in iUC in §3.1, with corruption explained in §3.2. We then present our protocol template in §3.3. In §3.4, we explain how protocol specifications can be composed in iUC to create new, more complex protocol specification. Finally, in §3.5, we present the realization relation and the composition theorem of iUC. As mentioned, concrete examples are given in our case study (cf. §5). We provide a precise mapping from iUC protocols to the underlying IITM model in Appendix I, which is crucial to verify that our framework indeed is an instantiation of the IITM model, and hence, 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.

3.1 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.

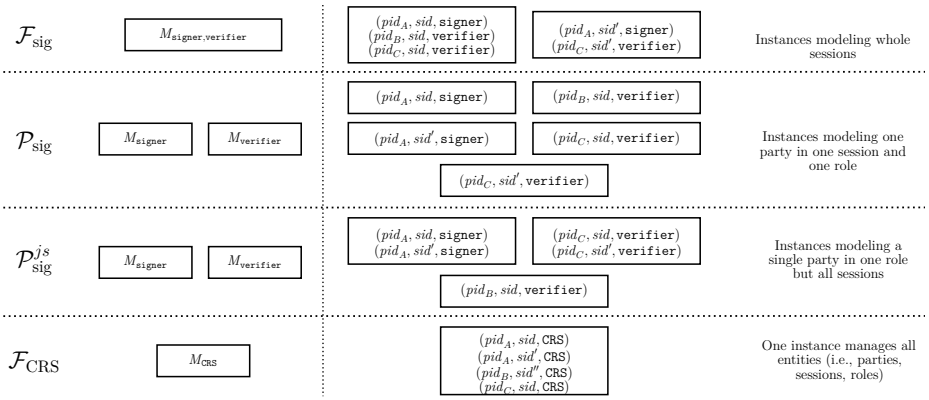
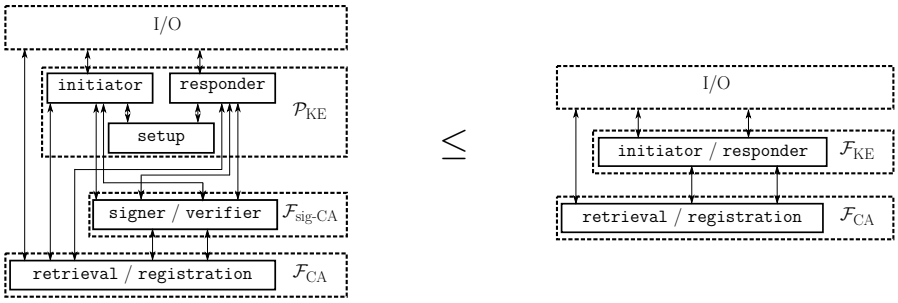


Fig. 2: Examples of static and dynamic structures of various protocol types. $\mathcal{F}_{\text{sig}}$ is an ideal protocol, $\mathcal{P}_{\text{sig}}$ a real protocol, $\mathcal{P}^{js}_{\text{sig}}$ a so-called joint-state realization, and $\mathcal{F}_{\text{CRS}}$ a global state protocol. On the 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).

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-hand side of Figure 2). Machines provide an I/O interface and a network interface for every role that they implement. The I/O interfaces of two roles of two different machines can be connected. This means that, in a run of a system, two entities (managed by two instances of machines) with connected roles can then directly send and receive messages to/from each other; in contrast, entities of unconnected roles cannot directly send and receive messages to/from each other.[7] Jumping ahead, in a protocol specification (see below) it is specified for each machine in that protocol to which other roles (subroutines) a machine connects to (see, e.g., also Figure 3a where the arrows denote connected roles/machines). The network interface of every role is connected to the adversary (or simulator), allowing for sending and receiving messages to and from the adversary. For addressing purposes, we assume that each role in $\mathcal{P}$ has a unique name. Thus, role names can be used for communicating with a specific piece of code, i.e., sending and receiving a message to/from the correct machine.

Public and private roles: We, in addition, introduce the concept of public and private roles, which, as we will explain, 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 (some of the) 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 its 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 an example below.

[7] This bidirectional connection of interfaces is an abstraction from tapes and tape names in the IITM model. Protocol designers need not care about those in iUC. More information about how connections are mapped to tapes in the sense of the IITM model is available in Appendix I.2.



(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 connections of I/O interfaces; network connections are omitted for simplicity. Solid boxes (labeled with one or two role names) denote individual machines, dotted boxes denote (sub-)protocols that are specified by one instance of our template each (cf. §3.3).

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 (the I/O interfaces of) their public roles. (We give a formal definition of connectable protocols in §3.4.) 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 name of their original protocol. We will often also explicitly write down this prefix in the protocol specification for better readability (cf. §3.3).

Examples illustrating the above concepts: Figure 3a, which is further explained in our case study (cf. §5), 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 that this does not affect the security statement: the environment knows the public verification algorithm and can obtain all verification keys from $\mathcal{F}_{\text{CA}}$, i.e., the environment can locally compute the results of the verification algorithm. 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 "|". 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 [7, 20] 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 the case of the network interface or some other entity (with a role that has a connected I/O interface) in the 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 $role_{rcv}$; a special user-defined **CheckID** algorithm (see §3.3) is then used to determine the instance of M that manages $(pid_{rcv}, sid_{rcv}, role_{rcv})$ and should hence receive the message. 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}[8] and either the actual role name $role_{snd}$ (if the sender is a known subroutine of the receiver, cf. §3.3) or an arbitrary but fixed number i (from an arbitrary but fixed range of natural numbers) denoting a specific I/O connection to some (unknown) sender role (if the sender is an unknown higher-level protocol or the environment[9]). 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 in a consistent way 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 via their I/O interfaces, 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 study. It, for example, also allows us to model the global functionality $\mathcal{F}_{\text{CRS}}$ from

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

[9] The environment can choose the number that it claims as a sender as long as it does not collide with a number used by another (higher-level) role in the protocol.

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.

3.2 Modeling Corruption

We now explain on an abstract level how our framework models corruption of entities. In §3.3, 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 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 an algorithm **AllowCorruption**, which can be used 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 (whether or not subroutines are corrupted can be determined using `CorruptionStatus`? requests, see later), 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 (but yields a less fine grained security result). One could also prevent corruption during a protected/trusted "setup" phase of the protocol, and allow corruption only afterwards.

If a `corrupt` request for some entity $(pid, sid, role)$ passes all checks and is accepted, then the state of the entity is leaked to the adversary (which can be customized by specifying an algorithm **LeakedData**) and 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 by specifying an algorithm **AllowAdvMessage**. This can be used, e.g., to prevent the adversary from accessing uncorrupted instances or from communicating with other (disjoint) sessions, as detailed in §3.3.

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 (see §3.3), it asks the adversary via a *restricting message*[10] 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 $(pid, sid, role)$.

[10] Recall from §2 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. This is a very typical use of restricting messages, which very much simplifies corruption modeling (see also [1]).

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. Note that the template does not specify public and private roles as those change depending on how several protocols (each defined via a copy of this template) are connected.

For modeling purposes, we allow other entities and the environment to obtain the current corruption status of an entity $(pid, sid, role)$.[11] 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 (this is specified by the protocol designer via an algorithm **DetermineCorrStatus**). 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.

3.3 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.

[11] This operation is purely for modeling purposes and does of course not exist in reality. It is crucial for obtaining a reasonable realization relation: The environment needs a way to check that the simulator in the ideal world corrupts exactly those entities that are corrupted in the real world, i.e., the simulation should be perfect also with respect to the corruption states. If we did not provide such a mechanism, the simulator could simply corrupt all entities in the ideal world which generally allows for a trivial simulation of arbitrary protocols.

Participating roles*:* This list of sets of roles specifies which roles are (jointly) implemented by a machine. To give an 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 §3.2: *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 (all roles of) the machine M_i will then be connected to the I/O interfaces of those roles, allowing M_i to access and send messages to those subroutines.[12] We note that (subroutine) roles are uniquely specified by their name since we assume globally unique names for each role. We also note that subroutines are specified on the level of roles, instead of the level of whole protocols, as this yields more flexibility and a more fine grained subroutine relationship, and hence, access structure.

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 I.2).

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., $sid_i = (parameter_1, parameter_2, sid'_i)$.

More precisely, the algorithm **CheckID**$(pid, sid, role)$ is a *deterministic algorithm* that computes on the input $(pid, sid, role)$, the internal state of the machine instance, and the security parameter. It runs in *polynomial time* in the length of the current input, the internal state, and the security parameter and 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. For this purpose, we provide access to a convenient framework specific

[12] 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.

list acceptedEntities that contains all entities that have been accepted so far (in the order in which they were first accepted). We note that **CheckID** cannot change the (internal) state of an instance; all changes caused by running **CheckID** are dropped after outputting a decision, i.e., the state of an instance is set back to the state before running **CheckID**.[13] In Appendix C 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 an 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**, as explained and motivated in §3.2, with the formal definition of these algorithms, including their default behavior if not specified, given in Appendix A. A protocol designer can access two useful framework specific variables for defining these algorithms: transcript, which, informally, contains a transcript of all messages sent and received by the current machine instance, and CorruptionSet, which contains all explicitly corrupted entities that are managed by the current machine instance. As these algorithms are part of our corruption conventions, they are used only if **Corruption model** is not set to *custom*.

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 similar to **Initialization** but is run once for each entity $(pid, sid, role)$ 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, 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,[14] 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, one would typically use **MessagePreprocessing** to prepare the input m for the **Main** algorithm, e.g., by dropping malformed messages or extracting some key information from m. The algorithm **Main** should 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 Appendix C; see our case study in §5 for examples.

This concludes the description of our template. As already mentioned, in Appendix I.2 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 §3.5).

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

[14] As mentioned in §3.2, if an entity is explicitly corrupted, it instead acts as a forwarder for messages to and from the adversary.

3.4 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 §3.1 and our case study in §5). In any case, the notation from §3.1 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 §3.1 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.

3.5 Realization Relation and Composition Theorems

In the following, we define the universal composability experiment and state the main composition theorem of iUC. Since iUC is an instantiation of the IITM model, as shown by our mapping mentioned in §3.3, both the experiment and theorem are directly carried over from the IITM model and hence do not need to be re-proven.

Definition 1 (Realization relation in iUC). *Let $\mathcal{P}$ and $\mathcal{F}$ be two environmentally bounded complete protocols with identical sets of public roles. The protocol $\mathcal{P}$* realizes *$\mathcal{F}$ (denoted by $\mathcal{P} \leq \mathcal{F}$) iff there exists a simulator (system) $\mathcal{S} \in \mathsf{Adv}(\mathcal{F})$ such that for all $\mathcal{E} \in \mathsf{Env}(\mathcal{P})$ it holds true that $\{\mathcal{E}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$.*[15]

Note that $\mathcal{E}$ 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 $\mathcal{F}$; see also Figure 1, where here we consider the case that $\mathcal{E}$ subsumes the adversary $\mathcal{A}$. (As shown in [1], whether or not the adversary $\mathcal{A}$ is considered does not change the realization relation. The resulting notions are equivalent.)

Now, the main composition theorem of iUC, which is a corollary of the composition of the IITM model, is as follows:

[15] Intuitively, the role names are used to determine which parts of $\mathcal{F}$ are realized by which parts of $\mathcal{P}$, hence they must have the same sets of public roles. Also, recall from §3.1 that the environment can use arbitrary (and also arbitrarily many) I/O connections of public roles. Hence, the realization proof has to hold true independently of (i) how many connections a public role provides to the environment (this is an arbitrary but fixed parameter in the proof) and (ii) which exact connections of a public role are used by the environment and which are used internally by (unknown) higher-level roles of $\mathcal{P}$, if any (this is also an arbitrary but fixed parameter). We emphasize that this requirement is trivially met by typical protocol specifications, where roles offer some kind of service/functionality to all (unknown) higher-level protocols.

Corollary 1 (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 environmentally bounded and complete, then $\mathcal{R} \leq \mathcal{I}$.*[16]

Just as in 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, again a corollary of a more general theorem in the IITM model. We discuss various types of composition, including composition for protocols with joint and/or global state, in more detail in §E.

4 Concepts and Discussion

Recall from the introduction that a main goal of iUC is to provide a flexible yet easy to use framework for universally composable protocol analysis and design. In this section, we briefly summarize and highlight some of the core concepts that allow us to retain the flexibility and expressiveness of the original IITM model while adding the usability with a handy set of conventions. We then highlight a selection of features that are supported by iUC due to the concepts iUC uses and that are not supported by other (conventions of) models, including the prominent UC and GNUC models. Our case study in §5 further illustrates the expressiveness of iUC. An extended discussion of concepts and features is available in Appendix E. Some of the most crucial concepts of iUC, discussed next, are the separation of entities and machine instances, public and private roles, a model independent interpretation of SIDs, support for responsive environments as well as a general addressing mechanism, which enables some of these concepts.

