Technical Report: Unifying Formally Verified and Cryptographically Sound Proofs of Security Protocols *

Michael Backes[1], Christian Jacobi[2], and Birgit Pfitzmann[1]
[1]IBM Zurich Research Lab, Rüschlikon, Switzerland,
[2]IBM Deutschland Entwicklung GmbH, Processor Development 2, Böblingen, Germany

September 29, 2006

Abstract

We investigate how formal methods can be used for the verification of cryptographic protocols such that the verified properties are valid for the concrete implementation of the protocol using actual cryptography. We give an abstract deterministic specification for secure message transmission with ordered channels along with a possible implementation that we prove to be secure in the sense of simulatability, which is the cryptographic notion of a secure refinement. The correctness of this proof relies on a composition theorem and a deterministic bisimulation, which we formally verify using the theorem prover PVS. We further use PVS to formally verify that message reordering is in fact prevented in the specification. We finally show that integrity properties are preserved under simulatability, which allows for carrying over the proven property to the concrete implementation. This yields the first example ever of a formally verified but nevertheless cryptographically sound proof of a security protocol.

Keywords: Security, cryptographic protocols, verification, integrity, simulatability

1 Introduction

Many practically relevant cryptographic protocols like SSL/TLS, S/MIME, IPSec, or SET use cryptographic primitives like signature schemes or encryption in a black-box way, while adding many non-cryptographic features. Vulnerabilities have accompanied the design of such protocols ever since early authentication protocols like Needham-Schroeder [61, 36], over carefully designed de-facto standards like SSL and PKCS [70, 31], up to current widely deployed products like Microsoft Passport [40]. However, proving the security of such protocols has been a very unsatisfactory task for a long time.

One possibility was to take the cryptographic approach. This means reduction proofs between the security of the overall system and the security of the cryptographic primitives, i.e., one shows that if an overall system could be broken, one of the underlying cryptographic primitives could also be broken with respect to their cryptographic definitions, e.g., adaptive chosen-message security for signature schemes. For authentication protocols, this approach was first used in [30]. In principle, proofs in this approach are as rigorous as typical proofs in mathematics. In practice, however, human beings are extremely fallible with this type of proofs. This is not due to the cryptography, but to the distributed-systems aspects of the protocols. It is well-known from non-cryptographic distributed systems that many

*Parts of this work were published in [11] and [10]. These parts were done while two of the authors were affiliated with Saarland University.

wrong protocols have been published even for very small problems. Hand-made proofs are highly error-prone because following all the different cases how actions of different machines interleave is extremely tedious. Humans tend to take wrong shortcuts and do not want to proof-read such details in proofs by others. If the protocol contains cryptography, this obstacle is even much worse: Already a rigorous definition of the goals gets more complicated, and often not only trace properties (integrity) have to be proven but also secrecy. Further, in principle the complexity-theoretic reduction has to be carried out across all these cases, and it is not at all trivial to do this rigorously. In consequence, there is almost no real cryptographic proof of a larger protocol, and several times supposedly proven, relatively small systems were later broken, e.g., [66, 37].

The other possibility was to use formal methods. There one leaves the tedious parts of proofs to machines, i.e., model checkers or automatic theorem provers. This means to code the cryptographic protocols into the language of such tools, which may need more or less start-up work depending on whether the tool already supports distributed systems or whether interaction models have to be encoded first. None of these tools, however, is currently able to deal with reduction proofs. Nobody even thought about this for a long time, because one felt that protocol proofs could be based on simpler, idealized abstractions from cryptographic primitives. Almost all these abstractions are variants of the Dolev-Yao model [38], which represents all cryptographic primitives as operators of a term algebra with cancellation rules. For instance, public-key encryption is represented by operators E for encryption and D for decryption with one cancellation rule, $\mathsf{D}(\mathsf{E}(m)) = m$ for all m. Encrypting a message m twice in this model does not yield another message from the basic message space but the term $\mathsf{E}(\mathsf{E}(m))$. Further, the model assumes that two terms whose equality cannot be derived with the cancellation rules are not equal, and every term that cannot be derived is completely secret. However, originally there was no foundation at all for such assumptions about real cryptographic primitives, and thus no guarantee that protocols proved with these tools were still secure when implemented with real cryptography. Although no previously proved protocol has been broken when implemented with standard provably secure cryptosystems, this was clearly an unsatisfactory situation, and artificial counterexamples can be constructed.

Three years ago, efforts started to get the best of both worlds. Essentially, [65, 67] started to define general cryptographic models that support idealization that is secure in arbitrary environments and under arbitrary active attacks, while [2] started to justify the Dolev-Yao model as far as one could without such a model. Both directions were significantly extended in subsequent papers, see the related work section below. At the time of the research of this report, formal proof tools have not been used for the verification of a concrete cryptographic protocol. We close this gap by presenting the first tool-supported security proof of a cryptographic protocol such that the proof is valid with respect to the cryptographic semantics. Our paper is based on a model of reactive systems in asynchronous networks [68, 24, 22], and it essentially consists of two parts:

In the first part, we define integrity properties in the underlying model, and we prove that they are preserved under simulatability, which captures the cryptographic notion of a secure refinement. This means that integrity properties automatically carry over from an abstract specification to a concrete implementation if the implementation is proved to be secure in the sense of simulatability, Moreover, we show that logic derivations among integrity properties are valid for the concrete implementation in the cryptographic sense, which is essential to make the properties accessible to theorem provers.

The second part of this paper is dedicated to the actual verification of a cryptographic protocol: secure message transmission with ordered channels. We present a detailed deterministic specification of secure message transmission with ordered channels and we subsequently derive a secure implementation by refining the specification with respect to simulatability. The correctness proof of this refinement mainly relies on a composition theorem of the underlying model and of a deterministic bisimulation which we formally verify in a theorem proving system. We finally verify the desired integrity property

– preventing message reordering – for the specification, and we use the integrity preservation theorem established in the first part of this work to carry over this property to the concrete secure implementation.

This yields the first example of a machine-aided proof of a cryptographic protocol that is nevertheless sound with respect to the cryptographic definitions.

Related Literature Both the cryptographic and the idealizing approach at proving cryptographic systems started in the early 80s. Early examples of cryptographic definitions and reduction proofs are [42, 43]. Applied to protocols, these techniques are at their best for relatively small protocols where there is still a certain interaction between cryptographic primitives, e.g., [29, 69]. The early methods of automating proofs based on the Dolev-Yao model are summarized in [48]. More recently, such work concentrated on using existing general-purpose model checkers [51, 60, 34] and theorem provers [39, 64], and on treating larger protocols, e.g., [28].

Work intended to bridge the gap between the cryptographic approach and the use of automated tools started independently with [65, 67] and [2]. In [2], Dolev-Yao terms, i.e., with nested operations, are considered specifically for symmetric encryption. However, the adversary is restricted to passive eavesdropping. Consequently, it was not necessary to define a reactive model of a system, its honest users, and an adversary, and the security goals were all formulated as indistinguishability of terms. This was extended in [1] from terms to more general programs, but the restriction to passive adversaries remains, which is not realistic in most practical applications. Further, there are no theorems about composition or property preservation from the abstract to the real system. Several papers extended this work for specific models or specific properties. For instance, [44] specifically considers strand spaces and information-theoretically secure authentication only. In [49] a deduction system for information flow is based on the same operations as in [2], still under passive attacks only.

The approach in [65, 67] was from the other end: It starts with a general reactive system model, a general definition of cryptographically secure implementation by simulatability, and a composition theorem for this notion of secure implementation. This work is based on definitions of secure *function* evaluation, i.e., the computation of one set of outputs from one set of inputs [41, 58, 27, 32]; earlier extensions towards reactive systems were either without real abstraction [50] or for quite special cases [45]. The approach was extended from synchronous to asynchronous systems in [68, 33, 24]. All the reactive works come with more or less worked-out examples of abstractions of cryptographic systems, however they have not investigated the use of formal methods for the verification of a concrete example. As of now (3,5 years after the original publication of the papers [11] and [10] that underlie this report), computational soundness has become a highly active line of research, see e.g., [3, 21, 15, 20, 26, 23, 4, 16, 59, 18, 17, 7, 13, 8, 12].

The relationship between integrity properties and simulatability was investigated in [67], where it was shown that integrity properties are preserved under simulatability for a synchronous timing model. However, a synchronous definition of time is difficult to justify in the real world since no notion of rounds is naturally given there and it seems to be very difficult to establish them for the Internet for example. In contrast to that, asynchronous scenarios are attractive, because no assumptions are made about network delays and the relative execution speed of the parties. Technically, the first part of our work can be seen as an extension of the results of [67] to asynchronous scenarios. This extension is not trivial since synchronous time is much easier to handle; moreover, both models do not only differ in the definition of time but also in subtle, but important details. Similar preservation results under simulatability have recently been shown for non-interference [14, 5] and liveness properties [19]. In general, results of these forms are particularly interesting since they offer security under system composition, which is known to be very difficult to achieve in general, see e.g., [54, 47, 55, 56, 57, 53, 35, 25, 6, 22, 9, 23, 16]).

