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.

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; 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 §3 and also illustrated by our case study (cf. §4), 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. §3, §4, and §4.3 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. §4.3). 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 §4). 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. §4.3). 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. §4.3). 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 §3 and illustrated by our case study in §4, 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 §4.3). A second illustrative example is given in the full version of this paper [2], 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: We describe the iUC framework in §2, with a discussion of the main concepts and features in §3. A case study further illustrates and highlights some features of iUC in §4. We conclude in §5. Full details are given in our full version [2].

2 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.

The following description of the iUC framework is kept independently of the IITM model, i.e., one can understand and use the iUC framework without knowing the IITM model. More details of the underlying IITM model are available in the full version [2]. Here we explain the IITM model not explicitly, but rather explain relevant parts as part of the description of the iUC framework. We start with some preliminaries in §2.1, mainly describing the general computational model, before we explain the general structure of protocols in iUC in §2.2, with corruption explained in §2.3. We then present our protocol template in §2.4. In §2.5, we explain how protocol specifications can be composed in iUC to create new, more complex protocol specification. Finally, in §2.6, we present the realization relation and the composition theorem of iUC. As mentioned, concrete examples are given in our case study (cf. §4). We provide a precise mapping from iUC protocols to the underlying IITM model in the full version, 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.

2.1 Preliminaries

Just as the IITM model, the iUC framework uses interactive Turing machines as its underlying computational model. Such interactive Turing machines can be connected to each other to be able to exchange messages. A set of machines $\mathcal{Q} = \{M_1, \ldots, M_k\}$ is called a *system*. In a run of $\mathcal{Q}$, there can be one or more instances (copies) of each machine in $\mathcal{Q}$. One instance can send messages to another instance. At any point in a run, only a single instance is active, namely, the one to receive the last message; all other instances wait for input. The active instance becomes inactive once it has sent a message; then the instance that receives the message becomes active instead and can perform arbitrary computations. The first machine to run is the so-called *master*. The master is also triggered if the last active machine did not output a message. In iUC, the environment (see next) will take the role of the master. Jumping ahead, in the iUC framework a special user-specified **CheckID** algorithm is used to determine which instance of a machine receives a message and whether a new instance is to be created (cf. §2.4).

To define the universal composability security experiment (cf. Figure 1 and §2.5), one distinguishes between three types of systems: protocols, environments, and adversaries. Intuitively, the security experiment in any universal composability model compares a protocol $\mathcal{P}$ with another protocol $\mathcal{F}$, where $\mathcal{F}$ is typically an ideal specification of some task, called *ideal protocol* or *ideal functionality*. The idea is that if one cannot distinguish $\mathcal{P}$ from $\mathcal{F}$, then $\mathcal{P}$ must be "as good as" $\mathcal{F}$. More specifically, the protocol $\mathcal{P}$ is considered secure (written $\mathcal{P} \leq \mathcal{F}$) if for all adversaries $\mathcal{A}$ controlling the network of $\mathcal{P}$ there exists an (ideal) adversary $\mathcal{S}$, called *simulator*, controlling the network of $\mathcal{F}$ such that $\{\mathcal{A}, \mathcal{P}\}$ and $\{\mathcal{S}, \mathcal{F}\}$ are indistinguishable for all environments $\mathcal{E}$. Indistinguishability means that the probability of the environment outputting 1 in runs of the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}\}$ is negligibly close to the probability of outputting 1 in runs of the system $\{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$ (written $\{\mathcal{E}, \mathcal{A}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$).

In the security experiment, systems are connected as follows (cf. arrows in Figure 1): Every (machine in a) protocol has an I/O interface that is used to connect to other protocol machines, higher-level protocols, or an environment, which, in turn, can simulate higher-level protocols. Every (machine in a) protocol also has a network interface to connect to a network adversary. We sometimes let the environment subsume the network adversary. That is, the environment performs both roles: on the left-hand side of Figure 1, instead of having the systems $\mathcal{E}$ and $\mathcal{A}$ we can have an environment $\mathcal{E}'$ that connects to both the I/O interface and the network interface of $\mathcal{P}$.