Separation of entities and machine instances: Traditionally, universal composability models do not distinguish between a machine instance and its interpretation. Instead, they specify that, e.g., a *real protocol instance* always represents a single party in a single session running a specific piece of code. Sometimes even composition theorems depend on this view. This has the major downside that, if the interpretation of a machine instance needs to be changed, then existing models, conventions, and composition theorems are no longer applicable and have to be redefined (and, in the case of theorems, reproven). For example, a typical *joint state protocol instance* [7, 20] manages a single party in *all sessions* and one role. Thus, in the case of the UC and GNUC models, the models had to be extended and reproven, including conventions and composition theorems. This is in contrast to iUC, which introduces the concept of *entities*. A protocol designer can freely define the interpretation of a machine instance by specifying the set of entities managed by that instance; the resulting protocol is still supported by our single template and the main composition theorem. This is a crucial feature that allows for the unified handling of real, ideal, joint-state, and (in combination with the next concept) also global state protocols.

We emphasize that this generality is made possible by the highly customizable addressing mechanism (**CheckID** in the template) used in iUC, which in turn is based on the very general addressing mechansim of the IITM model.

Public and private roles: Similar to the previous point, traditionally global state is defined by adding a special new global functionality with its own sets of conventions and proving specific global state composition theorems. However, whether or not state is global is essentially just a matter of access to that state. Our framework captures this property via the natural concept of *public roles*, which provides a straightforward way to make parts of a protocol accessible to the environment and other protocols. Thus, there is actually no difference between protocols with and without global state in terms of conventions or composition theorems in our framework.

[16] Technically speaking, public roles of $\mathcal{P}$ and $\mathcal{F}$ are not identically named but rather have a different prefix, i.e., a public role "$\mathcal{P}$: *role*" in $\mathcal{R}$ corresponds to/is replaced by the public role "$\mathcal{F}$: *role*" in $\mathcal{I}$. Hence, one technically has to slightly modify the higher level protocol $\mathcal{Q}$ to redirect messages appropriately, depending on which subroutine is to be used. By (slight) abuse of notation, we write $\mathcal{Q}$ in both systems $\mathcal{R}$ and $\mathcal{I}$ as the meaning is clear from the context.

A model independent interpretation of SIDs: In most other models, such as UC and GNUC, SIDs play a crucial role in the composition theorems. Composition theorems in these frameworks require protocols to either have disjoint sessions, where a session is defined via the SID, or at least behave as if they had disjoint sessions (in the case of joint-state composition theorems). This has two major implications: Firstly, one cannot directly model a protocol where different sessions share the same state and influence each other. This, however, is often the case for real world protocols that were not built with session separation in mind. For example, many protocols such as our case study (cf. §5) use the same signing key in multiple sessions, but do not include a session specific SID in the signature (as would be required for a joint-state realization). Secondly, sessions in ideal functionalities can consist only of parties sharing the same SID, which models so-called *global SIDs* or *pre-shared SIDs* [21]. That is, participants of a protocol session must share the same SID. This is in contrast to so-called *local SIDs* often used in practice, where participants with different SIDs can be part of the same protocol session (cf. 5.3). Because our main composition theorem is independent of (the interpretation of) SIDs, and in particular does not require state separation, we can also capture shared state and local SIDs in our framework.

Just as for the concept of entities and instances, this flexibility is made possible by the general addressing mechanism of iUC (and its underlying IITM model).

Support for responsive environments: Recall that responsive environments [1] allow for sending special messages on the network interface, called restricting messages, that have to be answered immediately by the adversary and environment. This is a very handy mechanism that allows protocols to exchange modeling related meta information with the adversary without disrupting the protocol run. For example, entities in our framework request their initial corruption status via a restricting message. Hence, the adversary has to provide the corruption status right away and the protocol run can continue as expected. Without responsive environments, one would have to deal with undesired behavior such as delayed responses, missing responses, as well as state changes and unexpected activations of (other parts of) the protocol before the response is provided. In the case of messages that exist only for modeling purposes, this adversarial behavior just complicates the protocol design and analysis without relating to any meaningful attack in reality, often leading to formally wrong security proofs and protocol specifications that cannot be re-used in practice. See Appendix E.8 and [1] for more information.

Selected Features of iUC. The iUC framework uses and combines the above concepts to support a wide range of protocols and composition types, some of which have not even been considered in the literature so far, using just *a single template* and *one main composition theorem*. We list some important examples:

i) Protocols with *local SIDs* and *global SIDs*, arbitrary forms of *shared state* including state that is shared across multiple protocol sessions, as well as *global state*. Our case study in §5 is an example of a protocol that uses and combines all of these protocol features, with a detailed explanation and discussion provided in §5.3.

ii) Ideal protocols that are structured into several subcomponents, unlike the monolithic ideal functionalities considered in other (conventions of) models. Parts of such structured ideal protocols can also be defined to be global, allowing for easily mixing traditional ideal protocols with global state. Again, this is also illustrated in our case study in §5. We also note that in iUC there is no need to consider so-called dummy machines in ideal protocols, which are often required in other models that do not allow for addressing the same machine instance with different IDs (entities).

iii) The general composition theorem, which in particular is agnostic to the specific protocols at hand, allows for combining and mixing classical composition of protocols with disjoint session, composition of joint-state protocols, composition of protocols with global state, and composition of protocols with arbitrarily shared state. One can also, e.g., realize a global functionality with another protocol (this required an additional composition theorem for the UC model [12] and is not yet supported by GNUC, whereas in iUC this is just another trivial special case of protocol composition). iUC even supports new types of compositions that have not been considered in the literature so far, such as joint-state realizations of two separate independent protocols (in contrast to traditional joint-state realizations of multiple independent sessions of the same protocol; cf. Appendix E.3).

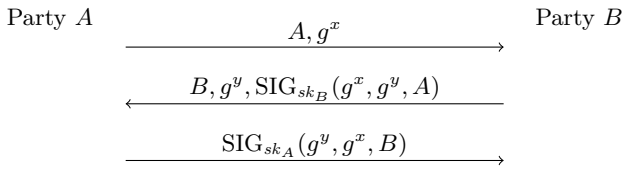


Fig. 5: ISO 9798-3 key exchange protocol for mutual authentication. A and B are the names of two parties that, at the end of the protocol, share a session key g^{xy}.

Besides our case study in §5, the flexibility and usability of iUC is also illustrated by another example in Appendix E, where we discuss that the iUC framework can capture the SUC model [10] as a mere special case. As already mentioned in the introdcution, the SUC model has been specifically designed for secure multi party computation (MPC) as a simpler version of the UC model, though it has to break out of (some technical aspects of) the UC model.

5 Case Study

In this section, we illustrate the usage of iUC by means of a concrete example, demonstrating usability, flexibility, and soundness of our framework. More specifically, we model and analyze a key exchange protocol of the ISO/IEC 9798-3 standard [17], an authenticated version of the Diffie-Hellman key exchange protocol, depicted in Figure 5. While this protocol has already been analyzed previously in universal composability models (e.g., in [6, 19]), these analyses were 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.

We have chosen this relatively simple protocol for our case study as it allows for showing how protocols can be modeled in iUC and highlighting several core features of the framework without having to spend much time on first explaining the logic of the protocol.

More specifically, our case study illustrates that our framework manages to combine *soundness* and *usability*: the specifications of the ISO protocol given in the figures below are formally complete, no details are swept under the rug, unlike the informal descriptions commonly encountered in the literature on universal composability. This allows for a precise understanding of the protocol, enabling formally sound proofs and re-using the protocol in higher-level protocols. At the same time, specifications of the ISO protocol are not overburdened by recurrent modeling related aspects as they make use of convenient defaults provided by the iUC framework. All parts of the ISO protocol are specified using *a single* template with one set of syntax rules, including real, ideal, and global state (sub-)protocols, allowing for a uniform treatment.

This case study also shows the *flexibility* of our framework: entites are grouped in different ways into machine instances to model different types of protocols and setup assumptions; we are able to share state across several sessions; we make use of the highly adjustable corruption model to precisely capture the desired corruption behavior of each (sub-)protocol; we are able to model both global state and locally chosen SIDs in a very natural way (we discuss some of these aspects, including locally chosen SIDs, in detail in §5.3).

We start by giving a high-level overview of how we model this ISO key exchange protocol in §5.1, then state our security result in §5.2, and finally discuss some of the features of our modeling in §5.3.

5.1 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 shown in Figure 3, which was partly explained already in §3.1. We provide a formal specification of $\mathcal{F}_{\mathrm{CA}}$ using our template and syntax in Figure 6. The remaining protocols specifications are given in Appendix B. The syntax is mostly self-explanatory, except for $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$, which denotes the currently active entity (that was accepted by **CheckID**),

$(\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})$, which denotes the entity that called the currently active entity on the I/O interface, and "_", which is a wildcard symbol. A formal definition of the syntax is provided in Appendix C. 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 §3.1, 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. [21]) 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)[17], 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 [20]), except that public keys are additionally registered with $\mathcal{F}_{\text{CA}}$ when being generated.

As also mentioned in §3.1, 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.

In summary, the real protocol that we analyze is the combined protocol $(\mathcal{P}_{\text{KE}}, \mathcal{F}_{\text{CA}} : \texttt{retrieval} \mid \mathcal{F}_{\text{sig-CA}}, \mathcal{F}_{\text{CA}} : \texttt{registration})$ (cf. left side of Figure 3). We note that we analyze this protocol directly in a multi-session setting. That is, the environment is free to spawn arbitrarily many entities belonging to arbitrary parties and having arbitrary local SIDs and thus there can be multiple key exchanges running in parallel. Analyzing a single session of this key exchange in isolation is not possible due to the shared signing keys and the use of local SIDs, which, as mentioned, precisely models how this protocol would usually be deployed in practice.[18]

We model the security properties of a multi-session 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

[17] Since we need only a single key pair per party, we set sid' to be the fixed value ϵ, i.e., the empty string.

[18] Note that this is true in *all* UC-like models that can express this setting: the assumption of disjoint sessions, which is necessary for performing a single session analysis, is simply not fulfilled by this protocol. This issue cannot even be circumvented by using a so-called joint-state realization for digital signatures, as such a realization not only requires global SIDs (cf. §5.3) but also changes the messages that are signed, thus creating a modified protocol with different security properties.

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. — *By this there is only a single machine instance that manages all entities.*

Main:

recv $(\texttt{Register}, key)$ **from** $\texttt{I/O}$ **to** $(\_, \_, \texttt{registration})$:
  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 $(\texttt{Register}, \texttt{success})$.

Allows every higher level protocol that connects to the `registration` *role to register a key. The key is stored for the PID and SID of the caller of $\mathcal{F}_{\text{CA}}$.*

recv $(\texttt{Retrieve}, (pid, sid))$ **from** $\_$ **to** $(\_, \_, \texttt{retrieval})$:
  reply $(\texttt{Retrieve}, \mathsf{keys}[pid, sid])$.

Everyone, including `NET`*, can retrieve keys registered by someone with PID pid and SID sid.*

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

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 §5.3). 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 13).

5.2 Security Result

For the above modeling, we obtain the following result.

Theorem 2. *Let* $\mathsf{groupGen}(1^\eta)$ *be an algorithm that outputs descriptions (G, n, g) of cyclical groups (i.e., G is a group of size n with generator g) such that n grows exponentially in η and the DDH assumption holds true. Then we have:*

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

Proof. Given in Appendix D.[19]

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 9 in the Appendix):

[19] We note that the realization proof trivially meets the requirements of Footnote 15 because all (sub-)protocols from our case study provide their services to all higher-level protocols and the environment no matter which exact connections they use. This is really the standard case for natural protocol definitions.

Lemma 1. *If the digital signature scheme used in $\mathcal{P}_{\text{sig-CA}}$ is existentially unforgable under chosen message attacks (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 [20].

By Corollary 1, 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 2. *If the conditions of Theorem 2 and Lemma 1 are fulfilled, then*

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

5.3 Discussion

In the following, we highlight some of the key details of our protocol specification where we are able to model reality very precisely and in a natural way, illustrating the *flexibility* of iUC, also compared to (conventions of) the UC and GNUC models.

Local SIDs: Many real-world protocols, including the key exchange in our case study, use so-called local session IDs in practice (cf. [21]). 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, but is not used as part of the actual protocol logic. In particular, multiple entities can 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 adding an additional roundtrip to exchange nonces, or that is chosen by and then transmitted from one entity to the others during the protocol run. As illustrated by the protocols $\mathcal{P}_{\text{KE}}$ (and $\mathcal{F}_{\text{KE}}$) in our case study, iUC can easily model such local SIDs in a natural way. This is in contrast to several other UC-like models, including the UC and GNUC models, that are built around global SIDs and thus do not directly support local SIDs with their conventions. While it might be possible to find workarounds by ignoring conventions, e.g., by modeling all sessions of a protocol in a single machine instance M, i.e., essentially ignoring the model's intended SID mechanism and taking care of the addressing of different sessions with another layer of SIDs within M itself, this has two major drawbacks: Firstly, it decreases overall usability of the models as this workaround is not covered by existing conventions of these models. Secondly, existing composition theorems of UC and GNUC do not allow one to compose such a protocol with a higher-level protocol modeled in the "standard way" where different sessions use different SIDs.[20] We emphasize that the difference between local and global SIDs is not just a minor technicality or a cosmetic difference: as argued by Küsters et al. [21], there are natural 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.