Organization of the Paper We start with a brief review of the model for reactive systems in asynchronous networks from [68] in Section 2. In Section 3 we define what it means for a system to provide integrity properties in a cryptographic sense. We then prove that (1) proofs of such properties made for an abstract specification also hold for the concrete implementation and (2) that logic derivations among integrity properties are valid for the concrete implementation with respect to cryptographic definitions. Section 4 contains our specification of secure message transmission with ordered channels. We give a possible implementation in Section 5, which is shown to securely implement the specification in Section 6, 7, and 8. More precisely, Section 6 establishes a security proof by defining a so-called simulator, and by applying a deterministic bisimulation for proving the correctness of the refinement. Section 7 deals with the actual verification of the bisimulation within the theorem proving system PVS [63]. In Section 8 we finally verify that message reordering is in fact prevented for the deterministic specification, again using PVS, and we use our preservation theorem to show that the verified property carries over to the concrete implementation. Section 9 summarizes.

2 The Model for Reactive Systems

In this section, we recapitulate the model for asynchronous probabilistic reactive systems as introduced by Pfitzmann and Waidner in [68].

Several definitions will only be sketched, whereas those that are important for understanding our upcoming definitions and proofs are given in full detail. All other details can be looked up in the original paper.

2.1 General System Model

Systems mainly are compositions of several machines. Usually we consider real systems that are built by a set $\hat{M}$ of machines $\{\mathsf{M}_1, \ldots, \mathsf{M}_n\}$, one for each user u from a set $\mathcal{M} = \{1, \ldots, n\}$, and ideal systems built by one machine $\{\mathsf{TH}\}$.

Communication between different machines is done via ports using messages composed from an alphabet Σ. Inspired by the CSP-Notation [46], we write output and input ports as $\mathsf{q}!$ and $\mathsf{q}?$ respectively. The ports of a machine M are denoted by $\mathsf{ports}(\mathsf{M})$. The subset of input and output ports are denoted by $\mathsf{in}(\mathsf{ports}(\mathsf{M}))$ and $\mathsf{out}(\mathsf{ports}(\mathsf{M}))$, respectively. Channels are defined implicitly by naming convention, that is port $\mathsf{q}!$ sends messages to $\mathsf{q}?$. To achieve asynchronous timing, a message is not directly sent to its recipient, but it is first stored in a special machine $\widetilde{\mathsf{q}}$ called a buffer and waits to be scheduled. If a machine wants to schedule the i-th message of buffer $\widetilde{\mathsf{q}}$ (this machine must have the unique clock-out port $\mathsf{q}^{\triangleleft}!$) it simply sends i at $\mathsf{q}^{\triangleleft}!$, see Figure 1. The buffer then schedules the i-th message and removes it from its internal list. In our case, most buffers are either scheduled by a master scheduler or the adversary, i.e., one of those has the clock-out port. In [68] the adversary and the master scheduler are the same entity. This gives the adversary complete control over the overall scheduling of network traffic and models the worst-case behavior we usually have to expect in an asynchronous system. We define the complement p^c of a port p to be the port which it connects to according to Figure 1, i.e., $\mathsf{q}!^c = \mathsf{q}^{\leftrightarrow}?$, $\mathsf{q}^{\triangleleft}!^c = \mathsf{q}^{\triangleleft}?$, $\mathsf{q}^{\leftrightarrow}!^c = \mathsf{q}?$, and vice versa. We use the same notation for sets of ports.

After introducing ports, we now focus on the definition of machines. Our machine model is probabilistic state-transition machines, similar to probabilistic I/O automata as sketched by Lynch [52]. If a machine is switched, it receives an input tuple at its input ports and performs its transition function yielding a new state and an output tuple in the deterministic case, or a finite distribution over the set of states and possible outputs in the probabilistic case. At each switching step of one particular machine, at most one value can arrive at every input port and the machine can produce at most one output per

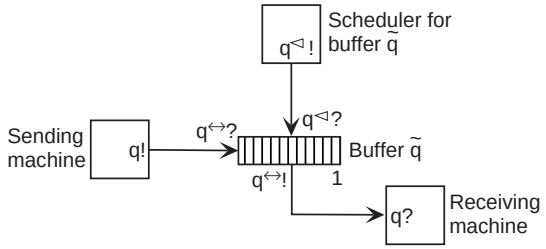

Figure 1: Ports and buffers.

port. Furthermore, each machine has a bound on the length of the considered inputs which allows time bounds independent of the environment.

Definition 2.1 *(Machines) A* machine *is a tuple*

$$\mathsf{M} = (name_{\mathsf{M}}, Ports_{\mathsf{M}}, States_{\mathsf{M}}, \delta_{\mathsf{M}}, l_{\mathsf{M}}, Ini_{\mathsf{M}}, Fin_{\mathsf{M}})$$

of a name $name_{\mathsf{M}} \in \Sigma^+$, a finite sequence $Ports_{\mathsf{M}}$ of ports (i.e., $Ports_{\mathsf{M}} = \mathsf{ports}(\mathsf{M})$), a set $States_{\mathsf{M}} \subseteq \Sigma^*$ of states, a computable probabilistic state-transition function δ_{M}, a length function $l_{\mathsf{M}} : States_{\mathsf{M}} \to (\mathbb{N} \cup \{\infty\})^{|\mathsf{in}(Ports_{\mathsf{M}})|}$, and sets $Ini_{\mathsf{M}}, Fin_{\mathsf{M}} \subseteq States_{\mathsf{M}}$ of initial and final states. Its input set is $\mathcal{I}_{\mathsf{M}} := (\Sigma^*)^{|\mathsf{in}(Ports_{\mathsf{M}})|}$; the i-th element of an input tuple denotes the input at the i-th in-port. Its output set is $\mathcal{O}_{\mathsf{M}} := (\Sigma^*)^{|\mathsf{out}(Ports_{\mathsf{M}})|}$. The empty word, ϵ, denotes no in- or output at a port. δ_{M} probabilistically maps each pair $(s, I) \in States_{\mathsf{M}} \times \mathcal{I}_{\mathsf{M}}$ of state and inputs to a pair $(s', O) \in States_{\mathsf{M}} \times \mathcal{O}_{\mathsf{M}}$ of successor states and outputs. Following two restrictions apply to δ_{M}: (1) The induced output distribution has to be finite, and (2) if $s \in Fin_{\mathsf{M}}$ or $I = (\epsilon, \ldots, \epsilon)$, then $\delta_{\mathsf{M}}(s, I)$ maps always to the same state and no output, i.e, $(s, (\epsilon, \ldots, \epsilon))$. Inputs are ignored beyond the length bounds, i.e., $\delta_{\mathsf{M}}(s, I) = \delta_{\mathsf{M}}(s, I\lceil_{l_{\mathsf{M}}(s)})$ for all $I \in \mathcal{I}_{\mathsf{M}}$, where $R\lceil_l := (r\lceil_l)_{r \in R}$ for $R \in (\Sigma^*)^*$ and $r\lceil_l$ denotes the l-bit prefix of a sequence $r \in \Sigma^*$. $\diamond$

In the text, we often write "M" also for $name_{\mathsf{M}}$. We only briefly state here that these machines have a natural realization as a probabilistic Turing machine.

A *collection* $\hat{C}$ of machines is a finite set of machines with pairwise different machine names and disjoint sets of ports. The *completion* $[\hat{C}]$ of a collection $\hat{C}$ is the union of all machines of $\hat{C}$ and the buffers needed for every channel. A port of a collection is called *free* if its connecting port is not in the collection. These port will be connected to the users and the adversary. The free ports of a completion $[\hat{C}]$ are denoted as $\mathsf{free}([\hat{C}])$. A collection $\hat{C}$ is called *closed* if its completion $[\hat{C}]$ has no free ports except a special master clock-in port $\mathsf{clk}^{\triangleleft}?$, i.e., $\mathsf{free}([\hat{C}]) = \{\mathsf{clk}^{\triangleleft}?\}$. The master clock-in port $\mathsf{clk}^{\triangleleft}?$ is used to give control to the master scheduler as shown below. By convention, we assume that the master scheduler expects a 1 as input on this port.

A closed collection represents a "runnable" system. For such a closed collection, a probability space of runs (sometimes called *traces* or *executions*) is defined. Scheduling of machines is done sequentially, so we have exactly one active machine M at any time. If this machine has clock-out ports, it is allowed to select the next message to be scheduled as explained above. If that message exists, it is delivered by the buffer and the unique receiving machine is the next active machine. If M tries to schedule multiple messages, only one is taken, and if it schedules none or the message does not exist, the special master scheduler is scheduled. Formally, runs are defined as follows.