The iUC framework includes support for so-called responsive environments and responsive adversaries introduced in [1]. Such environments/adversaries can

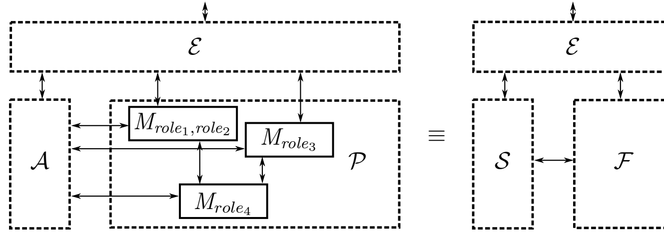


Fig. 1: The setup for the universal composability experiment ($\mathcal{P} \leq \mathcal{F}$) and internal structure of protocols. Here $\mathcal{E}$ is an environment, $\mathcal{A}$ and $\mathcal{S}$ are adversaries, and $\mathcal{P}$ and $\mathcal{F}$ are protocols. Arrows between systems denote connections/interfaces that allow for exchanging messages. The boxes M_i in $\mathcal{P}$ are different machines modeling various tasks in the protocol. Note that the machines in $\mathcal{P}$ and the way they are connected is just an example; other protocols can have a different internal structure.

be forced to answer certain messages on the network interface of the protocol immediately, without interfering with the protocol in between. These messages are called *restricting messages*. This mechanism is very handy to, e.g., exchange meta information such as the corruption state of a protocol participant or obtain cryptographic keys from the adversary; see our full version [2] and [1] for a more detailed discussion.

We require environments to be *universally bounded*, i.e., there is a fixed polynomial in the security parameter (and possibly external input) that upper bounds the runtime of an environment no matter to which protocol and adversary it is connected to. A system $\mathcal{Q}$ is called *environmentally bounded* if for every (universally bounded) environment $\mathcal{E}$ there is a polynomial that bounds the runtime of the system $\mathcal{Q}$ connected to $\mathcal{E}$ (except for potentially a negligible probability). This will mostly be required for protocols; note that natural protocols used in practice are typically environmentally bounded, including all protocols that run in polynomial time in their inputs received so far and the security parameter. This is the same runtime notion used in the IITM model. Compared to other models, this notion is very general and particularly simple (see [22] for a discussion).

We define $\mathsf{Env}(\mathcal{Q})$ to be the set of all universally bounded (responsive) environments that connect to a system $\mathcal{Q}$ via network and I/O interfaces. We further define $\mathsf{Adv}(\mathcal{P})$ to be the set of (responsive) adversaries that connect to the network interface of a protocol $\mathcal{P}$ such that the combined system $\{\mathcal{A}, \mathcal{P}\}$ is environmentally bounded.

2.2 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

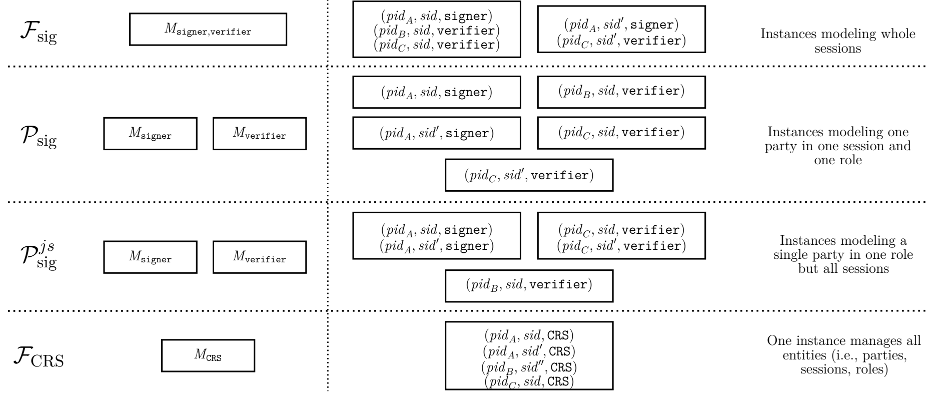


Fig. 2: Examples of static and dynamic structures of various protocol types. $\mathcal{F}_{\text{sig}}$ is an ideal protocol, $\mathcal{P}_{\text{sig}}$ a real protocol, $\mathcal{P}^{js}_{\text{sig}}$ a so-called joint-state realization, and $\mathcal{F}_{\text{CRS}}$ a global state protocol. On the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