Shared State: In iUC, entities can easily and naturally share arbitrary state in various ways, even across multiple protocol sessions, if so desired. This is illustrated, e.g., by $\mathcal{P}_{\text{KE}}$ in our case study, where every party uses just a single signature key pair across arbitrarily many key exchanges. This allows for a very flexible and precise modeling of protocols. In particular, for many real-world protocols this modeling is much more precise than so-called joint-state realizations that are often used to share state between sessions in UC-like models

[20] This is because such a higher level protocol would then access the same subroutine session throughout many different higher-level sessions, which violates session disjointness as required by both UC and GNUC.

that assume disjoint sessions to be the default, such as the UC and GNUC models. Joint-state realizations have to modify protocols by, e.g., prefixing signed messages with some globally unique SID for every protocol session (which is not done by many real-world protocols, including our case study). Thus, even if the modified protocol is proven to be secure, this does not imply security of the unmodified one. The UC and GNUC models do not directly support state sharing without resorting to joint-state realizations or global functionalities. While one might be able to come up with workarounds similar to what we described for local SIDs above, this comes with the same drawbacks in terms of usability and flexibility.

Global State: Our concept of public and private roles allows us to not only easily model global state but also to specify, in a convenient and flexible way, machines that are only partially global. This is illustrated by $\mathcal{F}_{\text{CA}}$ in our case study, which allows arbitrary other protocols to retrieve keys but limits key registration to one specific protocol to model that honest users will not register their signing keys for other contexts (which, in general, otherwise voids all security guarantees). This feature makes $\mathcal{F}_{\text{CA}}$ easier to use as a subroutine than the existing global functionality $\mathcal{G}_{bb}$ for certificate authorities by Canetti et al. [12], which does not support making parts of the functionality "private". Thus, everyone has full access to all operations of $\mathcal{G}_{bb}$, including key registration, allowing the environment to register keys in the name of (almost) arbitrary parties, even if they are supposed to be honest.

Note that our formulation of $\mathcal{F}_{\text{CA}}$ means that, if the ideal protocol $(\mathcal{F}_{\text{KE}}, \mathcal{F}_{\text{CA}} : \texttt{retrieval} \mid \mathcal{F}_{\text{CA}} : \texttt{registration})$ is used as a subroutine for a new hybrid protocol, then only $\mathcal{F}_{\text{KE}}$ but not the higher-level protocol can register keys in $\mathcal{F}_{\text{CA}}$. If desired, one can, however, also obtain a single global $\mathcal{F}_{\text{CA}}$ where both $\mathcal{F}_{\text{KE}}$ and the higher-level protocol can store keys in the following way: First analyze the whole hybrid protocol while using a second separate copy of $\mathcal{F}_{\text{CA}}$, say $\mathcal{F}'_{\text{CA}}$, where only the higher-level protocol can register keys. After proving this to be secure (which is simpler than directly using a global CA where multiple protocols register keys), one can replace both $\mathcal{F}_{\text{CA}}$ and $\mathcal{F}'_{\text{CA}}$ with a joint-state realization where keys are stored in and retrieved from the same $\mathcal{F}_{\text{CA}}$ subroutine along with a protocol dependent tag (see Appendix E.3 for this novel type of joint-state realization). Of course, this approach can be iterated to support arbitrarily many protocols using the same $\mathcal{F}_{\text{CA}}$. This modeling reflects reality where keys are certified for certain contexts/purposes.

6 Conclusion

We have introduced the iUC framework for universal composability. As illustrated by our case study, iUC is highly *flexible* in that it supports a wide range of protocol types, protocol features, and composition operations. This flexibility is combined with greatly improved *usability* compared to the IITM model due to its protocol template that fixes recurring modeling related aspects while providing sensible defaults for optional parts. Adding usability while preserving flexibility is a difficult task that is made possible, among others, due to the concepts of roles and entities; these concepts allow for having just *a single template* and *two composition theorems* that are able to handle arbitrary types of protocols, including real, ideal, joint-state, and global ones, and combinations thereof. The flexibility and usability provided by iUC also significantly facilitates the precise modeling of protocols, which is a prerequisite for carrying out formally complete and sound proofs. Our formal mapping from iUC to the IITM shows that iUC indeed is an instantiation of the IITM, and hence, immediately inherits all theorems, in particular, all composition theorems, of the IITM model. Since we formulate these theorems also in the iUC terminology, protocol designers can completely stay in the iUC realm when designing and analyzing protocols.

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. Camenisch, J., Enderlein, R.R., Krenn, S., Küsters, R., Rausch, D.: Universal Composition with Responsive Environments. In: Cheon, J.H., Takagi, T. (eds.) Advances in Cryptology - ASIACRYPT 2016 - 22nd International

Conference on the Theory and Application of Cryptology and Information Security. Lecture Notes in Computer Science, vol. 10032, pp. 807–840. Springer (2016), available at http://eprint.iacr.org/2016/034
2. Camenisch, J., Krenn, S., Küsters, R., Rausch, D.: iUC: Flexible Universal Composability Made Simple. In: ASIACRYPT 2019 - 25th International Conference on the Theory and Applications of Cryptology and Information Security (2019), to appear
3. Canetti, R.: Universally Composable Security: A New Paradigm for Cryptographic Protocols. Tech. Rep. 2000/067, Cryptology ePrint Archive (2000), available at http://eprint.iacr.org/2000/067 with new versions from December 2005, July 2013, and December 2018
4. Canetti, R.: Universally Composable Security: A New Paradigm for Cryptographic Protocols. In: Proceedings of the 42nd Annual Symposium on Foundations of Computer Science (FOCS 2001). pp. 136–145. IEEE Computer Society (2001)
5. Canetti, R., Dodis, Y., Pass, R., Walfish, S.: Universally Composable Security with Global Setup. In: Vadhan, S.P. (ed.) Theory of Cryptography, Proceedings of TCC 2007. Lecture Notes in Computer Science, vol. 4392, pp. 61–85. Springer (2007)
6. Canetti, R., Krawczyk, H.: 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. Lecture Notes in Computer Science, vol. 2332, pp. 337–351. Springer (2002)
7. Canetti, R., Rabin, T.: Universal Composition with Joint State. In: Advances in Cryptology, 23rd Annual International Cryptology Conference (CRYPTO 2003), Proceedings. Lecture Notes in Computer Science, vol. 2729, pp. 265–281. Springer (2003)
8. Canetti, R., Chari, S., Halevi, S., Pfitzmann, B., Roy, A., Steiner, M., Venema, W.Z.: Composable Security Analysis of OS Services. In: Applied Cryptography and Network Security - 9th International Conference, ACNS 2011, Nerja, Spain, June 7-10, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6715, pp. 431–448 (2011)
9. Canetti, R., Cheung, L., Kaynar, D.K., Liskov, M., Lynch, N.A., Pereira, O., Segala, R.: Analyzing Security Protocols Using Time-Bounded Task-PIOAs. Discrete Event Dynamic Systems **18**(1), 111–159 (2008)
10. Canetti, R., Cohen, A., Lindell, Y.: 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. Lecture Notes in Computer Science, vol. 9216, pp. 3–22. Springer (2015)
11. Canetti, R., Hogan, K., Malhotra, A., Varia, M.: A Universally Composable Treatment of Network Time. In: 30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017. pp. 360–375. IEEE Computer Society (2017)
12. Canetti, R., Shahaf, D., Vald, M.: 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. Lecture Notes in Computer Science, vol. 9615, pp. 265–296. Springer (2016)
13. Chaidos, P., Fourtounelli, O., Kiayias, A., Zacharias, T.: 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. Lecture Notes in Computer Science, vol. 11274, pp. 191–221. Springer (2018)
14. Chari, S., Jutla, C.S., Roy, A.: Universally Composable Security Analysis of OAuth v2.0. IACR Cryptology ePrint Archive **2011**, 526 (2011)
15. Hofheinz, D., Shoup, V.: GNUC: A New Universal Composability Framework. J. Cryptology **28**(3), 423–508 (2015)
16. Hogan, K., Maleki, H., Rahaeimehr, R., Canetti, R., van Dijk, M., Hennessey, J., Varia, M., Zhang, H.: On the Universally Composable Security of OpenStack. IACR Cryptology ePrint Archive **2018**, 602 (2018)
17. ISO/IEC IS 9798-3, Entity authentication mechanisms — Part 3: Entity authentication using assymetric techniques (1993)
18. Küsters, R.: Simulation-Based Security with Inexhaustible Interactive Turing Machines. In: Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW-19 2006). pp. 309–320. IEEE Computer Society (2006), see [22] for a full and revised version.
19. Küsters, R., Rausch, D.: A Framework for Universally Composable Diffie-Hellman Key Exchange. In: IEEE 38th Symposium on Security and Privacy (S&P 2017). pp. 881–900. IEEE Computer Society (2017)
20. Küsters, R., Tuengerthal, M.: 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).

pp. 270–284. IEEE Computer Society (2008), the full version is available at https://eprint.iacr.org/2008/006 and will appear in Journal of Cryptology
21. Küsters, R., Tuengerthal, M.: Composition Theorems Without Pre-Established Session Identifiers. In: Chen, Y., Danezis, G., Shmatikov, V. (eds.) Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011). pp. 41–50. ACM (2011)
22. Küsters, R., Tuengerthal, M., Rausch, D.: The IITM Model: a Simple and Expressive Model for Universal Composability. Tech. Rep. 2013/025, Cryptology ePrint Archive (2013), available at http://eprint.iacr.org/2013/025. To appear in Journal of Cryptology
23. Maurer, U.: Constructive Cryptography - A New Paradigm for Security Definitions and Proofs. In: TOSCA 2011. LNCS, vol. 6993, pp. 33–56 (2011)
24. Maurer, U., Renner, R.: Abstract Cryptography. In: Chazelle, B. (ed.) Innovations in Computer Science - ICS 2010. Proceedings. pp. 1–21. Tsinghua University Press (2011)
25. Pfitzmann, B., Waidner, M.: A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission. In: IEEE Symposium on Security and Privacy. pp. 184–201. IEEE Computer Society (2001)

A Full Definitions of Corruption Behavior Algorithms

This element of the template allows for customization of corruption related behavior of machines by specifying a custom version of one or more of the optional algorithms **DetermineCorrStatus**, **AllowCorruption**, **LeakedData**, and **AllowAdvMessage**. These algorithms are used to customize our corruption model, as explained in §3.2 (cf. that section for possible use cases). 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 next.

The **DetermineCorrStatus**$(pid, sid, role)$ algorithm is used to customize the response upon receiving `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 the message $(\texttt{CorruptionStatus}, \texttt{true})$ 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. If not specified, the **DetermineCorrStatus** algorithm always returns `false`. We note that *no other actions are performed* during or after responding to a `CorruptionStatus`? request. In particular, neither **Initialization**, **Main**, or such is performed nor is the adversary asked whether he wants to corrupt an entity.[21]

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, for example, depend on the corruption states of (entities in) subroutines, as explained in §3.2. 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 for eample 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**$(pid, sid, role)$ algorithm is used to determine the data that is leaked to the adversary upon successful (explicit) corruption of an entity $(pid, sid, role)$; 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

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

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_{receiver}, sid_{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 some other receiving 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 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. Note that this default does not restrict the adversary from sending messages to entities with a different SID. If this property is desired, for example, to model disjoint protocol sessions that do not interact with each other, then this algorithm can be customized appropriately.

B Postponed Protocol Definitions

This section provides the remaining definitions for the protocols from our case study in §5. The protocols are specified in Figures 7 to 14. We provide a formal definition of the syntax used for specifying these protocols in Appendix C.

C Notation

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

C.1 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).

C.2 Special variables

For notational convenience, each instance maintains two framework-specific global variables, namely the variable $\mathsf{entity}_{\mathsf{cur}} = (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ and the variable $\mathsf{entity}_{\mathsf{call}} = (\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})$, both consisting of a PID, an SID, and a role. The former triple, $\mathsf{entity}_{\mathsf{cur}}$, specifies the entity that was accepted by **CheckID** 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.

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 signing and verifications 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 $sid = (pid', sid')$.
If this check fails, output `reject`.
Otherwise, accept all entities with the same SID.

{ *An instance manages all parties and roles in a single session. See Appendix C.3 for the formal definition of this abbreviated statement.*

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`;[a]
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:

See Figure 8.