Definition 2.2 *(Runs) Given a closed collection $\hat{C}$ with master scheduler* X *and a tuple $ini \in Ini_{\hat{C}} := \times_{\mathsf{M} \in \hat{C}} Ini_{\mathsf{M}}$ of initial states, the probability space of* runs *is defined inductively by the following algorithm. It has a variable r for the resulting run, an initially empty list, a variable M_{CS} ("current*

scheduler") over machine names, initially $\mathsf{M_{CS}} := \mathsf{X}$*, and treats each port as a variable over* Σ^**, initialized with* ϵ *except for* $\mathsf{clk}^{\triangleleft}? := 1$*. Probabilistic choices only occur in Step (1).*

1. Switch current scheduler: *Switch machine* $\mathsf{M_{CS}}$*, i.e., for a given current state* s *and in-port values* I*, set the new state and output* (s', O) *to the output of* $\delta_{\mathsf{M_{CS}}}(s, I)$*. Then assign* ϵ *to all in-ports of* $\mathsf{M_{CS}}$*.*

2. Termination: *If* X *is in a final state, the run stops.*

3. Buffer messages: *For each simple out-port* $\mathsf{q}!$ *of* $\mathsf{M_{CS}}$*, in their given order, switch buffer* $\widetilde{\mathsf{q}}$ *with input* $\mathsf{q}^{\leftrightarrow}? := \mathsf{q}!$*, cf. Figure 1. Then assign* ϵ *to all these ports* $\mathsf{q}!$ *and* $\mathsf{q}^{\leftrightarrow}?$*.*

4. Clean up scheduling: *If at least one clock out-port of* $\mathsf{M_{CS}}$ *has a value* $\neq \epsilon$*, let* $\mathsf{q}^{\triangleleft}!$ *denote the first such port and assign* ϵ *to the others. Otherwise let* $\mathsf{clk}^{\triangleleft}? := 1$ *and* $\mathsf{M_{CS}} := \mathsf{X}$ *and go back to Step (1).*

5. Scheduled message: *Switch* $\widetilde{\mathsf{q}}$ *with input* $\mathsf{q}^{\triangleleft}? := \mathsf{q}^{\triangleleft}!$ *(cf. Figure 1), set* $\mathsf{q}? := \mathsf{q}^{\leftrightarrow}!$ *and then assign* ϵ *to all ports of* $\widetilde{\mathsf{q}}$ *and to* $\mathsf{q}^{\triangleleft}!$*. Let* $\mathsf{M_{CS}} := \mathsf{M}'$ *for the unique machine* M' *with* $\mathsf{q}? \in \mathsf{ports}(\mathsf{M}')$*. Go back to Step (1).*

Whenever a machine (this may be a buffer) with name $name_{\mathsf{M}}$ *is switched from* (s, I) *to* (s', O)*, we add a* step $(name_{\mathsf{M}}, s, I', s', O)$ *to the run* r *for* $I' := I\lceil_{l_{\mathsf{M}}(s)}$*, except if* s *is final or* $I' = (\epsilon, \ldots, \epsilon)$*. This gives a family of random variables indexed by the possible initial states*

$$run_{\hat{C}} := (run_{\hat{C},ini})_{ini \in Ini_{\hat{C}}}.$$

For a number $l \in \mathbb{N}$*, l-step prefixes* $run_{\hat{C},ini,l}$ *of runs are defined in the obvious way. For a function* $l(\cdot) : Ini_{\hat{C}} \to \mathbb{N}$*, this gives a family* $run_{\hat{C},l(\cdot)} = (run_{\hat{C},ini,l(ini)})_{ini \in Ini_{\hat{C}}}$*.* ◇

Definition 2.3 *(Views and Restrictions to Ports) The* view *of a subset* $\hat{M}$ *of a closed collection* $\hat{C}$ *in a run* r *is the restriction of* r *to* $\hat{M}$*, i.e., the subsequence of all steps* $(name_{\mathsf{M}}, s, I, s', O)$ *where* $name_{\mathsf{M}}$ *is the name of a machine* $\mathsf{M} \in \hat{M}$*. Similarly, for a set* S *of ports, we define the restriction* $r\lceil_S$ *of a run* r *to the set* S*, i.e., for every step of the run, we leave out the name* $name_{\mathsf{M}}$ *and the states* s, s'*, and restrict the sets* I *and* O *to the ports in* S*. This gives two families of random variables*

$$view_{\hat{C}}(\hat{M}) = (view_{\hat{C},ini}(\hat{M}))_{ini \in Ini_{\hat{C}}} \text{ and}$$

$$run_{\hat{C}}\lceil_S = (run_{\hat{C},ini}\lceil_S)_{ini \in Ini_{\hat{C}}}$$

and similarly for l-step prefixes. For a singleton $\hat{M} = \{\mathsf{H}\}$ *we write* $view_{\hat{C}}(\mathsf{H})$ *instead of* $view_{\hat{C}}(\{\mathsf{H}\})$*.* ◇

2.2 Security-specific System Model

For security purposes, special collections are needed, because an adversary may have taken over parts of the initially intended system. Therefore, a system consists of several possible remaining structures. First, the system part is defined and then the environment, consisting of users and adversaries.

Definition 2.4 *(Structures and Systems)*

a) *A* structure *is a pair $struc = (\hat{M}, S)$ where $\hat{M}$ is a collection of simple machines (i.e., with only normal in- and output ports and clock-out ports) called* correct machines*, and $S \subseteq \mathsf{free}([\hat{M}])$ is called* specified ports*. If $\hat{M}$ is clear from the context, let $\bar{S} := \mathsf{free}([\hat{M}]) \setminus S$. We call $\mathsf{forb}(\hat{M}, S) := \mathsf{ports}(\hat{M}) \cup \bar{S}^c$ the* forbidden ports*, i.e., those ports that an honest user should be forbidden to have. (The ports in $\mathsf{ports}(\hat{M})$ belong to the structure and must hence not be used by the user because of name clashes; the ports in $\bar{S}^c$ should belong to the adversary.)*

b) *A* system *Sys is a set of structures. It is polynomial-time iff all machines in all its collections $\hat{M}$ are polynomial-time.*

$\diamond$

The separation of the free ports into specified ports and others is an important feature of the upcoming security definitions. The specified ports are those where a certain abstract service is guaranteed. Typical examples of inputs at specified ports are "send message m to id" for a message transmission system or "pay amount x to id" for a payment system. The ports in $\bar{S}$ are additionally available for the adversary. The ports in $\mathsf{forb}(\hat{M}, S)$ will therefore be forbidden for an honest user to have.

A structure can be completed to a *configuration* by adding machines H and A, modeling the joint honest users and the adversary, respectively. The machine H is restricted to the specified ports S, A connects to the remaining free ports of the structure and both machines can interact, e.g., in order to model active attacks.

Definition 2.5 *(Configurations)*

a) *A* configuration *of a system Sys is a tuple $conf = (\hat{M}, S, \mathsf{H}, \mathsf{A})$ where $(\hat{M}, S) \in Sys$ is a structure, H is a machine without forbidden ports, i.e., $\mathsf{ports}(\mathsf{H}) \cap \mathsf{forb}(\hat{M}, S) = \emptyset$, and the completion $\hat{C} := [\hat{M} \cup \{\mathsf{H}, \mathsf{A}\}]$ is a closed collection. The set of configurations is written $\mathsf{Conf}(Sys)$.*

b) *The initial states of all machines in a configuration are a common security parameter k in unary representation. This means that we consider the families of runs and views of the collection $\hat{C}$ restricted to the subset $Ini'_{\hat{C}} := \{(1^k)_{\mathsf{M} \in \hat{C}} | k \in \mathbb{N}\}$ of $Ini_{\hat{C}}$. We write run_{conf} and $view_{conf}(\hat{M})$ for the families $run_{\hat{C}}$ and $view_{\hat{C}}(\hat{M})$ restricted to $Ini'_{\hat{C}}$, and similar for l-step prefixes. Furthermore, we identify $Ini'_{\hat{C}}$ with $\mathbb{N}$ and thus write $run_{conf,k}$ etc. for the individual random variables.*

c) *The set of configurations of Sys with polynomial-time user H and adversary A is called $\mathsf{Conf}_{\mathsf{poly}}(Sys)$. The index $_{\mathsf{poly}}$ is omitted if it is clear from the context.*

$\diamond$

We only briefly state here that several machines can be combined into one single machine (which has the original machines as submachines), cf. [68] for more details. Moreover, the view of every submachine remains unchanged by this combination.

2.3 Defining Security with Simulatability

As we will see below, the system model provides a powerful instrument to compare two systems and to assess whether one system securely implements another one. Based on this, our approach in defining security is as follows: (1) We define the abstract specification of a secure service as an ideal system Sys_{id} consisting of a single machine TH. Given the simplicity of the idealized machine, the correctness of the specification is often intuitively clear. Furthermore, we can gain additional confidence by analyzing TH

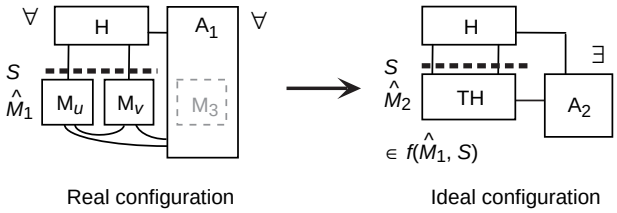

Figure 2: Example of simulatability. The view of H is compared.

using formal methods and automated tools. (2) Given any concrete real system Sys_{real} implementing the desired service, we then prove its security by showing that it securely implements Sys_{id}.