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

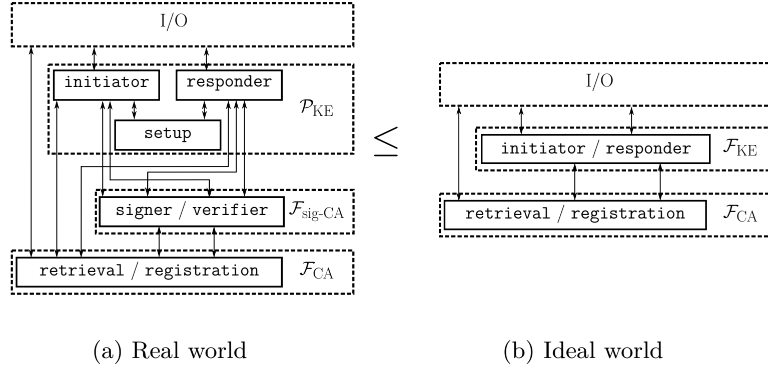
Roles: As already mentioned, a role is a piece of code that performs a specific task in a protocol $\mathcal{P}$. Every role in $\mathcal{P}$ is implemented by a single unique machine M_i, but one machine can implement more than one role. This is useful for sharing state between several roles: for example, consider an ideal functionality $\mathcal{F}_{\text{sig}}$ for digital signatures consisting of a `signer` and a `verifier` role. Such an ideal protocol usually stores all messages signed by the `signer` role in some global set that the `verifier` role can then use to prevent forgery. To share such a set between roles, both roles must run on the same (instance of a) machine, i.e., $\mathcal{F}_{\text{sig}}$ generally consists of a single machine $M_{\texttt{signer,verifier}}$ implementing both roles. In contrast, the real protocol $\mathcal{P}_{\text{sig}}$ uses two machines $M_{\texttt{signer}}$ and $M_{\texttt{verifier}}$ as those roles do not and cannot directly share state in a real implementation (cf. left-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. 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.

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 §2.5.) 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. §2.4).

Examples illustrating the above concepts: Figure 3a, which is further explained in our case study (cf. §4), 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



(a) Real world (b) Ideal world

Fig. 3: The static structures of the ideal key exchange functionality $\mathcal{F}_{\mathrm{KE}}$ (right side) and its realization $\mathcal{P}_{\mathrm{KE}}$ (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. §2.4).

the actual key exchange and one private role `setup` that generates some global system parameters. The protocol $\mathcal{P}_{\mathrm{KE}}$ 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 $\mathcal{F}_{\mathrm{CA}}$ 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 $\mathcal{F}_{\mathrm{CA}}$ 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 $\mathcal{F}_{\mathrm{CA}}$ remains public, modeling that public keys are generally considered to be known to everyone, so not only $\mathcal{P}_{\mathrm{KE}}$ but also the environment (and possibly other protocols later using $\mathcal{P}_{\mathrm{KE}}$) 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 $\mathcal{F}_{\mathrm{CA}}$, 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 ($\texttt{initiator}, \texttt{responder}, \texttt{retrieval} \mid \texttt{setup}, \texttt{signer}, \texttt{verifier}, \texttt{registration}$).

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 $M_{\texttt{signer},\texttt{verifier}}$, 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 §2.4) 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}[4] and either the actual role name $role_{snd}$ (if the sender is a known subroutine of the receiver, cf. §2.4) 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[5]). 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 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.

2.3 Modeling Corruption

We now explain on an abstract level how our framework models corruption of entities. In §2.4, 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.

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

[5] 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.

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 §2.4.

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 §2.4), it asks the adversary via a *restricting message*[6] whether he wants to corrupt $(pid, sid, role)$ before performing any other computations. The answer of the adversary is processed as discussed before, i.e., the entity decides whether to accept or reject a corruption request. This gives the adversary the power to corrupt new entities right from the start, if he desires; note that in the case of static corruption, this is also the last point in time where an adversary can explicitly corrupt $(pid, sid, role)$.

For modeling purposes, we allow other entities and the environment to obtain the current corruption status of an entity $(pid, sid, role)$.[7] This is done by sending

[6] Recall from §2.1 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]).

[7] 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

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.

2.4 Specifying Protocols

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

The template is given in Figure 4. Some parts are self-explanatory; the other parts are described in more detail in the following. The first section of the template specifies properties of the whole protocol that apply to all machines.

Participating roles*:* This list of sets of roles specifies which roles are (jointly) implemented by a machine. 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 §2.3: *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

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.

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. 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.

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.[8] 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

[8] 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.

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 the full version).

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`.