[a] By sending the message responsively, the adversary is forced to provide the expected answer before interacting with the protocol again. This makes the definition of $\mathcal{F}_{\text{sig-CA}}$ simpler as we do not have to specify what happens if the adversary does not provide the expected response. It also makes $\mathcal{F}_{\text{sig-CA}}$ easier to use for higher-level protocols as they get the guarantee that, when they send, e.g., a `sign` request, the response will be returned immediately without the adversary being able to interfere in unexpected ways (just as is the case in an actual implementation of a signature scheme). This in turn simplifies proofs as there are less edge cases to consider. See [1] for a more detailed discussion of the advantages and use cases of this mechanism.

Fig. 7: The ideal signature functionality $\mathcal{F}_{\text{sig-CA}}$.

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

Main:

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

recv (`Sign`, msg) **from** `I/O` **to** (pidowner, _, `signer`) **s.t.** KeysGenerated = `ready`:
$\sigma \leftarrow \mathsf{sig}^{(\mathsf{p})}(msg, \mathsf{sk})$.
{*Run at most polynomially many steps of* sig. *See Appendix C.7 for a formal definition.*
$b \leftarrow \mathsf{ver}^{(\mathsf{p})}(msg, \sigma, \mathsf{pk})$.
{*Sign and check that verification succeeds.*
if $\sigma = \bot \vee b \neq$ `true`:
reply (`Signature`, $\bot$).
{*Signing or verification test failed.*
else:
add msg **to** msglist.
reply (`Signature`, σ).
{*Record msg for verification and return signature.*

recv (`Verify`, msg, σ, pk) **from** `I/O` **to** (_, _, `verifier`):
$b \leftarrow \mathsf{ver}^{(\mathsf{p})}(msg, \sigma, pk)$.
{*Verify signature.*
if $pk = \mathsf{pk} \wedge b =$ `true` $\wedge\ msg \notin$ msglist $\wedge$ (pidowner, $\mathsf{sid}_{\mathsf{cur}}$, `signer`) $\notin$ CorruptionSet:
{*cf. §3.3 for* CorruptionSet.
reply (`VerResult`, `false`).
{*Prevent forgery.*
else:
reply (`VerResult`, b).
{*Return verification result.*

Fig. 8: The ideal signature functionality $\mathcal{F}_{\text{sig-CA}}$ (continued).

C.3 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 (i.e., the first entity in the list acceptedEntities). Output `accept` iff $pid = pid_1$; output `reject` otherwise."

The above specification of **CheckID** of a machine M says that each instance of M is responsible for all entities belonging to a single party pid, where different instances are responsible for different parties. In other words, a machine instance models a party. More specifically, if an instance is fresh, then it will just accept the first entity it sees. This fixes the party pid that is managed by this machine instance. For all following activations, an entity is accepted iff it belongs to pid. Note that, since entities of party pid are never rejected, there will never be a second instance of M that also accepts pid.

The informal statements

"accept all entities with the same SID",
"accept all entities with the same SID and role", etc.

are interpreted analogously to "accept all entities with the same PID". We also allow for using the statement

"accept a single entity",

which is formally defined as follows: "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 (i.e., the first entity in the list acceptedEntities). Output `accept` iff $(pid, sid, role) = (pid_1, sid_1, role_1)$; output `reject` otherwise."

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

Participating roles: $\texttt{signer}, \texttt{verifier}$
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): $\texttt{signer}$
Subroutines: $\mathcal{F}_{\text{CA}} : \texttt{registration}$
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 $\texttt{reject}$.
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 $role_{\text{receiver}} = \mathcal{F}_{\text{CA}} : \texttt{registration}$, also check that $sid = (pid, sid')$ (for some sid').
If all checks succeed, output $\texttt{true}$, otherwise output $\texttt{false}$.

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

Main:

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

recv $(\texttt{Sign}, msg)$ **from** $\texttt{I/O}$ **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): $\texttt{verifier}$
CheckID$(pid, sid, role)$**:**
Check that $sid = (pid', sid')$; otherwise output $\texttt{reject}$.
Accept a single entity. {*An instance manages exactly one entity.*

Main:

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

Fig. 9: The real signature protocol $\mathcal{P}_{\text{sig-CA}}$.

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

Participating roles: `initiator`, `responder`, `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): `initiator`
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.*
- $\mathsf{state} \in \{\bot, \texttt{started}, \texttt{inSession}, \texttt{finished}\} = \bot$ {*Current state in key exchange.*
- $\mathsf{caller} \in (\{0,1\}^*)^3 \cup \{\bot\} = \bot$ {*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 or the verification subroutine is corrupted.*

  if $\mathsf{intendedPID} = \bot$:
    return **corr**$(pid, (pid, \epsilon), \mathcal{F}_{\text{sig-CA}} : \texttt{signer})$.

  else:
    return **corr**$(pid, (pid, \epsilon), \mathcal{F}_{\text{sig-CA}} : \texttt{signer})$
      $\vee$ **corr**$(\mathsf{intendedPID}, (\mathsf{intendedPID}, \epsilon), \mathcal{F}_{\text{sig-CA}} : \texttt{signer})$
      $\vee$ **corr**$(pid, (\mathsf{intendedPID}, \epsilon), \mathcal{F}_{\text{sig-CA}} : \texttt{verifier})$.

  {*The* **corr** *macro sends* `CorruptionStatus?` *to the specified entity, waits for the response, and then outputs that response.*

- **AllowAdvMessage**$(pid, sid, role, pid_{\text{receiver}}, sid_{\text{receiver}}, role_{receiver}, m)$:
  If $role_{receiver} = \mathcal{F}_{\text{sig-CA}} : \texttt{signer}$, return `false`.[a]
  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:
  See Figure 11.

[a] In our modeling, the corruption status of `signer` entities indicates whether the adversary has access to the corresponding signature keys, i.e., whether he can sign his own messages (as in this case the `signer` entity should be considered compromised). Thus, the adversary is not allowed to access uncorrupted `signer` entities. If the signer entity is corrupted, then the adversary already knows the secret key and can sign messages on his own, so there is no need to give him access in this case.

Fig. 10: A real key exchange protocol $\mathcal{P}_{\text{KE}}$ for realizing $\mathcal{F}_{\text{KE}}$ (part 1). Note that each instance of $M_{\texttt{initiator}}$ and $M_{\texttt{responder}}$ corresponds to a single entity.

Description of $M_{\texttt{initiator}}$ (continued):

Main:

recv $(\texttt{InitKE}, \mathit{intendedPartner})$ **from** $\texttt{I/O}$ **s.t.** $\mathsf{state} = \bot$: { *Start KE and send first message.*

$(\mathsf{state}, \mathsf{caller}, \mathsf{intendedPID}) \leftarrow (\texttt{started}, \mathsf{entity}_{\mathsf{call}}, \mathit{intendedPartner})$.
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.** $\mathsf{state} = \texttt{finished}$:[a]
$m = (\mathsf{h}_{\mathsf{resp}}, \mathsf{g}^{\mathsf{e}_{\mathsf{init}}}, \mathsf{intendedPID})$.
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** $\texttt{NET}$.

[a] Allow the adversary to retrieve the third protocol message. This modeling 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). We note that this issue is not specific to the IITM model or iUC but also appears in other models, such as the UC and GNUC models.

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

Implemented role(s): $\texttt{setup}$
Internal state:
– $(\mathsf{G}, \mathsf{n}, \mathsf{g}) \in (\{0,1\}^* \cup \{\bot\})^3 = (\bot, \bot, \bot)$ { *Global group parameters.*
CheckID$(\mathit{pid}, \mathit{sid}, \mathit{role})$**:** Accept all entities.
Corruption behavior:
– **AllowCorruption**$(\mathit{pid}, \mathit{sid}, \mathit{role})$: 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. 11: A real key exchange protocol $\mathcal{P}_{\text{KE}}$ for realizing $\mathcal{F}_{\text{KE}}$ (part 2).

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.*
- $\mathsf{state} \in \{\bot, \texttt{started}, \texttt{inSession}, \texttt{finished}\} = \bot$ {*Current state in key exchange.*
- $\mathsf{caller} \in (\{0,1\}^*)^3 \cup \{\bot\} = \bot$ {*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 or the verification subroutine is corrupted.*

  if $\mathsf{intendedPID} = \bot$: return **corr**$(pid, (pid, \epsilon), \mathcal{F}_{\text{sig-CA}} : \texttt{signer})$.
  else: return **corr**$(pid, (pid, \epsilon), \mathcal{F}_{\text{sig-CA}} : \texttt{signer})$
  $\vee$ **corr**$(\mathsf{intendedPID}, (\mathsf{intendedPID}, \epsilon), \mathcal{F}_{\text{sig-CA}} : \texttt{signer})$
  $\vee$ **corr**$(pid, (\mathsf{intendedPID}, \epsilon), \mathcal{F}_{\text{sig-CA}} : \texttt{verifier})$.

- **AllowAdvMessage**$(pid, sid, role, pid_{\text{receiver}}, sid_{\text{receiver}}, role_{receiver}, m)$:
  If $role_{receiver} = \mathcal{F}_{\text{sig-CA}} : \texttt{signer}$, return `false`. {*cf. explanation in Figure 10.*
  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.*
$(\mathsf{state}, \mathsf{caller}, \mathsf{intendedPID}) \leftarrow (\texttt{started}, \mathsf{entity}_{\mathsf{call}}, intendedPartner)$.
send `InitKE` **to** `NET`. {*Notify network that the key exchange has started and the responder is ready to receive the first message.*

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. 12: A real key exchange protocol $\mathcal{P}_{\text{KE}}$ for realizing $\mathcal{F}_{\text{KE}}$ (part 3).

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.*
- $\mathsf{state} : (\{0,1\}^*)^3 \to \{\bot, \texttt{started}, \texttt{inSession}, \texttt{finished}\}$ {*Stores the current state of entities in key exchange; initially $\bot$.*
- $\mathsf{caller} : (\{0,1\}^*)^3 \to (\{0,1\}^*)^3 \cup \{\bot\}$ {*Stores the calling entity for each entity in key exchange; initially $\bot$.*
- $\mathsf{intendedPID} : (\{0,1\}^*)^3 \to \{0,1\}^* \cup \{\bot\}$ {*Mapping from entity to intended partner PID; initially $\bot$.*
- $\mathsf{sessions} \subseteq (\{0,1\}^*)^3 \times (\{0,1\}^*)^3 = \emptyset$ {*Pairs of entities in the same global key exchange session.*
- $\mathsf{sessionKeys} : (\{0,1\}^*)^3 \to \{0,1\}^* \cup \{\bot\}$ {*Mapping from entity to session key; initially $\bot$.*

CheckID$(pid, sid, role)$**:** Accept all entities. {*All entites are managed in a single instance, which then internally pairs them to "key exchange sessions".*