The definition of one system securely implementing another one is based on the common concept of *simulatability*. The notion of simulatability was introduced in [71] and has asserted its position as a fundamental concept of modern cryptography. Simulatability essentially means that whatever might happen to an honest user in a concrete system Sys_{real} can also happen in an ideal system Sys_{id}. As by definition only good things can happen in the ideal system, simulatability guarantees that no bad things can happen in the real system. More precisely, for every configuration $conf_1 \in \mathsf{Conf}(Sys_{\mathsf{real}})$, there exists a configuration $conf_2 \in \mathsf{Conf}(Sys_{\mathsf{id}})$ yielding indistinguishable views of the same user in both configurations. We abbreviate this by $Sys_{\mathsf{real}} \geq_{\mathsf{sec}} Sys_{\mathsf{id}}$ and we say that Sys_{real} is "at least as secure" as the system Sys_{id}. A typical situation is illustrated in Figure 2.

However, we do not want to compare a structure $(\hat{M}_1, S_1) \in Sys_{\mathsf{real}}$ with arbitrary structures of Sys_{id}, but only with certain "suitable" ones. What suitable actually means can be defined by a mapping f from Sys_{real} to the powerset of Sys_{id}. The mapping f is called *valid* if it maps structures with the same set of specified ports.

The upcoming simulatability definition is based on indistinguishability of views.

Definition 2.6 *(Indistinguishability) Two families* $(\mathsf{var}_k)_{k\in\mathbb{N}}$ *and* $(\mathsf{var}'_k)_{k\in\mathbb{N}}$ *of random variables (or probability distributions) on common domains* D_k *are*

a) perfectly indistinguishable *("=") if for each* k*, the two distributions* var_k *and* var'_k *are identical.*

b) statistically indistinguishable *("*$\approx_{SMALL}$*") for a suitable class* $SMALL$ *of functions from* $\mathbb{N}$ *to* $\mathbb{R}_{\geq 0}$ *if the distributions are discrete and their statistical distances*

$$\Delta(\mathsf{var}_k, \mathsf{var}'_k) := \frac{1}{2} \sum_{d \in D_k} |P(\mathsf{var}_k = d) - P(\mathsf{var}'_k = d)| \in SMALL$$

(as a function of k*).* $SMALL$ *must be closed under addition, and with a function* g *also contain every function* $g' \leq g$*.*

c) computationally indistinguishable *("*$\approx_{\mathsf{poly}}$*") if for every algorithm* Dis *(the distinguisher) that is probabilistic polynomial-time in its first input,*

$$|P(\mathsf{Dis}(1^k, \mathsf{var}_k) = 1) - P(\mathsf{Dis}(1^k, \mathsf{var}'_k) = 1)| \in NEGL.$$

Intuitively, given the security parameter and an element chosen according to either var_k *or* var'_k*,* Dis *tries to guess which distribution the element came from. The class* $NEGL$ *denotes the set of all negligible functions, i.e.,* $g\colon \mathbb{N} \to \mathbb{R}_{\geq 0} \in NEGL$ *if for all positive polynomials* Q*,* $\exists k_0 \forall k \geq k_0$*:* $g(k) \leq 1/Q(k)$*.*

We write $\approx$ if we want to treat all three cases simultaneously. $\diamond$

We now present the simulatability definition.

Definition 2.7 *(Simulatability) Let systems Sys_1 and Sys_2 with a valid mapping f be given.*

*a) We say $Sys_1 \geq_{\mathsf{sec}}^{f,\mathsf{perf}} Sys_2$ (*perfectly at least as secure as*) if for every configuration $conf_1 = (\hat{M}_1, S, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}(Sys_1)$, there exists a configuration $conf_2 = (\hat{M}_2, S, \mathsf{H}, \mathsf{A}_2) \in \mathsf{Conf}(Sys_2)$ with $(\hat{M}_2, S) \in f(\hat{M}_1, S)$ (and the same H) such that*

$$view_{conf_1}(\mathsf{H}) = view_{conf_2}(\mathsf{H}).$$

*b) We say $Sys_1 \geq_{\mathsf{sec}}^{f,SMALL} Sys_2$ (*statistically at least as secure as*) for a class $SMALL$ if the same as in a) holds with $view_{conf_1,l}(\mathsf{H}) \approx_{SMALL} view_{conf_2,l}(\mathsf{H})$ for all polynomials l, i.e., statistical indistinguishability of all families of l-step prefixes of the views.*

*c) We say $Sys_1 \geq_{\mathsf{sec}}^{f,\mathsf{poly}} Sys_2$ (*computationally at least as secure as*) if the same as in a) holds with configurations from $\mathsf{Conf}_{\mathsf{poly}}(Sys_1)$ and $\mathsf{Conf}_{\mathsf{poly}}(Sys_2)$ and computational indistinguishability of the families of views.*

In all cases, we call $conf_2$ an indistinguishable configuration *for $conf_1$. Where the difference between the types of security is irrelevant, we simply write $\geq_{\mathsf{sec}}^{f}$, and we omit the index f if it is clear from the context.* $\diamond$

Clearly, perfect simulatability implies statistical simulatability for every non-empty class $SMALL$. Similarly, statistical simulatability for a class $SMALL$ implies computational simulatability if $SMALL \subseteq NEGL$.

An important feature of the system model is transitivity of $\geq_{\mathsf{sec}}$, i.e., the preconditions $Sys_1 \geq_{\mathsf{sec}} Sys_2$ and $Sys_2 \geq_{\mathsf{sec}} Sys_3$ together imply $Sys_1 \geq_{\mathsf{sec}} Sys_3$, which has been proved in [68].

2.4 Composition

We conclude this section with a brief review of what has already been proven about composition of reactive systems. Assume that we have already proven that a system Sys_0 is at least as secure as another system Sys_0'. Typically Sys_0 is a concrete system whereas Sys_0' is an ideal specification of the concrete system. If we now consider larger protocols that use Sys_0' as an ideal primitive we would like to securely replace it with Sys_0. In practice this means that we replace the specification of a system with its implementation yielding a concrete system.

Usually, replacing means that we have another system Sys_1 using Sys_0'; we call this composition Sys^*, cf. Figure 3. We now want to replace Sys_0' with Sys_0 inside of Sys^* which gives a composition $Sys^\#$. Typically $Sys^\#$ is a completely real system whereas Sys^* is at least partly ideal. This is illustrated in the left and middle part of Figure 3. The composition theorem now states that this replacement maintains security, i.e., $Sys^\#$ is at least as secure as Sys^* (see [68] for details).

However, typically a specification of the overall system should not prescribe that the implementation must have two subsystems; e.g., in specifying a payment system, it should be irrelevant whether the implementation uses secure message transmission as a subsystem. Hence, the overall specification is typically monolithic, cf. Sys^{spec} in Figure 3. Moreover, such specifications are well-suited for formal verification, because they typically are deterministic and single machines are furthermore much easier to validate. Our specification in Section 4 is of this kind.

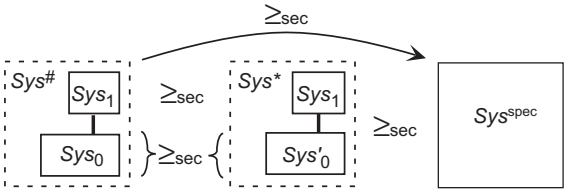

Figure 3: Composition of Systems.

3 Integrity Properties

In this section, we show how the relation "at least as secure as" relates to integrity properties a system should fulfill, e.g., safety properties expressed in temporal logic.

3.1 Definition of Integrity Properties

As a rather general version of integrity properties, independent of the concrete formal language, we consider those that have a linear-time semantics, i.e., that correspond to a set of allowed traces of in- and outputs. We allow different properties for different sets of specified ports, since different requirements of various parties in cryptography are often made for different trust assumptions. We will show later on that integrity properties are preserved under simulatability which allows sound refinement of abstract systems. Clearly this can only hold for properties formulated in terms of inputs and outputs at the specified ports of a given structure, since only these ports are considered by simulatability.

Definition 3.1 (Integrity Properties) An integrity property Req for a system Sys is a function that assigns to each set S with $(\hat{M}, S) \in Sys$ a set of traces at the ports in S. Informally speaking, Req states which are the "good" traces for the given structure. More precisely such a trace is a sequence $(v_t)_{t\in I}$ of values over port names and Σ^* with $I = \{1, \ldots, l\}$ for $l \in \mathbb{N}$ or $I = \mathbb{N}$, i.e., sets of port-value pairs so that v_t is of the form $v_t := \bigcup_{p\in\mathcal{S}'}\{p : v_{p,t}\}$ for a subset $\mathcal{S}' \subseteq S$ and $v_{p,t} \in \Sigma^*$. Intuitively, $\mathcal{S}'$ contains those ports where "something happens". ◇

After introducing what integrity properties are, we have to define what it means that a system fulfills them. We will see that there are different grades of fulfillment. We distinguish between *perfect*, *statistical*, and *computational* fulfillment, depending on whether the integrity property always holds, or only with overwhelming probability, i.e., the probability of failure should be statistically small or negligible in polynomial-time configurations, respectively.