Whenever one (entity in one) instance of a machine, the adversary, or the environment sends a message m to some entity $(pid, sid, role)$ (via the entity's I/O interface or network interface), the following happens: m is delivered to the first instance of the machine, say M, that implements *role*, where instances of a machine are ordered by the time of their creation. That instance then runs **CheckID**$(pid, sid, role)$ to determine whether it manages $(pid, sid, role)$, and hence, whether the message m (that was sent to $(pid, sid, role)$) should be accepted. If **CheckID** accepts the entity, then the instance gets to process the message m; otherwise, it resets itself to the state before running **CheckID** and the message is given to the next instance of M (according to the order of instances mentioned before) which then runs **CheckID**$(pid, sid, role)$, and so on. If no instance accepts, or no instance exists yet, then a new one is created that also runs **CheckID**$(pid, sid, role)$. If that final instance accepts, it also gets to process m; otherwise, the new instance is deleted, the message m is dropped, and the environment is triggered (with a default trigger message).

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 list $\mathsf{acceptedEntities}$ 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**.

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 §2.3, with the formal definition of these algorithms, including their default behavior if not specified, given in the full version. 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 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,[9] the algorithms **MessagePreprocessing** and **Main** are run. The **MessagePreprocessing** algorithm is executed first. If it does not

[9] As mentioned in §2.3, if an entity is explicitly corrupted, it instead acts as a forwarder for messages to and from the adversary.

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 the full version; see our case study in §4 for examples.

This concludes the description of our template. As already mentioned, in the full version of this paper 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 §2.6).

2.5 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 §2.2 and our case study in §4). In any case, the notation from §2.2 of the form $(\mathit{role}_1^{pub} \ldots \mathit{role}_i^{pub} \mid \mathit{role}_1^{priv} \ldots \mathit{role}_j^{priv})$ should be used for this purpose.

For pairwise connectable protocols $\mathcal{Q}_1, \ldots, \mathcal{Q}_n$ we define $\mathsf{Comb}(\mathcal{Q}_1, \ldots, \mathcal{Q}_\mathsf{n})$ to be the (finite) set of all protocols $\mathcal{R}$ that can be obtained by connecting $\mathcal{Q}_1, \ldots, \mathcal{Q}_n$. 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 $(\mathcal{Q}_1, \ldots, \mathcal{Q}_i \mid \mathcal{Q}_{i+1}, \ldots, \mathcal{Q}_n)$ we denote the protocol $\mathcal{R} \in \mathsf{Comb}(\mathcal{Q}_1, \ldots, \mathcal{Q}_\mathsf{n})$, where the public roles of $\mathcal{Q}_1, \ldots, \mathcal{Q}_i$ remain public in $\mathcal{R}$ and all other roles are private. This notation can be mixed with the notation from §2.2 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.

2.6 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 §2.4, 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 $\{\mathcal{E}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$.*[10]

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:

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

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 (see the full version [2]).

3 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

[10] 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.

that are not supported by other (conventions of) models, including the prominent UC and GNUC models. Our case study in §4 further illustrates the expressiveness of iUC. An extended discussion of concepts and features is available in the full version [2]. 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.

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. §4) 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. 4.3). 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 our full version 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 §4 is an example of a protocol that uses and combines all of these protocol features, with a detailed explanation and discussion provided in §4.3.
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 §4. 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. §4.3).

Besides our case study in §4, the flexibility and usability of iUC is also illustrated by another example in the full version, 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.

4 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*

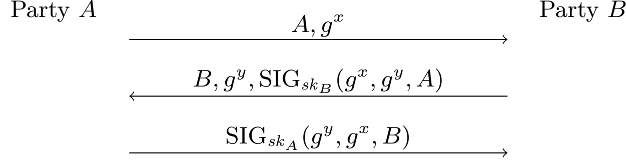


Fig. 5: ISO 9798-3 key exchange protocol for mutual authentication. A and B are the names of two parties that, at the end of the protocol, share a session key g^{xy}.

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 §4.3).

We start by giving a high-level overview of how we model this ISO key exchange protocol in §4.1, then state our security result in §4.2, and finally discuss some of the features of our modeling in §4.3.

4.1 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 §2.2. 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 the full version due to space limitations. 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. In the following, we give a high-level overview of each protocol.