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:
    (G, n, g) ← groupGen(1^η).
    if sender = NET ∧ m = InitGroup:
        send (LeakGroup, (G, n, g)) to NET.
    else:
        send responsively (LeakGroup, (G, n, g)) to NET;
        wait for _.
```

{*Allow adversary to start initialization and then return the generated group. No other actions are performed in this case.*

{*Leak the group parameters to the adversary. Note that this command forces the adversary to respond before interacting with the protocol in any other way, i.e., the run can continue as expected.*

Main: See Figure 14.

Fig. 13: The ideal key exchange functionality $\mathcal{F}_{\text{KE}}$ (part 1).

C.4 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 the 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

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

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 $entity_I$ as $(pid_I, sid_I, \texttt{initiator})$ and
  parse $entity_R$ as $(pid_R, sid_R, \texttt{responder})$ with the following constraints:
    $(\mathsf{intendedPID}[entity_I] = pid_R \vee entity_I \in \mathsf{CorruptionSet})$[a]
  $\wedge$ $(\mathsf{intendedPID}[entity_R] = pid_I \vee entity_R \in \mathsf{CorruptionSet})$
  $\wedge$ $(\mathsf{state}[entity_I] = \texttt{started} \vee entity_I \in \mathsf{CorruptionSet})$[b]
  $\wedge$ $(\mathsf{state}[entity_R] = \texttt{started} \vee entity_R \in \mathsf{CorruptionSet})$.
  if the above parsing succeeds:
    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.
    $\mathsf{state}[entity_I] \leftarrow \texttt{inSession}$.
    $\mathsf{state}[entity_R] \leftarrow \texttt{inSession}$.
    reply $(\texttt{GroupSession}, \texttt{success})$.
    { *By this, honest entities cannot be grouped into a second group; however, corrupted ones can be part of many groups.*
  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:[c]
  send $(\texttt{Register}, pk)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \epsilon, \mathcal{F}_{\mathrm{CA}} : \texttt{registration})$;
  wait for $(\texttt{Register}, response)$.
  reply $(\texttt{Register}, response)$.

[a] If an entity is not corrupted: ensure that its partner is correct.
[b] If an entity is not corrupted: ensure that it has started the exchange and is not already in another session.
[c] Allow the adversary to register arbitrary keys in $\mathcal{F}_{\mathrm{CA}}$ for honest entities. Note that $\mathcal{F}_{\mathrm{KE}}$ provides security for session keys independently of any keys stored in $\mathcal{F}_{\mathrm{CA}}$, so giving the adversary full access also for honest entities does not weaken security guarantees.

Fig. 14: The ideal key exchange functionality $\mathcal{F}_{\mathrm{KE}}$ (part 2).

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 `SUB` if a message is to be received on the I/O interface from some arbitrary (known) subroutine, the constant bit string `I/O` if a message is to be received on the I/O interface from some arbitrary (unknown) higher-level protocol/the environment, 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 a fixed bit string of a known subroutine, a fixed number indicating a connection to some (unknown) higher-level protocol, or a variable denoting an I/O connection to a subroutine or higher-level protocol (cf. paragraph "exchanging messages" in §3.1). 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 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 the header can be omitted altogether and it suffices to provide ⟨code⟩ only.

C.5 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 ⟨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 I.2.6 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 (see also the remarks in Appendix I.2.6).
As before, ⟨sender⟩ can be omitted. In that case, the sender will default to $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$.

– In line with Camenisch et al. [1], 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. [1] and Appendix I.2.6 for details.

C.6 Macros

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

– To obtain the corruption status of a subroutine entity $(pid_{sub}, sid_{sub}, role_{sub})$ one can use the following macro:

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

  Formally, this macro sends the special message `CorruptionStatus`? to $entity_{sub}$, which, as explained in Section 3.2, will respond with the corruption status of the entity (note that, when using default algorithms, 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 I.2.6.

C.7 Running externally provided algorithms

It is sometimes necessary to run an arbitrary algorithm alg provided by the adversary or environment. For example, this is a common mechanism used in ideal functionalities such as $\mathcal{F}_{\text{sig-CA}}$ (see Figures 7 and 8) to provide security guarantees independently of a specific algorithm. However, as alg is an arbitrary algorithm which might not terminate within polynomial time, a protocol $\mathcal{P}$ running alg would no longer be environmentally bounded.

To solve this issue, we introduce the following syntax: given a polynomial $p \in \mathbb{Z}[x]$, we write $\mathsf{alg}^{(p)}(x)$ to say that we simulate the algorithm alg with input x, but abort the simulation after at most $p(\eta + |x|)$ steps. The overall output of the simulation is defined to be $\perp$ if the simulation is aborted, and otherwise to be the output of alg.

Note that since the polynomial p is fixed, a protocol $\mathcal{P}$ can run $\mathsf{alg}^{(p)}(x)$ while still being environmentally bounded. Also note that for every PPT algorithm alg, there is a runtime bound p such that $\mathsf{alg}^{(p)}(x)$ never aborts and thus behaves just as $\mathsf{alg}(x)$.

D Security Proof of our Case Study

In this section we provide a proof sketch of Theorem 2, 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 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 the *full protocol* $\mathcal{R}$, including its behavior on the network interface to the environment. Note that this simulation includes in particular a second copy of $\mathcal{F}_{\text{CA}}$; the simulator manually ensures that the public verification keys stored in this internally simulated copy are consistent with the keys stored in $\mathcal{F}_{\text{CA}}$ of $\mathcal{I}$. 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 $(\texttt{InitKE}, \mathit{entity}_{\mathit{partner}})$ from an honest[22] entity $\mathit{entity}_{\mathit{sender}}$, $\mathcal{S}$ forwards this message in the name of a higher-level protocol to the simulated entity $\mathit{entity}_{\mathit{sender}}$ in $\mathcal{R}$. This triggers the start of a key exchange in the simulation.
- As soon as an honest entity $\mathit{entity}_1 = (\mathit{pid}_1, \mathit{sid}_1, \mathit{role}_1)$ 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 $\mathit{entity}_2 = (\mathit{pid}_2, \mathit{sid}_2, \mathit{role}_2)$ such that role_2 is `responder` and both entities agree on the public Diffie-Hellman key shares of each other. Then $\mathcal{S}$ sends $(\texttt{GroupSession}, \mathit{entity}_1, \mathit{entity}_2)$ to some honest entity[23] 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 $\mathcal{F}_{\text{CA}}$ of $\mathcal{I}$ by sending a `Register` request via an entity of $\mathcal{F}_{\text{KE}}$ with the same PID and SID.[24]
- 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.

[22] 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.

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

[24] Recall that, by the definition of $\mathcal{F}_{\text{KE}}$ given in Figure 14, the simulator may register arbitrary keys in its subroutine $\mathcal{F}_{\text{CA}}$ also in the name of honest entities. This does not weaken security guarantees as $\mathcal{F}_{\text{KE}}$ provides security for session keys of honest entities independently of $\mathcal{F}_{\text{CA}}$.

– 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 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 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 the keys in $\mathcal{F}_{\text{CA}}$ (of $\mathcal{I}$) are consistent with the simulation of $\mathcal{R}$ in $\mathcal{S}$. Also observe that $\mathcal{S}$ already simulates $\mathcal{R}$ perfectly on the network interface. In particular, upon corruption of a simulated public entity of $\mathcal{P}_{\text{KE}}$, $\mathcal{S}$ 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 of $\mathcal{F}_{\text{KE}}$: 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.[25] 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, $entity_{intended}$ is a `responder` with overwhelming probability.
2. By definition of $\mathcal{S}$, honest `responder` entities are paired only if there is an honest `initiator` entity that has accepted the signature on the second protocol message $(\mathsf{h}_{\mathsf{init}}, \mathsf{h}_{\mathsf{resp}}, pid_{init})$. This message is accepted only 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.

[25] Note that here we need that $\mathcal{F}_{\text{KE}}$ allows for dynamic corruption: while entities of $\mathcal{P}_{\text{KE}}$ can be explicitly corrupted only in a static way, they also forward the corruption state of signature keys in $\mathcal{F}_{\text{sig-CA}}$, which can change dynamically at any point.

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 two honest entities. Since we consider static corruption of the real protocol, honest entities can only be explicitly corrupted at the start of the run, and thus will never leak 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 $\mathcal{E}$. □

E Applying iUC

This section provides further explanation of how iUC can be used to capture various kinds of protocols and settings in natural ways, illustrating its flexibility and expressiveness. As part of this 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 §3.5). We emphasize that all of these cases, except for unbounded self composition, are covered directly by the concurrent composition theorem (cf. Corollary 1). Thus *a single theorem* covers most use cases, which illustrates the power and generality of the composition theorem in our framework.

E.1 Single Session Composition

In addition to concurrent composition (cf. §3.5 and Corollary 1), 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). On 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 3 (Unbounded self composition (informal)). *Let $\mathcal{P}$ and $\mathcal{F}$ be two complete protocols such that they are 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 F, 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 F. Note that, 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 comprise all entites that share the same prefix of their SIDs, or share the same (prefix of the) PID. This is in contrast to many other UC-like models, including the UC and GNUC models, that fix in their model and theorems how a "session" is defined.

E.2 Joint-State Composition for Multiple Sessions

Modeling protocols with disjoint sessions (cf. §E.1) 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 [7, 15, 20]. 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 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 §E.1, 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. For a more concrete example of a joint-state realization in our framework, we refer the reader to Appendix G.

In general, we say that a protocol $\mathcal{P}_{js}$ has joint-state if it shares some state between sessions, but is still supposed to behave just like a protocol $\mathcal{F}$ that does not share state between sessions. This notion of a protocol with joint-state is just a special case of the very general protocol definition of iUC, unlike in many other models. Hence, once we have shown that $\mathcal{P}_{js}$ realizes $\mathcal{F}$ (as per Definition 1), we can directly apply our main composition theorem (Corollary 1) as follows, where clearly this corollary applies no matter which inner structure $\mathcal{P}_{js}$ has, and in fact, no matter whether it is a joint-state protocol or not:

Corollary 4 (Concurrent composition with joint-state). *Let $\mathcal{P}_{js}$ be an environmentally bounded protocol with joint-state and $\mathcal{F}$ be a environmentally bounded protocol with disjoint sessions such that $\mathcal{P}_{js} \leq \mathcal{F}$. 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 environmentally bounded, then

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

Proof. This is a mere special case of Corollary 1 and as such trivially implied.

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 the handling of joint-state for protocol designers as they are able to work with the same syntax, definitions, and theorems. Also, by this, joint-state can seamlessly be combined with other concepts (e.g., other forms of shared state, including global state).

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 $\mathcal{P}_{\text{sig}}^{js}$ that re-uses the same signature key in all sessions. Now, if an adversary corrupts the single signing key in $\mathcal{P}_{\text{sig}}^{js}$ 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.

E.3 Joint-State Composition for Multiple Protocols

So far, most of the literature has considered only the above type of joint-state where a single instance of *one* machine realizes multiple instances of *one* machine.

Our framework, however, also supports various other types of joint-state composition via the standard concurrent composition theorem (cf. Corollary 1). 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* an 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. In some cases, 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 Corollary 1: 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 $\mathcal{F}_{\text{sig}}$). More specifically, 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 1) 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).

E.4 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 §5).

Just as for joint-state, most universal composability models had to introduce additional extensions to model so-called global state [5, 12, 15]. 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 §3.4); to model a global CRS, one considers the combined protocol $(\texttt{somePublicRole}, \texttt{retrieveCRS} \mid \texttt{somePrivateRole})$ instead (which can also be written as $(\mathcal{P}, \mathcal{F}_{\mathsf{CRS}})$).

In general, we say that a protocol $\mathcal{P}$ has global state if (one or more) subroutine roles of $\mathcal{P}$ are public and hence allow the environment to access state stored in those subroutines. Just as for protocols with joint-state (cf. Appendix E.2), a protocol with global state is again a special case of the very general protocol definition of iUC, unlike in many other models. Hence, once we have shown that a protocol $\mathcal{P}$ with global state realizes some other protocol $\mathcal{F}$, we can directly apply the main composition theorem (Corollary 1) as follows, where clearly this corollary applies no matter which inner structure $\mathcal{P}$ has, and in fact, no matter whether it is a protocol with or without global state:

Corollary 5 (Concurrent composition with global state). *Let $\mathcal{P}$ and $\mathcal{F}$ be two 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 environmentally bounded, then*

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

Proof. This is a mere special case of Corollary 1 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 very user friendly.

We 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 §5 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.5 Global and Local Session IDs

As already explained in §5.3, most universal composability 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 [21] 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 §5. 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".