Definition 3.2 (Fulfillment of Integrity Properties) Let an arbitrary system Sys and an integrity property Req for Sys be given. Then Sys fulfills Req

a) *perfectly* (written $Sys \models^{\mathsf{perf}} Req$) if for any configuration $conf = (\hat{M}, S, \mathsf{H}, \mathsf{A}) \in \mathsf{Conf}(Sys)$, the restrictions $r\lceil_S$ of all runs of this configuration to the specified ports S lie in $Req(S)$. In formulas, $[(run_{conf,k}\lceil_S)] \subseteq Req(S)$ for all k, where $[\cdot]$ denotes the carrier set of a probability distribution.

b) *statistically* for a class $SMALL$ ($Sys \models^{SMALL} Req$) if for any configuration $conf = (\hat{M}, S, \mathsf{H}, \mathsf{A}) \in \mathsf{Conf}(Sys)$, the probability that $Req(S)$ is not fulfilled is small, i.e., for all polynomials l (and as a function of k),

$$P(run_{conf,k,l(k)}\lceil_S \notin Req(S)) \in SMALL.$$

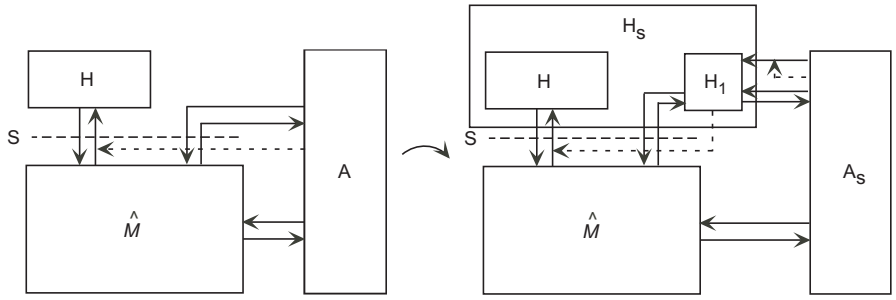

Figure 4: Sketch of the proof of Lemma 3.1

The class $SMALL$ must be closed under addition and making functions smaller.

c) *computationally* ($Sys \models^{\mathsf{poly}} Req$) if for any polynomial configuration $conf = (\hat{M}, S, \mathsf{H}, \mathsf{A}) \in \mathsf{Conf}_{\mathsf{poly}}(Sys)$, the probability that $Req(S)$ is not fulfilled is negligible, i.e.,

$$P(run_{conf,k}\lceil_S \notin Req(S)) \in NEGL.$$

For the computational and statistical case, the trace has to be finite. Note that a) is normal fulfillment. We write "$\models$" if we want to treat all three cases together. $\diamond$

Obviously, perfect fulfillment implies statistical fulfillment for every non-empty class $SMALL$ and statistical fulfillment for a class $SMALL$ implies fulfillment in the computational case if $SMALL \subseteq NEGL$.

3.2 Preservation of Integrity Properties Under Refinement

In this section, we show that our definitions of integrity properties and their fulfillment behaves well under simulatability. Usually, defining a cryptographic system starts with an abstract specification stating what the system should do. After that, this specification can be refined stepwise with respect to simulatability, which finally yields a secure implementation. At this time, we may wonder whether the verification of these properties made for the ideal specification carries over to the concrete implementation. This is essential for modular proofs. We can answer this question in the affirmative yielding the preservation theorem presented below.

The actual proof will be done by contradiction, i.e., we will show that if the concrete implementation did not fulfill its goals, the two systems could be distinguished. However, in order to exploit simulatability, we have to consider an honest user that connects to *all* specified ports. Otherwise, the contradiction might stem from those specified ports which are connected to the adversary, but those ports are not considered by simulatability. The following lemma circumvents this problem:

Lemma 3.1 *Let a system Sys be given. For every configuration $conf = (\hat{M}, S, \mathsf{H}, \mathsf{A}) \in \mathsf{Conf}(Sys)$, there is a configuration $conf_{\mathsf{s}} = (\hat{M}, S, \mathsf{H}_{\mathsf{s}}, \mathsf{A}_{\mathsf{s}}) \in \mathsf{Conf}(Sys)$ with $S \subseteq \mathsf{ports}(\mathsf{H}_{\mathsf{s}})$, such that $run_{conf}\lceil_S = run_{conf_{\mathsf{s}}}\lceil_S$, i.e., the probability of the runs restricted to the set S of specified ports is identical in both configurations. If $conf$ is polynomial-time, then $conf_{\mathsf{s}}$ is also polynomial-time.* $\square$

Proof (sketch). Since the proof is quite technical, we only give a brief sketch. For a complete proof we refer the reader to Appendix A. We define a new machine H_1 which is inserted between the system and the adversary, so that H_1 now exactly uses the specified ports formerly connected to A (cf. Figure 4). This machine mainly forwards messages, so it does not change the probability of the runs at the specified

ports. Combination of H_1 and the original H yields the intended user H_s. The adversary A_s is mainly derived by port renaming of A with the only difference that clock-out ports of A have to be simulated by A_s in a different way, mainly by additional output ports. This will give us a configuration $conf_\mathsf{s} \in \mathsf{Conf}(Sys)$ as shown in the right side of Figure 4, where the honest user H_s connects to all specified ports. The main difficulty of the proof is that we have to ensure that the new honest user H_s is polynomial-time in case of a polynomial-time configuration. This aspect requires a thorough look at the details and significantly lengthens the proof, cf. Appendix A. ∎

Before we now turn our attention to the actual preservation theorem, we state the following well-known lemma which we will need in the theorem's proof.

Lemma 3.2 *The statistical distance $\Delta(\phi(\mathsf{var}_k), \phi(\mathsf{var}'_k))$ between a function ϕ of two random variables is at most $\Delta(\mathsf{var}_k, \mathsf{var}'_k)$.* □

Theorem 3.1 (Preservation of Integrity Properties) Let a system Sys_2 be given that fulfills an integrity property Req, i.e., $Sys_2 \models Req$, and let $Sys_1 \geq^f_{\mathsf{sec}} Sys_2$ for a valid mapping f. Then also $Sys_1 \models Req$. This holds in the perfect and statistical sense, and in the computational sense if membership in the set $Req(S)$ is decidable in polynomial time for all S. □

Proof. Req is well-defined on Sys_1, since simulatability implies that for each $(\hat{M}_1, S_1) \in Sys_1$ there exists $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$ with $S_1 = S_2$. We will now prove that if Sys_1 did not fulfill the property, the two systems could be distinguished yielding a contradiction.

Assume that a configuration $conf_1 = (\hat{M}_1, S_1, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}(Sys_1)$ contradicts the theorem. As already described above, we need an honest user that connects to all specified ports. This is precisely what Lemma 3.1 does, i.e., there is a configuration $conf_{\mathsf{s},1}$ in which the user connects to all specified ports, with $run_{conf_{\mathsf{s},1}}\lceil_{S_1} = run_{conf_1}\lceil_{S_1}$, so $conf_{\mathsf{s},1}$ also contradicts the theorem. Note that all specified ports are now connected to the honest user; thus, we can exploit simulatability. In the proof for the synchronous timing model, this problem was avoided by combining the honest user and the adversary to the new honest user. However, in the asynchronous model, this combination contradicts the definition of configurations, since this user would not be valid any more, cf. Definition 2.5.

Because of our precondition $Sys_1 \geq^f_{\mathsf{sec}} Sys_2$, there exists an indistinguishable configuration $conf_{\mathsf{s},2} = (\hat{M}, S, \mathsf{H}_\mathsf{s}, \mathsf{A}_2) \in \mathsf{Conf}(Sys_2)$, i.e., $view_{conf_{\mathsf{s},1}}(\mathsf{H}_\mathsf{s}) \approx view_{conf_{\mathsf{s},2}}(\mathsf{H}_\mathsf{s})$. By assumption, the property is fulfilled for this configuration $conf_{\mathsf{s},2}$ (perfectly, statistically, or computationally). Furthermore, the view of H_s in both configurations contains the trace at $S := S_1 = S_2$.

In the perfect case, the distributions of the views are identical. This immediately contradicts the assumption that $[(run_{conf_{\mathsf{s},1},k}\lceil_S)] \not\subseteq Req(S)$ while $[(run_{conf_{\mathsf{s},2},k}\lceil_S)] \subseteq Req(S)$.