The ISO key exchange (Figure 5) is modeled as a real protocol $\mathcal{P}_{\mathrm{KE}}$ 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 $\mathcal{F}_{\mathrm{CA}}$ modeling a certificate authority (CA) that is used to distribute public verification keys generated by $\mathcal{F}_{\text{sig-CA}}$. The real protocol $\mathcal{P}_{\mathrm{KE}}$, as already mentioned in §2.2, 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)[11], 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 §2.2, 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.[12]

---

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

[12] 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

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 (Register, key) from I/O to (_, _, registration):
    if keys[pid_call, sid_call] ≠ ⊥:
        reply (Register, failed).
    else:
        keys[pid_call, sid_call] = key
        reply (Register, 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 (Retrieve, (pid, sid)) from _ to (_, _, retrieval):
    reply (Retrieve, 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.

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 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 §4.3). 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.

---

not only requires global SIDs (cf. §4.3) but also changes the messages that are signed, thus creating a modified protocol with different security properties.

4.2 Security Result

For the above modeling, we obtain the following result, with a proof provided in the full version.

Theorem 1. *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:*

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

Note that we can realize $\mathcal{F}_{\text{sig-CA}}$ via a generic implementation $\mathcal{P}_{\text{sig-CA}}$ of a digital signature scheme (we provide a formal definition of $\mathcal{P}_{\text{sig-CA}}$ in the full version):

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*

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

Proof. Analogous to the proof in [20].

By Corollary 1, we can thus immediately replace the subroutine $\mathcal{F}_{\text{sig-CA}}$ of $\mathcal{P}_{\mathrm{KE}}$ 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 1 and Lemma 1 are fulfilled, then*

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

4.3 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.[13] 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 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

[13] 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 are only partially global. This is illustrated by $\mathcal{F}_{\mathrm{CA}}$ 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 $\mathcal{F}_{\mathrm{CA}}$ 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 $\mathcal{F}_{\mathrm{CA}}$ means that, if the ideal protocol ($\mathcal{F}_{\mathrm{KE}}, \mathcal{F}_{\mathrm{CA}}$: `retrieval` | $\mathcal{F}_{\mathrm{CA}}$: `registration`) is used as a subroutine for a new hybrid protocol, then only $\mathcal{F}_{\mathrm{KE}}$ but not the higher-level protocol can register keys in $\mathcal{F}_{\mathrm{CA}}$. If desired, one can, however, also obtain a single global $\mathcal{F}_{\mathrm{CA}}$ where both $\mathcal{F}_{\mathrm{KE}}$ 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 $\mathcal{F}_{\mathrm{CA}}$, say $\mathcal{F}'_{\mathrm{CA}}$, 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 $\mathcal{F}_{\mathrm{CA}}$ and $\mathcal{F}'_{\mathrm{CA}}$ with a joint-state realization where keys are stored in and retrieved from the same $\mathcal{F}_{\mathrm{CA}}$ subroutine along with a protocol dependent tag (we discuss this novel type of joint-state realization in detail in the full version). Of course, this approach can be iterated to support arbitrarily many protocols using the same $\mathcal{F}_{\mathrm{CA}}$. This modeling reflects reality where keys are certified for certain contexts/purposes.

5 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: ASIACRYPT 2016. LNCS, vol. 10032, pp. 807–840. Springer (2016), available at http://eprint.iacr.org/2016/034.
2. Camenisch, Krenn, S., Küsters, R., Rausch, D.: iUC: Flexible Universal Composability Made Simple (Full Version). Tech. Rep. 2019/1073, Cryptology ePrint Archive (2019), available at http://eprint.iacr.org/2019/1073
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: 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: TCC 2007. LNCS, vol. 4392, pp. 61–85. Springer (2007)
6. Canetti, R., Krawczyk, H.: Universally Composable Notions of Key Exchange and Secure Channels. In: EUROCRYPT 2002. LNCS, vol. 2332, pp. 337–351. Springer (2002)
7. Canetti, R., Rabin, T.: Universal Composition with Joint State. In: CRYPTO 2003, LNCS, 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: ACNS 2011. LNCS, 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: CRYPTO 2015 - Proceedings, Part II. LNCS, vol. 9216, pp. 3–22. Springer (2015)
11. Canetti, R., Hogan, K., Malhotra, A., Varia, M.: A Universally Composable Treatment of Network Time. In: CSF 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: PKC 2016 - Proceedings, Part II. LNCS, 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: ASIACRYPT 2018 - Proceedings, Part III. LNCS, 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: CSFW 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: 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: 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: 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: S&P 2001. pp. 184–201. IEEE Computer Society (2001)