E.6 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 her own mapping between instances and entities using the **CheckID** algorithm (as also illustrated in §E.1). 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 classical 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 $\mathcal{F}_{\text{CA}}$ and $\mathcal{F}_{\text{KE}}$ in our case study in §5 for examples. 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.

E.7 Corruption Model

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 specifying one or more of the four corruption related algorithms in an appropriate way. 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 §3.2). 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 and reasonable protocol. 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.

E.8 Responsive Environments

In the specifications of both real protocols and ideal functionalities, it is often required for the adversary (and the environment) to provide some meta-information via the network interface to the protocol or the functionality, such as cryptographic algorithms, cryptographic values of signatures, ciphertexts, and keys, or corruption-related messages. Conversely, protocols/functionalities often have to provide the adversary with meta-information, for example, signaling information (e.g., the existence of machines) or again corruption-related messages. Such meta-information does not correspond to any real network messages, but is merely used for modeling purposes. Typically, giving the adversary/environment the option to not respond immediately to such modeling-specific messages does not translate to any real attack scenario. Hence, often it is natural for protocol designers to expect that the adversary/environment (answers and) returns control back to the protocol/functionality immediately when the adversary is requested to provide meta-information or when the adversary receives meta-information from the protocol/functionality.

As discussed in detail in [1], without responsive environments protocol designers have to manually deal with various delicate problems caused by non-immediate responses: i) While waiting for a response to a meta-information request, a protocol/ideal functionality might receive other requests, and hence, protocol designers have to take care of interleaving and dangling requests. ii) While a protocol/ideal functionality is waiting for an answer from the adversary to a meta-information request, other parties and parts of the protocol/ideal functionality can be activated in the meantime (via the network or the I/O interface), which might change their state, even their corruption status, and which in turn might lead to race conditions.

This is difficult to deal with and results in unnecessarily complex and artificial specifications of protocols and ideal functionalities, which, in addition, are then hard to re-use. In some cases, one might not even be able to express certain desired properties. As discussed in [1], there is no generic and generally applicable way to deal with non-responsiveness environments, and hence, one has to resort to solutions specifically tailored to the protocols at hand. This problem also propagates to all higher-level protocols as they might not get responses from their subprotocols as expected. The security proofs become more complex because one, again, has to deal with runs having various dangling and interleaving requests as well as unexpected and unintuitive state changes, which do not translate into anything in the real world, but are just an artifact of the modeling. In fact, since these artifacts are so unnatural, many protocol designers in the literature seem to implicitly assume that meta-information requests *are* answered immediately. For models without responsive environments, this has lead, besides others, to underspecified protocols and flawed security proofs (as detailed in [1]).

The iUC framework inherits the feature of responsive environments from the IITM model and thus gets rid of all of the above issues completely. This feature, as explained in §2, allows a protocol designer to mark meta-messages on the network as *restricting*, forcing the adversary/environment to indeed answer them immediately. Here, immediately means that the adversary may not interact with the protocol in any way until he has provided the expected response to the requesting entity. This drastically simplifies protocol specifications as well as security proofs since protocol designers do not have to manually deal with dangling requests and edge cases that can arise from non-immediate answers (and that do not relate to real attack scenarios anyway). Responsive environments also make protocol specifications more expressive as they allow protocol designers to capture the property of immediate responses of subroutines modeling local computations (such as $\mathcal{F}_{\text{sig-CA}}$ given in Figure 7, which provides immediate responses to `Sign` requests even when it has not been initialized yet).

E.9 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/session 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 in the classical sense.
(iii) All communication is over authenticated channels.
(iv) The corruption model is fixed for ideal protocols in the ideal world.
(v) Machines do not have any subroutines except for possibly (multiple instances/sessions of) a single ideal functionality. That is, all program logic is contained in a single machine.

Since some of those changes have to break out of the UC model, in particular the first two, 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 $[1, \ldots, n]$.

(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 ideal 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 somewhat artifical UC protocols exists).

F Single Session Analysis

This section illustrates how a single-session security analysis, which was explained on a high-level in §E.1, can be performed in iUC. As part of this, we provide one concrete specification for protocols with disjoint sessions (others are possible), which, thus, can be analyzed by considering just a single-session.

We start by defining (disjoint) protocol sessions 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 independent of 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* (which is the fixed way to deal with sessions in most other model, including in UC and GNUC), a protocol session can also 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. These notions are derived from to the session identifier functions and σ-session versions in the IITM model.

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 $\mathcal{E} \in \mathsf{Env}(\mathcal{P})$ 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 which, intuitively, is allowed to call at most a single session of $\mathcal{P}$. Again, the concept of such single-session environments is derived from a similar concept in the IITM model. More formally:

Definition 4. *Let $\mathcal{E} \in \mathsf{Env}(\mathcal{P})$ and let σ be a PSID function. We say that $\mathcal{E}$ is a* σ-single-session environment *if the following holds true for all systems $\mathcal{Q}$ that can connect to $\mathcal{E}$ and in all runs of the combined system $\{\mathcal{E}, \mathcal{Q}\}$:*

Let m be the first message that $\mathcal{E}$ sends on one of its external connections (connecting to $\mathcal{Q}$). Then m is sent to an entity $(pid, sid, role)$ such that $\sigma(pid, sid, role) \neq \bot$. Furthermore, every following message m' on an external connection of $\mathcal{E}$ is addressed to an entity $(pid_1, sid_1, role_1)$ such that $\sigma(pid, sid, role) = \sigma(pid_1, sid_1, role_1)$.

We denote the set of all σ-single-session (responsive and universally bounded) environments for a protocol $\mathcal{P}$ by $\mathsf{Env}_{\sigma\text{-single}}(\mathcal{P})$.

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 3 in §E). Both the realization relation and the composition theorem are natural translations of the corresponding statements in the IITM model.

Definition 5 (Single session realization relation). *Let σ be a PSID function, and let $\mathcal{P}$ and $\mathcal{F}$ be two 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 $\mathcal{S} \in \mathsf{Adv}(\mathcal{F})$ such that $\{\mathcal{E}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$ for all $\mathcal{E} \in \mathsf{Env}_{\sigma\text{-single}}(\mathcal{P})$.*[26]

Corollary 6 (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 [1]. However, unlike Corollary 1, 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 J.

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:

[26] 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.

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 no 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, initialization, and core logic algorithms:
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. §I.2.3; note that protocol designers using our syntax from Appendix C need not care about headers as they are automatically added). 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 on the receiver).

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

Fig. 15: 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 16 for how subroutine protocols $\mathcal{S}_i$ are defined.

(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 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 C, 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 15 and 16.

We directly obtain that a protocol $\mathcal{R}$ that is constructed as described above (cf. Figures 15, 16) is a σ-session protocol. Thus we can use Corollary 6 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, in the above theorem, 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. For example, consider a protocol $\mathcal{P}$ that uses a subroutine $\mathcal{S}_1$ that provides common reference strings (CRSs), where a different independent CRS is provided for different sessions of $\mathcal{P}$. We can make the subroutine

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 $sid^{accepted}_{prefix}$ be the prefix of the SID of the first entity that was accepted.
Output `accept` if and only if $sid^{accepted}_{prefix} = sid_{prefix}$.
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 $sid_{receiver}$ 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, initialization, and core logic algorithms:
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. §I.2.3; note that protocol designers using our syntax from Appendix C need not care about headers as they are automatically added). 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. 16: 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 15 for how the highest-level protocol $\mathcal{P}$ is defined.

$\mathcal{S}_1$ public in the combined protocol $\mathcal{R}$, modeling globally available CRSs that can also be used by other protocols independently of $\mathcal{P}$. Even in this setting with global state, we can still apply Corollary 6 to be able to analyze just a single session of $\mathcal{R}$, i.e., one session of $\mathcal{P}$ using a single CRS provided by $\mathcal{S}_1$.

G Example: Joint-State Realization

This section illustrates how one can use iUC to model and analyze joint-state realizations (see Appendix E.2 and Appendix E.3 for a general explanation of joint-state composition). We do so by means of an example: We consider an ideal signature functionality $\mathcal{F}_{\text{sig}}$ with disjoint sessions, i.e., where each session (defined via the SID of entities) uses an independent key pair. We want to realize this functionality with a joint-state realization $\mathcal{P}^{js}_{\text{sig}}$ where the same key pair is re-used across multiple session.

The ideal signature functionality $\mathcal{F}_{\text{sig}}$ is given in Figure 17. It is analogous to existing signature functionalities from the literature (e.g., [20]) and almost identical to the functionality $\mathcal{F}_{\text{sig-CA}}$ from our case study (cf. §5 and Figure 7), except that it does not register verification keys in an ideal $\mathcal{F}_{\text{CA}}$ subroutine. SIDs in $\mathcal{F}_{\text{sig}}$ are of the form $(pid_{\text{owner}}, sid')$, where pid_{owner} is the owner of a key and sid' denotes a specific key pair of pid_{owner}. In particular, for each sid' there is a separate independent key pair.

In our joint-state realization $\mathcal{P}^{js}_{\text{sig}}$ of $\mathcal{F}_{\text{sig}}$, we want to re-use the same signing key for all sessions of a key owner, i.e., every party pid_{owner} only has a single signing key for all sid'. We achieve this as follows: $\mathcal{P}^{js}_{\text{sig}}$ (given in Figures 18 and 19) implements the `signer` and `verifier` roles, which act as multiplexers for a

subroutine $\mathcal{F}_{\text{sig}}$. For each party *pid*, there is one session of the subroutine $\mathcal{F}_{\text{sig}}$ with fixed SID (pid, ϵ). We define the **CheckID** algorithm of the `signer` and `verifier` machines to accept all entities belonging to the same party, i.e., an instance of the `signer`/`verifier` machines models one party in all sessions (as is common for typical joint-state realizations). If a request for signing a message m in session/with key (pid, sid') is received, then $\mathcal{P}^{js}_{\text{sig}}$ instead prefixes m with sid' and uses the subroutine $\mathcal{F}_{\text{sig}}$ in session (pid, ϵ) to sign (sid', m); the response is then returned. Prefixing messages with sid' ensures that, even though the same signing key is used, signatures on messages for different SIDs sid' are still independent, just as in the ideal world for $\mathcal{F}_{\text{sig}}$. This is a standard technique often employed for obtaining joint-state realizations; in fact, our realization $\mathcal{P}^{js}_{\text{sig}}$ is based on the joint-state realization for digital signatures proposed in [20]. We obtain the following result for $\mathcal{P}^{js}_{\text{sig}}$:

Lemma 2. *Let $\mathcal{F}_{\text{sig}}$ and $\mathcal{P}^{js}_{\text{sig}}$ be as above. Then:*

$$(\mathcal{P}^{js}_{\text{sig}} \mid \mathcal{F}_{\text{sig}}) \leq (\mathcal{F}_{\text{sig}}) \ .$$

Proof. Analogous to [20].

Hence, we can replace $\mathcal{F}_{\text{sig}}$ used by some protocol $\mathcal{Q}$ (which might have disjoint sessions, each of them using separate signature key pairs) with its joint-state realization $\mathcal{P}^{js}_{\text{sig}}$ via our general composition theorem:

Corollary 7. *Let $\mathcal{Q}$ be a protocol such that $\mathcal{Q}$ and $\mathcal{F}_{\text{sig}}$ are connectable. Let $\mathcal{R} \in \mathsf{Comb}(\mathcal{Q}, (\mathcal{P}^{\mathsf{js}}_{\text{sig}} \mid \mathcal{F}_{\text{sig}}))$ and let $\mathcal{I} \in \mathsf{Comb}(\mathcal{Q}, \mathcal{F}_{\text{sig}})$ such that $\mathcal{R}$ and $\mathcal{I}$ have the same sets of public roles. If $\mathcal{R}$ is environmentally bounded, then*

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

Proof. This is an application of Corollary 1.

Let us highlight a few crucial features of iUC that we used above to model and compose the joint-state realization $\mathcal{P}^{js}_{\text{sig}}$:

- The flexible adressing mechanism **CheckID**, which allows for dynamically mapping entities to machine instances, allows us to easily create machine instances that manage/model a party in all sessions, as done for $\mathcal{P}^{js}_{\text{sig}}$. In particular, one can then share the same state for that party throughout all sessions.
- Our framework does not impose any restrictions on how different machines are connected and how entities of different machines are allowed to communicate. Hence, entities in $\mathcal{P}^{js}_{\text{sig}}$ that have different SIDs (but belong to the same party *pid*) can directly access the same signer entity (with fixed SID (pid, ϵ)) in the subroutine $\mathcal{F}_{\text{sig}}$. Similar to the previous point, this feature allows for easily sharing state between multiple sessions.