In the statistical case, let any polynomial l be given. The statistical distance $\Delta(view_{conf_{\mathsf{s},1},k,l(k)}(\mathsf{H}_\mathsf{s}), view_{conf_{\mathsf{s},2},k,l(k)}(\mathsf{H}_\mathsf{s}))$ is a function $g(k) \in SMALL$. We apply Lemma 3.2 to the characteristic function $1_{v\lceil_S \notin Req(S)}$ on such views v. This gives

$$|P(run_{conf_{\mathsf{s},1},k,l(k)}\lceil_S \notin Req(S)) - P(run_{conf_{\mathsf{s},2},k,l(k)}\lceil_S \notin Req(S))| \leq g(k).$$

As $SMALL$ is closed under addition and under making functions smaller, this gives the desired contradiction.

In the computational case, we define a distinguisher Dis: Given the view of machine H_s, it extracts the run restricted to S and verifies whether the result lies in $Req(S)$. If yes, it outputs 0, otherwise 1. This distinguisher is polynomial-time (in the security parameter k) because the view of H_s is of polynomial length, and membership in $Req(S)$ was required to be polynomial-time decidable. Its advantage in

distinguishing is

$$\begin{aligned} & |P(\mathsf{Dis}(1^k, view_{conf_{\mathsf{s},1},k}) = 1) - P(\mathsf{Dis}(1^k, view_{conf_{\mathsf{s},2},k}) = 1)| \\ = \; & |P(run_{conf_{\mathsf{s},1},k}\lceil_S \notin Req(S)) - P(run_{conf_{\mathsf{s},2},k}\lceil_S \notin Req(S))|. \end{aligned}$$

If the difference were negligible, then the first term would have to be negligible because the second term is and $NEGL$ is closed under addition. Again this yields the desired contradiction. ∎

3.3 Logic Derivations

In order to apply this theorem to integrity properties formulated in a logic, e.g., temporal logic, we have to show that abstract derivations in the logic are valid with respect to the cryptographic sense. This can be proven similar to the version with synchronous time, we only include it for reasons of completeness.

Theorem 3.2 Let Sys be a system, and Req_1, Req_2 be integrity properties for Sys. Then the following holds:

a) If $Sys \models Req_1$ and $Req_1 \subseteq Req_2$, then also $Sys \models Req_2$.

b) If $Sys \models Req_1$ and $Sys \models Req_2$, then also $Sys \models Req_1 \cap Req_2$.

Here "$\subseteq$" and "'$\cap$" are interpreted pointwise, i.e., for each S. This holds in the perfect and statistical sense, and in the computational sense if for a) membership in $Req_2(S)$ is decidable in polynomial time for all S . □

Proof. Part a) is trivially fulfilled in all three cases. Part b) is trivial in the perfect case. For the statistical case and every $conf = (\hat{M}, S, \mathsf{H}, \mathsf{A}) \in \mathsf{Conf}(Sys)$,

$$\begin{aligned} & P(run_{conf,k,l(k)}\lceil_S \notin (Req_1(S) \cap Req_2(S)) \\ \leq \; & P(run_{conf,k,l(k)}\lceil_S \notin Req_1(S)) + P(run_{conf,k,l(k)}\lceil_S \notin Req_2(S)) \in SMALL \end{aligned}$$

because both summands are in $SMALL$ which is closed under addition. The computational case holds analogously because $NEGL$ is closed under addition. ∎

The first part of Theorem 3.2 resembles the Boolean "implies" operator, whereas the second part resembles the Boolean "and". We now have to show that the common deduction rules hold. For example, we consider modus ponens, i.e., if one has derived that a and $a \rightarrow b$ are valid in a given model, then b is also valid in this model. If Req_a etc. denote the semantics of the formulas, i.e., the trace sets they represent, we have to show that

$$(Sys \models Req_a \text{ and } Sys \models Req_{a \rightarrow b}) \text{ implies } Sys \models Req_b.$$

From Theorem 3.2b we conclude $Sys \models Req_a \cap Req_{a \rightarrow b}$. Obviously, $Req_a \cap Req_{a \rightarrow b} = Req_{a \wedge b} \subseteq Req_b$ holds, so the claim follows from Theorem 3.2a.

4 A Specification for Secure Message Transmission in Correct Order

In this section an abstract specification for *ordered secure message transmission* is presented, so neither reordering the messages in transit nor replay attacks are possible for the adversary. In the subsequent sections, a secure implementation for this specification is derived following the composition approach from Section 2.4. We include all definition details like ports and structures as needed for the notion of simulatability because the abstract specification is the abstract cryptographic module based on which protocols should be proved in future work. Hence it has to be defined precisely, and encoded faithfully into proof tools. We start with a brief review on standard cryptographic systems.

4.1 A Brief Review of Standard Cryptographic Systems

In real life, every user u usually has exactly one machine M_u, which is correct if and only if its user is honest. The machine M_u has special ports $\mathsf{in}_u?$ and $\mathsf{out}_u!$ for connecting to the user u. A standard cryptographic system Sys can now be derived by a trust model, which consists of an access structure $\mathcal{ACC}$ and a channel model χ. If n denotes the number of participants, then $\mathcal{ACC}$ is a set of subsets $\mathcal{H}$ of $\mathcal{M} := \{1, \ldots, n\}$ and denotes the possible sets of correct machines. For each set $\mathcal{H}$ there will be exactly one structure consisting of the machines belonging to the set $\mathcal{H}$; the remaining machines are considered part of the adversary. The channel model classifies every connection as secure (private and authentic), authenticated or insecure and derives the correspondent network connectivity. These changes can easily be done via port renaming and duplication (cf. [68]). For a fixed set $\mathcal{H}$ and a fixed channel model, we obtain a modified machine $\mathsf{M}_{u,\mathcal{H}}$ for every machine M_u with $u \in \mathcal{H}$. We denote the set of them by $\hat{M}_\mathcal{H}$ (i.e., $\hat{M}_\mathcal{H} := \{\mathsf{M}_{u,\mathcal{H}} \mid u \in \mathcal{H}\}$), so real systems are given by $Sys_{\mathsf{real}} = \{(\hat{M}_\mathcal{H}, S_\mathcal{H}) \mid \mathcal{H} \in \mathcal{ACC}\}$. Ideal systems are typically of the form $Sys_{\mathsf{id}} = \{(\{\mathsf{TH}_\mathcal{H}\}, S_\mathcal{H}) \mid \mathcal{H} \in \mathcal{ACC}\}$ with the same sets $S_\mathcal{H}$ as in the corresponding real system Sys_{real}, i.e., each structure consists of only *one* machine $\mathsf{TH}_\mathcal{H}$ that we refer to as *trusted host*.

4.2 The Abstract Specification

Given a number n of participants and a tuple L of parameters (about lengths and bounds) discussed in Section 4.2.1, our specification is a typical ideal system

$$Sys_{n,L}^{\mathsf{OSM},\mathsf{spec}} = \{(\mathsf{TH}_\mathcal{H}^{\mathsf{OSM}}, S_\mathcal{H}^{\mathsf{OSM}}) \mid \mathcal{H} \in \{1, \ldots, n\}\}$$

as described in Section 4.1, where $\mathcal{H}$ denotes the set of honest users (i.e., the access structure makes no restriction on the possible corruptions). When $\mathcal{H}$ is clear from the context, let $\mathcal{A} := \mathcal{M} \setminus \mathcal{H}$ denote the indices of corrupted machines. The ideal machine $\mathsf{TH}_\mathcal{H}^{\mathsf{OSM}}$ models initialization, sending and receiving of messages. The ports of $\mathsf{TH}_\mathcal{H}^{\mathsf{OSM}}$ intended for the users are

$$userports_\mathcal{H}^{\mathsf{OSM}} := \{\mathsf{in}_u?, \mathsf{out}_u!, \mathsf{out}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\}.$$

Intuitively each u represents one user. The ports of the users which connect to those ports are

$$S_\mathcal{H}^{\mathsf{OSM}\,c} := \{\mathsf{in}_u!, \mathsf{out}_u?, \mathsf{in}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\}.$$

For the adversary, the machine $\mathsf{TH}_\mathcal{H}^{\mathsf{OSM}}$ offers ports

$$advports_\mathcal{H}^{\mathsf{OSM}} := \{\mathsf{from\_adv}_u?, \mathsf{to\_adv}_u!, \mathsf{to\_adv}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\}.$$

Altogether, this yields

$$\mathsf{ports}(\mathsf{TH}_\mathcal{H}^{\mathsf{OSM}}) := userports_\mathcal{H}^{\mathsf{OSM}} \cup advports_\mathcal{H}^{\mathsf{OSM}}.$$

4.2.1 Lengths and Bounds

To allow a polynomial-time implementation to be as secure as this abstract specification, we use functions $\mathsf{max\_len}$, $\mathsf{max\_in\_user}$, and $\mathsf{max\_in\_adv}$ bounding the length of each message that should be transmitted, the number of inputs that $\mathsf{TH}_\mathcal{H}$ accepts from each user, and the number of inputs that $\mathsf{TH}_\mathcal{H}$ accepts from the adversary for each user, respectively. The tuple of these three functions is the system parameter L. Each function must be bounded by a polynomial and efficiently computable.

The reason for including these functions is to ensure that only a polynomial number of inputs will be processed by the machine $\mathsf{TH}_\mathcal{H}^{\mathsf{OSM}}$ independent of the environment. This is essential for applying existing results of the underlying model, in particular for the composition theorem. For real applications, one would choose these functions so large that they will never be reached.

4.2.2 States

The state of $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ consists of seven arrays:

- $(sc\_in_u^{\mathsf{OSM}})_{u\in\mathcal{H}}$ over $\{0,\ldots,\mathsf{max\_in\_user}(k)\}$ for counting the number of inputs that $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ has received at $\mathsf{in}_u?$,
- $(sc\_out_u^{\mathsf{OSM}})_{u\in\mathcal{H}}$ over $\{0,\ldots,\mathsf{max\_in\_adv}(k)\}$ for counting the number of inputs that $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ has received at $\mathsf{from\_adv}_u?$,
- $(init_{u,v}^{\mathsf{OSM}})_{u,v\in\mathcal{M}}$ over $\{0,1\}$ for modeling initialization of users, where $init_{u,u}^{\mathsf{OSM}}=1$ means that u has generated its encryption and signature key pair, and $init_{u,v}^{\mathsf{OSM}}=1$ that v has received the public keys of u,
- $(msg\_in_{u,v}^{\mathsf{OSM}})_{u\in\mathcal{H},v\in\mathcal{M}}$ over $\{0,\ldots,\mathsf{max\_in\_user}(k)\}$ for counting the number of messages sent from u to v,
- $(msg\_out_{u,v}^{\mathsf{OSM}})_{u,v\in\mathcal{H}}$ over $\{0,\ldots,\mathsf{max\_in\_adv}(k)\}$ for storing the number of the next expected message. This array is used to achieve the desired ordering (cf. the description below),
- $(stopped_u^{\mathsf{OSM}})_{u\in\mathcal{H}}$ over $\{0,1\}$ for storing whether the machine of user u has already been stopped, i.e., whether it has reached its runtime bounds (again cf. the below description),
- $(deliver_{u,v}^{\mathsf{OSM}})_{u,v\in\mathcal{H}}$ of lists for storing the actual messages.

The first six arrays are initialized with 0 everywhere, except that $msg\_out_{u,v}^{\mathsf{OSM}}$ is initialized with 1 everywhere. The last array is initialized with empty lists everywhere. Roughly, the five arrays $init_{u,v}^{\mathsf{OSM}}$, $msg\_out_{u,v}^{\mathsf{OSM}}$, $msg\_in_{u,v}^{\mathsf{OSM}}$, $stopped_u^{\mathsf{OSM}}$, and $deliver_{u,v}^{\mathsf{OSM}}$ ensure functional correctness, whereas the arrays $sc\_in_u^{\mathsf{OSM}}$ and $sc\_out_u^{\mathsf{OSM}}$ are included to allow a polynomial-time system to be as secure as this specification, cf. Section 4.2.1.

4.2.3 Inputs and their Evaluation

We now define the precise inputs and how $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ evaluates them based on its abstract state. First, the machine model contains length functions which allow to bound how many bits of input are accepted at each port, depending on the current state. The length functions are determined by the domain specified for each input in the part "for ..." after the parameter list, i.e., the overall length function for each port in each state is the maximum of the possible lengths of possible inputs in that state; it can easily be computed. In the following, we introduce commands for initialization, for sending or receiving messages and for stopping a particular machine. If these commands are entered with correct parameters at a permitted port according to the below description, we speak of *well-formed* inputs. If an input is not well-formed, we call it *trash*.

Initialization. Assume that the user u wants to generate its encryption and signature keys and distribute the corresponding public keys over authenticated channels. He can do so by sending a command $(\mathsf{snd\_init})$ to $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$. For the sake of readability, we exemplarily annotate this transition in detail.

Upon receiving on input $(\mathsf{snd\_init})$ the system checks that the user has not already reached his input bound (which is improbable in this case unless he tried to send trash all the time), and that no key generation of this user already occurred in the past. These checks correspond to $sc\_in_u^{\mathsf{OSM}} < \mathsf{max\_in\_user}(k)$, and $init_{u,u}^{\mathsf{OSM}}=0$, respectively. If at least the first check holds, the counter $sc\_in_{u,v}^{\mathsf{OSM}}$ is increased. If both checks hold, the keys are distributed over authenticated channels, modeled by an

output $(\mathsf{snd\_init})$ to the adversary which either can schedule them immediately, later or even leave them on the channels forever. Because of the asynchronous timing model, $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ has to wait for a term $(\mathsf{rec\_init}, u)$ input by the adversary at $\mathsf{from\_adv}_v?$ signaling that a connection between u and v should be established.

- *Send initialization:* $(\mathsf{snd\_init})$ at $\mathsf{in}_u?$:

  If $sc\_in_u^{\mathsf{OSM}} < \mathsf{max\_in\_user}(k)$, set $sc\_in_u^{\mathsf{OSM}} := sc\_in_u^{\mathsf{OSM}} + 1$, otherwise do nothing. If the test holds check $init_{u,u}^{\mathsf{OSM}} = 0$. In this case set $init_{u,u}^{\mathsf{OSM}} := 1$ and output $(\mathsf{snd\_init})$ at $\mathsf{to\_adv}_u!$, 1 at $\mathsf{to\_adv}_u{}^{\triangleleft}!$.

- *Receive initialization:* $(\mathsf{rec\_init}, u)$ at $\mathsf{from\_adv}_v?$ for $u \in \mathcal{M}$:

  If $stopped_v^{\mathsf{OSM}} = 0$, $init_{u,v}^{\mathsf{OSM}} = 0$, and $[u \in \mathcal{H} \Rightarrow init_{u,u}^{\mathsf{OSM}} = 1]$, set $init_{u,v}^{\mathsf{OSM}} := 1$, otherwise do nothing. If $sc\_out_u^{\mathsf{OSM}} < \mathsf{max\_in\_adv}(k)$ set $sc\_out_u^{\mathsf{OSM}} := sc\_out_u^{\mathsf{OSM}} + 1$ and output $(\mathsf{rec\_init}, u)$ at $\mathsf{out}_v!$, 1 at $\mathsf{out}_v{}^{\triangleleft}!$.

Sending and receiving messages. Sending a message m to a user v is triggered by a command (send, m, v). If v is honest, the message is stored in the array $deliver_{u,v}^{\mathsf{OSM}}$ of $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ together with the counter $msg\_in_{u,v}^{\mathsf{OSM}}$ indicating the number of the message. After that, the information $(\mathsf{send\_blindly}, i, l, v)$ is output to the adversary, where l and i denote the length of the message m and its position in the array, respectively. This models that a real-world adversary may see that a message is sent and it may even see its length. We speak of tolerable imperfections that are explicitly given to the adversary. Because of the asynchronous timing model, $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ again has to wait for a term $(\mathsf{receive\_blindly}, v, i)$ input by the adversary at $\mathsf{from\_adv}_v?$, signaling that the ith message in $deliver_{u,v}^{\mathsf{OSM}}$ should be delivered to v. Now $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ reads $(m, j) := deliver_{u,v}^{\mathsf{OSM}}[i]$ and checks whether $j \geq msg\_out_{u,v}^{\mathsf{OSM}}$ holds. This test prevents replay and message reordering. If the test is successful the message is delivered, yielding an output $(\mathsf{receive}, u, m)$ to user v, and the counter $msg\_out_{u,v}^{\mathsf{OSM}}$ is set to $j + 1$.

If v is dishonest, $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ simply outputs (send, m, v) to the adversary. The adversary can also send a message m to a user u by inputting a command $(\mathsf{receive}, v, m)$ to the port $\mathsf{from\_adv}_u?$ of $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ for a corrupted user v.

- *Send:* (send, m, v) at $\mathsf{in}_u?$ for $v \in \mathcal{M} \setminus \{u\}$, $m \in \Sigma^*$, $l := \mathsf{len}(m) \leq \mathsf{max\_len}(k)$:

  If $sc\_in_u^{\mathsf{OSM}} < \mathsf{max\_in\_user}(k)$, set $sc\_in_u^{\mathsf{OSM}} := sc\_in_u^{\mathsf{OSM}}+1$ and $msg\_in_{u,v}^{\mathsf{OSM}} := msg\_in_{u,v}^{\mathsf{OSM}}+1$, otherwise do nothing. If $init_{u,u}^{\mathsf{OSM}} = 1$ and $init_{v,u}^{\mathsf{OSM}} = 1$ holds:

  If $v \in \mathcal{A}$ then { output $(\mathsf{send}, (m, msg\_in_{u,v}^{\mathsf{OSM}}), v)$ at $\mathsf{to\_adv}_u!$, 1 at $\mathsf{to\_adv}_u{}^{\triangleleft}!$ } else {set $i := \mathsf{size}(deliver_{u,v}^{\mathsf{OSM}}) + 1$, $deliver_{u,v}^{\mathsf{OSM}}[i] := (m, msg\_in_{u,v}^{\mathsf{OSM}})$ and output $(\mathsf{send\_blindy}, i, l, v)$ at $\mathsf{to\_adv}_u!$, 1 at $\mathsf{to\_adv}_u{}^{\triangleleft}!$ }.