- The main composition theorem is independent of (the interpretation of) SIDs and in particular does not require protocols to have disjoint session. Instead, the theorem supports a general protocol definition, where both protocols with and without joint-state (such as $\mathcal{P}^{js}_{\text{sig}}$ and $\mathcal{F}_{\text{sig}}$) are mere special case. Hence, we can directly apply the theorem also to such a joint-state realization.
- Our corruption model allows the simulator in the ideal world to corrupt fresh entities as soon as they are created. As already explained in detail in Appendix E.2, this is crucial to be able to prove Lemma 2: Once the single key of a party *pid* in the real world has been corrupted, this corresponds to infinitely many corrupted sessions of $\mathcal{F}_{\text{sig}}$ in the ideal world. Hence, the simulator needs a way to perform those corruptions even though he has only polynomial runtime. iUC addresses this tension by providing a reactive mechanism, i.e., corruption can occur as soon as an entity is first triggered.

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

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): $\{\texttt{signer}, \texttt{verifier}\}$
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 $sid = (pid', sid')$:
If this check fails, output `reject`.
Otherwise, accept all entities with the same SID. {*A single instance manages all parties and roles in a single session.*
Corruption behavior:
– **LeakedData**$(pid, sid, role)$: If $(pid, sid, role)$ determines its initial corruption status, use the default behavior of **LeakedData**.
Otherwise, if $role = \texttt{signer}$ and $pid = \mathsf{pidowner}$, return $\mathsf{KeysGenerated}$. In all other cases return $\bot$.
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** $(\mathsf{pidowner}, \_, \texttt{signer})$: {*Successful initialization. Note that* `signer` *can submit* `InitSign` *multiple times, always with the same effect.*
$\mathsf{KeysGenerated} \leftarrow \texttt{ready}$.
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 $pk = \mathsf{pk} \wedge b = \texttt{true} \wedge msg \notin \mathsf{msglist} \wedge (\mathsf{pidowner}, \mathsf{sid}_{\mathsf{cur}}, \texttt{signer}) \notin \mathsf{CorruptionSet}$:
reply $(\texttt{VerResult}, \texttt{false})$. {*Prevent forgery.*
else:
reply $(\texttt{VerResult}, b)$. {*Return verification result.*

Fig. 17: The ideal signature functionality $\mathcal{F}_{\text{sig}}$.

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

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 $M_{\texttt{signer}}$:

Implemented role(s): $\texttt{signer}$
Subroutines: $\mathcal{F}_{\text{sig}} : \texttt{signer}$
Internal state:
- $\mathsf{initedSids} \subseteq \{0,1\}^* \cup \{\bot\} = \emptyset$. { *Which sessions have already sent an* `InitSign` *request?*

CheckID$(pid, sid, role)$**:**
Check that $sid = (pid', sid')$; otherwise output `reject`.
Accept all entities with the same PID. { *Accept one party in all sessions.*

Corruption behavior:
- **DetermineCorrStatus**$(pid, sid, role)$: Output **corr**$(pid, (pid, \epsilon), \mathcal{F}_{\text{sig}} : \texttt{signer})$.
- **AllowCorruption**$(pid, sid, role)$: Output **corr**$(pid, (pid, \epsilon), \mathcal{F}_{\text{sig}} : \texttt{signer})$.
  { *Allow corruption only if the corresponding subroutine entity of $\mathcal{F}_{\text{sig}}$ has been corrupted.*

MessagePreprocessing:

recv m **from** `I/O`:
  Try to parse $\mathsf{sid}_{\mathsf{cur}}$ as $(\mathsf{pid}_{\mathsf{cur}}, sid')$.
  if parsing fails:
    abort.

{ *Signers can perform actions only in sessions where they own the key, i.e., where their PID is a prefix of the SID. Otherwise, do nothing.*

Main:

{ *Due to* **MessagePreprocessing** *we have that, for I/O requests, the current entity is of the form* $(pid, (pid, sid'), \texttt{signer})$.

recv `InitSign` **from** `I/O`:
  send `InitSign` **to** $(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{pid}_{\mathsf{cur}}, \epsilon), \mathcal{F}_{\text{sig}} : \texttt{signer})$;
  wait for $(\texttt{InitSign}, \texttt{success}, pk)$.
  Add $\mathsf{sid}_{\mathsf{cur}}$ to $\mathsf{initedSids}$.
  reply $(\texttt{InitSign}, \texttt{success}, pk)$.

recv $(\texttt{Sign}, msg)$ **from** `I/O` **s.t.** $\mathsf{sid}_{\mathsf{cur}} \in \mathsf{initedSids}$:
  send $(\texttt{Sign}, (\mathsf{sid}_{\mathsf{cur}}, msg))$ **to** $(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{pid}_{\mathsf{cur}}, \epsilon), \mathcal{F}_{\text{sig}} : \texttt{signer})$;
  { *Prefix message with the current SID to obtain disjoint message spaces for different sessions.*
  wait for $(\texttt{Signature}, \sigma)$.
  reply $(\texttt{Signature}, \sigma)$.

Fig. 18: The joint-state realization $\mathcal{P}_{\text{sig}}^{js}$ of $\mathcal{F}_{\text{sig}}$ (Part 1). It uses a single instance of $\mathcal{F}_{\text{sig}}$ as a subroutine to realize multiple instances (in different sessions) of $\mathcal{F}_{\text{sig}}$.

H 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 §2. We note that the level of detail given in §2 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 I and proving the unbounded self-composition theorem in Appendix J. Full technical details of the IITM model with responsive environments are available in [1].

Named input and output tapes. In §2 we presented a slightly abstracted form of named tapes: Recall from §2 that ITMs have named tapes and two ITMs (in a system of ITMs) are connected if they each have one tape with the same name. Instances of such ITMs can then send messages to each other due to the connection provided by the named tapes, i.e., the connected tapes allow for bidirectional communication. Such connections of tapes are required to be unique within a system, i.e., for each tape name there are at most two ITMs that have a tape with that name.

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

Implemented role(s): `verifier`
Subroutines: $\mathcal{F}_{\text{sig}}$
CheckID$(pid, sid, role)$**:**
Check that $sid = (pid', sid')$; otherwise output `reject`.
Accept all entities with the same PID. {*Accept one party in all sessions.*
Corruption behavior:
– **DetermineCorrStatus**$(pid, sid, role)$: Parse sid as (pid_{signer}, sid').
Output **corr**$(pid_{signer}, (pid_{signer}, \epsilon), \mathcal{F}_{\text{sig}} : \texttt{signer})$
$\vee$ **corr**$(pid, (pid_{signer}, \epsilon), \mathcal{F}_{\text{sig}} : \texttt{verifier})$.
{*If either of these two entities has been corrupted, then the adversary can forge messages.*
– **AllowAdvMessage**$(pid, sid, role, pid_{\text{receiver}}, sid_{\text{receiver}}, role_{receiver}, m)$:
Check that $pid = pid_{\text{receiver}}$ and $role_{receiver} \neq \mathcal{F}_{\text{sig}} : \texttt{signer}$.
If all checks succeed, output `true`, otherwise output `false`
Main:

recv $(\texttt{Verify}, msg, \sigma, pk)$ **from** `I/O`:
Parse $\mathsf{sid}_{\mathsf{cur}}$ as (pid_{signer}, sid').
send $(\texttt{Verify}, (\mathsf{sid}_{\mathsf{cur}}, msg), \sigma, pk)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, (pid_{signer}, \epsilon), \mathcal{F}_{\text{sig}} : \texttt{signer})$;
{*Prefix message with the current SID to obtain disjoint message spaces for different sessions.*
wait for $(\texttt{VerResult}, b)$.
reply $(\texttt{VerResult}, b)$.

Fig. 19: The joint-state realization $\mathcal{P}^{js}_{\text{sig}}$ of $\mathcal{F}_{\text{sig}}$ (Part 2). It uses a single instance of $\mathcal{F}_{\text{sig}}$ as a subroutine to realize multiple instances (in different sessions) of $\mathcal{F}_{\text{sig}}$.

Formally, in the IITM model machines (ITMs) may have tapes with names (*named tapes*). A named tape belongs to one machine. Such a tape is either an *input tape* or an *output tape*. A machine can have several input and output tapes, where all of them have different names. Also, systems of ITMs are defined in such a way that there are no two machines with an input tape with the same name. The same is true for output tapes. In a run of a system $\mathcal{Q}$, an (instance of an) ITM M with an output tape named n can write a message m to that tape; if there is a machine M' in $\mathcal{Q}$ with an input tape named n, then the message m is written to an input tape with name n of (one of the instances of) M'.[27] We also say that M and M' or their named tapes *are connected* (while formally these are still different tapes). Otherwise, if there is no machine with an input tape named n in $\mathcal{Q}$, the message m is discarded and the master ITM is activated instead.

In this work, we consider a special case of systems of ITMs, where machines are always connected via pairs of tapes, thus allowing for bidirectional communication. That is, if a machine M has an input tape named n and there is a machine M' in the system with an output tape named n, then M also has an output tape named $\hat{n}$, for some name $\hat{n}$, and M' has an input tape named $\hat{n}$. Hence, (instances of) M and M' can both send and receive messages to/from each other.

This motivated the slightly more abstracted presentation of tapes in §2: When we say that an ITM M has a *tape* with name n, we actually mean that it has a pair of input and output tapes named n and $\hat{n}$. When we say that another ITM M' also has a tape named n, then we mean that it has a pair of input and output tapes named $\hat{n}$ and n, i.e., M' and M can communicate in both directions via the pairs of named tapes. For simplicity of presentation, in the following we will often say *(named) bidirectional tape* or simply *(named) tape* when we actually mean a pair of input and output tapes.

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 [1] 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

[27] As explained in §2, the **CheckAddress** mode is used to determine which instance of M' gets to receive and process the incoming message.

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 [1], 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 §2 (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 [1, 22], we have to consider protocol systems where instances of machines in these systems have protocol session IDs (PSIDs);[28] 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 machines.

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 emphasize that, similarly to Definition 4, a single-session environment must have this property in every context it runs (not only when interacting with $\mathcal{P}$); a formal definition is available in [1]. We denote the set of single-session (responsive and universally bounded) environments for a system $\mathcal{Q}$ by $\mathsf{Env}_{\sigma\text{-single}}(\mathcal{Q})$.

We say that $\mathcal{P}$ *single-session realizes* $\mathcal{F}$ ($\mathcal{P} \leq_{\sigma\text{-single}} \mathcal{F}$) if there exists a simulator $\mathcal{S} \in \mathsf{Adv}(\mathcal{F})$ such that $\{\mathcal{E}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$ for all $\mathcal{E} \in \mathsf{Env}_{\sigma\text{-single}}(\mathcal{P})$. 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 3 (Unbounded self-composition [1]). *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 $\mathcal{P} \leq \mathcal{F}$.*

I Formal Mapping of Protocols to ITMs

In this section, we explain how the templates specified in §3 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 §2, §H, and [1]). Hence, all theorems of the responsive

[28] 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 F.

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 I.1. Then, in Appendix I.2, we explain how a single instance of our template from §3.3 is mapped to a system of ITMs, including the exact specification of those ITMs. Finally, in Appendix I.3 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).

I.1 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 §2. 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 C. 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 §2 and formally defined in Appendix H, 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:

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

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 [1], 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.

I.2 Mapping Templates/Machines

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 and machines therein are 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 §I.3 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 §2 and §3, this section provides full details.

I.2.1 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 §3 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 template is 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.

I.2.2 Tapes Recall from §2 and Appendix H that a machine M has named tapes for (bidirectional) communication with other machines. An instance of a machine M_1 can write a message on a tape t named n; if there is another machine M_2 with a tape t' also named n, then the message is delivered to (an instance of) M_2 on tape t'. In this case, we say that M_1 and M_2 (respectively their tapes t and t') are connected. Within a system of ITMs, it is required that tapes connect uniquely, i.e., for each name there are at most two machines with tapes of that name. Also recall that tapes are grouped into network and I/O tapes.

On a high level, we have to use tapes to represent the abstract connections from the iUC framework. That is, we have to represent network connections each role has to the adversary, (internal) I/O connections between various roles, including subroutine roles, in a protocol, and a set of connections from each (public) role to the environment that can be used to simulate higher-level protocols. To give an intuition right away, let us look at an example.

Example 1. In Figure 3a on Page 11 each of the arrows between the box labeled "I/O" (representing an environment) and the protocol will be represented via a parameterized set of I/O tapes. Each of the internal arrows between roles of the protocol will be represented by a single unique (bidirectional) tape. In addition (not shown in the Figure) each role will have one (bidirectional) network tape for connecting to the adversary.

In the following, we formalize the tapes that a single machine M offers for others to connect to. Later, in Appendix I.3, we detail how multiple machines are connected in the context of a protocol, where some (roles of a) machine are public and others are private.

Let M be the machine of $\mathcal{P}$ from above. For each role $role_i$ implemented by M, there is one network tape that is used to connect the adversary, allowing $role_i$ to send an receive messages from the network. Furthermore, for each role $role_i$ and for each subroutine role $subrole_j$ of M, there is a I/O tape that is used to connect $role_i$ to $subrole_j$, allowing them to directly send and receive messages to each other. Finally, each role $role_i$ also has a parameterized number of I/O tapes that allow other (unknown) higher-level protocols as well as the environment to connect and send direct messages to $role_i$. All tapes of M are uniquely named (the exact names can be chosen arbitrarily and do not matter for the purpose of this mapping. Later on, in Appendix I.3, the names of I/O tapes will be chosen such that tapes connect to subroutine roles and higher-level protocols/the environment as expected).

Jumping ahead, the exact parameterized number of I/O tapes for a role will be fixed depending on whether the role is public in the context of a protocol $\mathcal{Q}$ (cf. Appendix I.3): If $role_i$ is private, then it offers exactly as many I/O tapes as are used by other roles of $\mathcal{Q}$ that want to connect to $role_i$ as a subroutine. For public roles, the number of I/O tapes is arbitrarily large but fixed, allowing the environment to connect to an arbitrary number of I/O tapes and use them for simulating higher-level protocols.

Example 2 (Example 1 continued.). In Figure 3a the `signer` role is private (in the context of the combined protocol) and hence offers exactly two I/O tapes for the `initiator` and `responder` roles to connect to (represented by the two arrows between those roles). In contrast, the `initiator` role is public and hence offers an arbitrary number of I/O tapes for the environment to connect to (these are represented by the single arrow pointing towards the box labeled "I/O").

I.2.3 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). More specifically, a message 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. The expected format for messages m received and sent on network tapes is shortened to be $m = ((pid_{rcv}, sid_{rcv}), m')$ and $m = ((pid_{snd}, sid_{snd}), m')$ respectively, as one of the communication partners is always the network, i.e., there is no second entity involved in the communication. Messages that do not adhere to this format are automatically dropped by the **CheckAddress** mode (cf. Appendix I.2.4).

Now, suppose M receives a message $m = ((pid_{snd}, sid_{snd}), (pid_{rcv}, sid_{rcv}), m')$ on one of its I/O tapes t belonging to role $role_i$. The sending and receiving entities are computed from the message m and the tape t as follows. The receiving entity is set to $(pid_{rcv}, sid_{rcv}, role_i)$. For the sending entity, there are two cases to distinguish: if t is a tape connecting $role_i$ to one of its known subroutine roles $subrole_j$, then the sending entity is set to be $(pid_{snd}, sid_{snd}, subrole_j)$. Otherwise, t is one of the parameterized many I/O tapes of $role_i$. In that case, the sending entity is set to be $(pid_{snd}, sid_{snd}, l)$ where $l \in \mathbb{N}$ is a number that uniquely identifies the tape t (i.e., all of the parameterized many tapes are numbered consecutively). Note that, while technically speaking l is not the name of a role, it still identifies a unique name of a role that is connected to tape t (cf. paragraph "exchanging messages" in §3.1).[29]

Conversely, suppose an entity $(pid_{snd}, sid_{snd}, role_i)$ managed by an instance of M wants to send a message body m' to some other entity $(pid_{rcv}, sid_{rcv}, subrole_j)$ or $(pid_{rcv}, sid_{rcv}, l)$, $l \in \mathbb{N}$. The resulting message m, including the header, and the tape t that is used for sending are computed as follows. The message m is set to be $((pid_{snd}, sid_{snd}), (pid_{rcv}, sid_{rcv}), m')$. The tape t is either the I/O tape connecting $role_i$ to its subroutine role $subrole_j$, or the parameterized I/O tape connecting to a higher-level protocol which is identified by the number l.

Sending and receiving entities are computed analogously for messages that are sent/received on a network tape. We note that protocol designers using our syntax from Appendix C 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.

I.2.4 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 20: 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/or intended receiver of a message. Otherwise, the instance determines the intended receiving entity $(pid_{rcv}, sid_{rcv}, role_{rcv})$ as described in §I.2.3 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.

[29] In other words, M is not aware of the precise name of the role as the connecting higher-level protocol is unknown/only simulated by the environment. In particular, M cannot depend on this name in their code. Note that having the tuple $(pid_{snd}, sid_{snd}, l)$ is typically sufficient as it allows for uniquely addressing the sending entity, e.g., to return a message. A machine can simply send a message with sender (pid_{snd}, sid_{snd}) on the tape with number l.

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. §I.2.3*

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. §I.2.3*

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

Output *decision*.

Fig. 20: The **CheckAddress** mode of protocol machines in our framework.

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**.

I.2.5 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 21, 22, and 23. 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. If $m =$ `CorruptionStatus`? on an I/O tape, then the corruption status of $(pid_{rcv}, sid_{rcv}, role_{rcv})$ is determined and returned to the sender immediately. In particular, 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 a 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 a 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. We note that already in Step 4. the message to be processed might be a corruption request. In this case, in 4. the adversary is not asked whether or not he wants to corrupt the entity.
6. If the instance was activated by the initial message $m =$ `InitEntity` on an I/O tape, then the instance reports a successful initialization by sending a message to the sender *sender*. This ends the activation and none of the following steps are performed. We note that, by default, a sender of a `InitEntity` request receives a response even if the adversary decides to corrupt $(pid_{rcv}, sid_{rcv}, role_{rcv})$ in Step 4.
7. If $(pid_{rcv}, sid_{rcv}, role_{rcv})$ is explicitly corrupted by the adversary, then messages are forwarded to/from the network.
8. Otherwise, if the receiver $(pid_{rcv}, sid_{rcv}, role_{rcv})$ is honest, then first the algorithm **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.

State variable acceptedEntities $\subseteq (\{0,1\}^*)^3 = \emptyset$. {*List of entities that have been accepted so far. Mainly for use in the* **CheckID** *algorithm.*

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, append it to the list acceptedEntities if it is not yet included, 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. §I.2.2 and §I.2.3.*

if Corruption model $\neq$ `custom` $\wedge\ m' =$ `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 $\mathsf{entity}_{\mathsf{cur}} \notin$ entityInitDone:
  Add $\mathsf{entity}_{\mathsf{cur}}$ to entityInitDone.
  Run **EntityInitialization**$(\mathsf{entity}_{\mathsf{cur}})$.

Continue in Figure 22.

Fig. 21: The **Compute** mode of protocol machines (part 1).

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 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.[30] 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. §I.2.3) including a header that specifies sender and receiver. Note that this is automatically taken care of if our convenient syntax from §C 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`

[30] This access is kept implicit in Figures 21 and 22 as we do not want to clutter the calls to the algorithms with several additional parameters.

if Corruption model $\neq$ `custom`:
  if $\mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}] = \bot$: {*Step 4.: Initialize corruption status of new entity.*

    if $t = \mathsf{t}_{\mathsf{net}} \wedge m' = (\mathtt{SetCorruptionStatus}, b) \wedge b \in \{\mathtt{true}, \mathtt{false}\}$: {*First message already sets corruption status.*

      if $b = \mathtt{true}$ and **Corruption model** allows corruption of the entity at this point:
        {*See §3.2 for when corruption of an entity is allowed.*
        $\mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}] \leftarrow$ **AllowCorruption**$(\mathsf{entity}_{\mathsf{cur}})$.
        {*Use* **AllowCorruption** *to decide whether corruption succeeds.*
      else:
        $\mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}] \leftarrow \mathtt{false}$.
      if $\mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}] = \mathtt{true}$: {*Return a response to the adversary. Note that this stops the activation, none of the following steps are performed.*
        $\mathit{leakage} \leftarrow$ **LeakedData**().
        send $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), (\mathtt{CorruptionStatus}, \mathtt{true}, \mathit{leakage}))$ **on** $\mathsf{t}_{\mathsf{net}}$.
      else:
        send $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), (\mathtt{CorruptionStatus}, \mathtt{false}, \bot))$ **on** $\mathsf{t}_{\mathsf{net}}$.
    else: {*For all other first messages.*
      send responsively $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), \mathtt{CorruptMe?})$ **on** $\mathsf{t}_{\mathsf{net}}$;
      wait for $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), (\mathtt{SetCorruptionStatus}, b))$ **on** $\mathsf{t}_{\mathsf{net}}$ **s.t.** $b \in \{\mathtt{true}, \mathtt{false}\}$.
      if $b = \mathtt{true}$ and **Corruption model** allows corruption of the entity at this point:
        {*See §3.2 for when corruption of an entity is allowed.*
        $\mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}] \leftarrow$ **AllowCorruption**$(\mathsf{entity}_{\mathsf{cur}})$.
        {*Use* **AllowCorruption** *to decide whether corruption succeeds.*
      else:
        $\mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}] \leftarrow \mathtt{false}$.
      if $\mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}] = \mathtt{true}$: {*Leak information and give control to either the adversary or, if this instance was triggered by an* `InitEntity` *command, to* $\mathsf{entity}_{\mathsf{call}}$.
        $\mathit{leakage} \leftarrow$ **LeakedData**().
        if t is an I/O tape and $m' = \mathtt{InitEntity}$:
          send responsively $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), (\mathtt{CorrStatusRestrict}, \mathtt{true}, \mathit{leakage}))$ **on** $\mathsf{t}_{\mathsf{net}}$;
          wait for $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), \mathtt{OK})$ **on** $\mathsf{t}_{\mathsf{net}}$.
          send $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), (\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}), \mathtt{InitEntityDone})$ **on** t.
        else:
          send $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), (\mathtt{CorruptionStatus}, \mathtt{true}, \mathit{leakage}))$ **on** $\mathsf{t}_{\mathsf{net}}$.

  else if $\mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}] = \mathtt{false}$: {*Step 5.: Process corruption requests for already existing entities.*

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

Continue in Figure 23.

Fig. 22: The **Compute** mode of protocol machines (part 2).

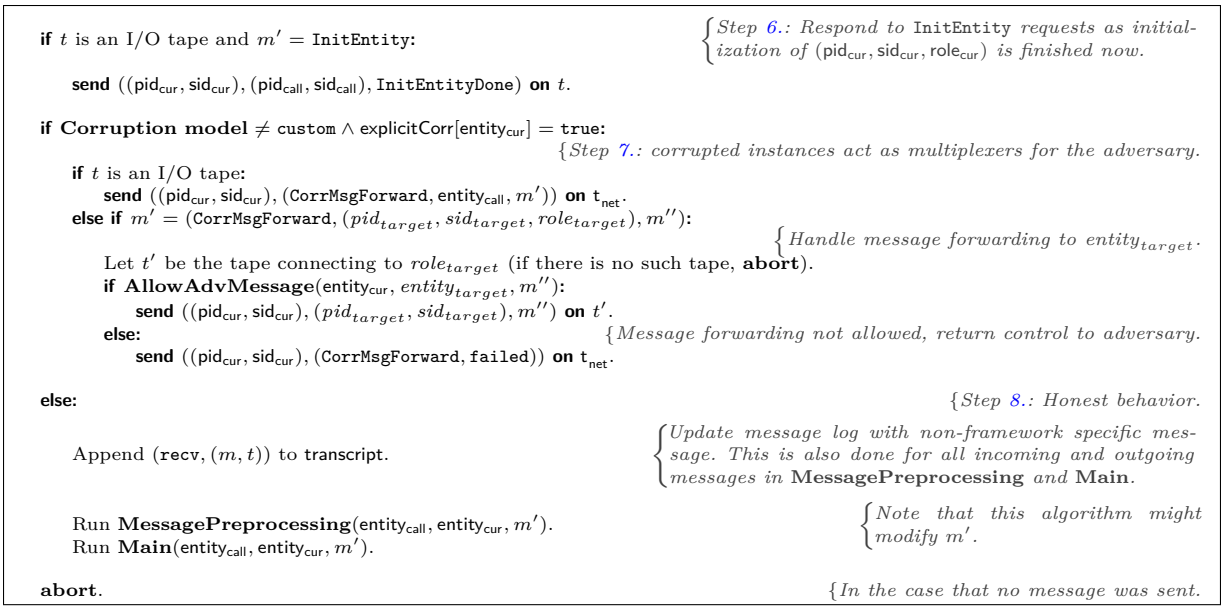


Fig. 23: The **Compute** mode of protocol machines (part 3).

- `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 core logic of a protocol machine is defined via the four algorithms **Initialization**, **EntityInitialization**, **MessagePreprocessing**, and **Main**. 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 §3.2 and §3.3, 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 7.) 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. takes care of all `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 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., 5., and 7., 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 an `InitEntity` request from the I/O interface (cf. Step 6.). 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 C). 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 7. in particular is not executed. Overall, this yields the desired behavior.

I.2.6 Mapping the syntax from §C Interpreting the send and receive commands presented in §C 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 §I.2.3). 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 §I.2.3 and m' is the message payload as specified in the **send responsively** command. By definition of the restriction (cf. §I.1), 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 message 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 four 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 §C: 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 §C, 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.

I.3 Mapping Protocols

Now that 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 each specified using the template. 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 $\mathcal{P}$) via suitable tape name choices (again, the exact names are arbitrary and not important for this mapping):

- Recall that $role_i$ has an I/O tape t for each of the subroutine roles $role_{sub}$ of M_j. Since $\mathcal{P}$ is complete, we have that $role_{sub}$ is implemented by one of the machines of $\mathcal{P}$. If (the machine implementing) $role_{sub}$ also has an I/O tape t' to connect to $role_i$ (because $role_{sub}$ specifies $role_i$ as a subroutine), then t and t' are connected and we require the (bidirectional) tapes t and t' to be identical since we need only one (bidirectional) tape for $role_i$ and $role_{sub}$ to communicate . 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 and the environment 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, all roles of $\mathcal{P}$ that use $role_i$ as a subroutine can actually connect to $role_i$; however, there are no additional (unconnected) tapes that would allow 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 $\mathcal{P}$ that use $role_i$ as a subroutine can connect to it. In other words, not only can roles of $\mathcal{P}$ connect to $role_i$ (if they use this role as subroutine), 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 environment/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. This was already illustrated by Example 1.

J Proof of the Unbounded Self-Composition Theorem

In this section we provide a proof for the unbounded self-composition theorem in iUC (cf. Corollary 6 in Appendix F).

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 3 in Appendix H) 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 H), show that $\mathcal{P}$ and $\mathcal{F}$ are $\widetilde{\sigma}$-session protocols in the sense of the IITM model (cf. Appendix H), and show that $\mathcal{P} \leq_{\widetilde{\sigma}\text{-single}} \mathcal{F}$ (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 I.2.3) 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 sending 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 sending 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 an arbitrary (not necessarily single-session) 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 message 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.[31] 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 violated 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 environmentally bounded as in particular $\widetilde{\sigma}$ is efficiently computable.

Now let $\mathcal{E} \in \mathsf{Env}^{\sim}_{\widetilde{\sigma}\text{-single}}(\mathcal{P})$ 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 $\mathcal{E} \in \mathsf{Env}_{\sigma\text{-single}}(\mathcal{P})$. 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 $\widetilde{\mathcal{P}} \leq^{\sim}_{\widetilde{\sigma}\text{-single}} \widetilde{\mathcal{F}}$.

We can now apply the unbounded self-composition theorem of the IITM model (cf. Theorem 3) 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 $\mathcal{P} \leq \mathcal{F}$. □

[31] For example, a machine M that is used only as an internal subroutine within a protocol might, upon being activated by a sender entity from a higher-level protocol/role of the same protocol, compute some response and return this response to the sending entity. In the context of the whole protocol, the sender entity is always of the same session as the receiver entity in M, and hence M would also return the message to an entity in the same session. However, when running in an arbitrary context that can claim arbitrary sender entities, the context might choose a sender entity that is from a different session than the receiver entity of M. Thus, M would return/send a message to an entity in a different session, violating property 3 of $\widetilde{\sigma}$-session protocols.