- *Receive from honest party u:* $(\mathsf{receive\_blindly}, u, i)$ at $\mathsf{from\_adv}_v?$ for $u \in \mathcal{H}$, $i \in \mathbb{N}$:

  If $stopped_v^{\mathsf{OSM}} = 0$, $init_{v,v}^{\mathsf{OSM}} = 1$, $init_{u,v}^{\mathsf{OSM}} = 1$, $sc\_out_v^{\mathsf{OSM}} < \mathsf{max\_in\_adv}(k)$ and $(m, j) := deliver_{u,v}^{\mathsf{OSM}}[i] \neq \downarrow$, check $j \geq msg\_out_{u,v}^{\mathsf{OSM}}$ ($j = msg\_out_{u,v}^{\mathsf{OSM}}$ in the perfect ordered system). If this holds set $sc\_out_v^{\mathsf{OSM}} := sc\_out_v^{\mathsf{OSM}} + 1$, $msg\_out_{u,v}^{\mathsf{OSM}} := j + 1$ and output $(\mathsf{receive}, u, m)$ at $\mathsf{out}_v!$, 1 at $\mathsf{out}_v{}^{\triangleleft}!$.

- *Receive from dishonest party u:* $(\mathsf{receive}, u, m)$ at $\mathsf{from\_adv}_v?$ for $u \in \mathcal{A}$, $m \in \Sigma^*$, $\mathsf{len}(m) \leq \mathsf{max\_len}(k)$:

If $stopped_v^{\mathsf{OSM}} = 0$, $init_{v,v}^{\mathsf{OSM}} = 1$, $init_{u,v}^{\mathsf{OSM}} = 1$ and $sc\_out_v^{\mathsf{OSM}} < \mathsf{max\_in\_adv}(k)$, set $sc\_out_v^{\mathsf{OSM}} := sc\_out_v^{\mathsf{OSM}} + 1$ and output $(\mathsf{receive}, u, m)$ at $\mathsf{out}_v!$, 1 at $\mathsf{out}_v{}^{\triangleleft}!$.

Stop commands. The adversary is further to cause the machine of any user u to stop processing inputs received from the network by entering a command (stop) at $\mathsf{from\_adv}_u?$. This roughly corresponds to exceeding the machine's runtime bounds in the real world.

- *Stop:* (stop) at $\mathsf{from\_adv}_u?$:
  If $stopped_u^{\mathsf{OSM}} = 0$ and $sc\_out_v^{\mathsf{OSM}} < \mathsf{max\_in\_adv}$, set $stopped_u^{\mathsf{OSM}} := 1$ and output (stop) at $\mathsf{out}_u!$, 1 at $\mathsf{out}_u{}^{\triangleleft}!$.

Trash inputs. Finally, if $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ receives an input at a port $\mathsf{in}_u?$ which is not comprised by the above transitions (i.e., the user sends some kind of trash), it increases the counter $sc\_in_u^{\mathsf{OSM}}$. Similarly, if $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ receives such an input at a port $\mathsf{from\_adv}_v?$ it increases the counter $sc\_out_v^{\mathsf{OSM}}$.

$Sys_{n,L}^{\mathsf{OSM,spec}}$ is as abstract as we hoped for. It is deterministic without containing any cryptographic objects. Furthermore it is simple, so that its state-transition function can easily by expressed in formal languages, e.g., in PVS. In the following we write $Sys^{\mathsf{OSM,spec}}$ instead of $Sys_{n,L}^{\mathsf{OSM,spec}}$ if the parameters n and L are not necessary for understanding.

4.3 The Security Property

Our goal is to prove that message reordering in $Sys_{n,L}^{\mathsf{OSM,spec}}$ is not possible for the adversary. Formally, this means that for $u, v \in \mathcal{H}$, the messages that v received from u via $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ always have to be a sublist of those messages that u sent to v. The former list is called the *receive-list*, the latter the *send-list*. More formally, this means that for $u, v \in \mathcal{H}$, a trace tr, and a point t in time, we define the send-list $send\_list_{u,v}^{tr}(t)$ at time t as follows:

1. The trace tr is first restricted to inputs at $\mathsf{in}_u?$.
2. The resulting sub-trace is further restricted to inputs of the form (send, m, v) with $\mathsf{len}(m) \leq \mathsf{max\_len}(k)$.
3. Finally, every element (send, m, v) is replaced by m.

Similarly, the receive-list $recv\_list_{u,v}^{tr}(t)$ at time t is defined as follows:

1. The trace tr is first restricted to outputs at $\mathsf{out}_u!$.
2. The resulting sub-trace is further restricted to outputs of the form $(\mathsf{receive}, u, m)$ with $\mathsf{len}(m) \leq \mathsf{max\_len}(k)$.
3. Finally, every element $(\mathsf{receive}, u, m)$ is replaced by m.

We are now ready to introduce the desired integrity property req^{OSM}, which we call *ordering property*:

Definition 4.1 (Ordering Property) *Let $S_{\mathcal{H}}^{\mathsf{OSM}}$ be the specified ports of $Sys_{n,L}^{\mathsf{OSM,spec}}$ as defined in Section 4.2. Then a trace tr is contained in $Req^{\mathsf{OSM}}(S_{\mathcal{H}}^{\mathsf{OSM}})$ if for all $u, v \in \mathcal{H}$ and any time t:*

$$recv\_list_{u,v}^{tr}(t) \subseteq send\_list_{u,v}^{tr}(t),$$

where "$\subseteq$" is the sublist relation. $\diamond$

The following theorem finally captures the security of the system $Sys_{n,L}^{\mathsf{OSM,spec}}$ with respect to the ordering property.

Theorem 4.1 (Ordering Property of the Ideal System for Ordered Secure Message Transmission) *Let $Sys_{n,L}^{\mathsf{OSM,spec}}$ be the ideal system for ordered secure message transmission defined in Section 4.2 for arbitrary parameters n, L, and let Req^{OSM} be the integrity property of Definition 4.1. Then $Sys_{n,L}^{\mathsf{OSM,spec}} \models^{\mathsf{perf}} Req^{\mathsf{OSM}}$.* □

We will prove this theorem in Section 8 by means of the theorem proving system PVS.

5 The Cryptographic Implementation

In this section, we derive a possible implementation of the proposed specification, and we will prove this implementation to be secure in the subsequent sections. We start with the definition of an intermediate, called *hybrid* system in Sections 5.1-5.3. If we take a look at Figure 3, the system $Sys^{\mathsf{OSM,spec}}$ plays the role of the monolithic specification Sys^{spec}. We now split our specification into a system $Sys^{\mathsf{OSM,hybr}}$ (corresponding to Sys^{*} in Figure 3) such that $Sys^{\mathsf{OSM,hybr}} \geq_{\mathsf{sec}} Sys^{\mathsf{OSM,spec}}$ holds. $Sys^{\mathsf{OSM,hybr}}$ is the combination of two systems Sys^{filt} and $Sys^{\mathsf{SM,spec}}$. The system $Sys^{\mathsf{SM,spec}}$ is the ideal system for secure unordered message transmission presented in [68], and the system Sys^{filt} will filter messages that are out of order. Finally, replacing the subsystem $Sys^{\mathsf{SM,spec}}$ with the concrete system for secure message transmission $Sys^{\mathsf{SM,real}}$ from [68] and using the composition theorem yields a concrete system $Sys^{\mathsf{OSM,real}}$ that is as secure as $Sys^{\mathsf{OSM,spec}}$.

We start with the definition of the filtering system.

5.1 The Filtering System

Given a number n of participants and the tuple L of functions as introduced in Section 4.2.1, the filtering system is given by

$$Sys_{n,L}^{\mathsf{filt}} = \{(\hat{M}_{\mathcal{H}}^{\mathsf{filt}}, S_{\mathcal{H}}^{\mathsf{filt}}) \mid \mathcal{H} \subseteq \{1, \ldots, n\}\},$$

where $\hat{M}_{\mathcal{H}}^{\mathsf{filt}} := \{\mathsf{M}_u^{\mathsf{filt}} \mid u \in \mathcal{H}\}$ and $\mathsf{ports}(\mathsf{M}_u^{\mathsf{filt}}) := \{\mathsf{in}_u?, \mathsf{out}_u!, \mathsf{out}_u{}^{\triangleleft}!\} \cup \{\mathsf{in}_u^{\mathsf{filt}}?, \mathsf{out}_u^{\mathsf{filt}}!, \mathsf{out}_u^{\mathsf{filt}\triangleleft}!\}$. All free ports of $[\hat{M}_{\mathcal{H}}^{\mathsf{filt}}]$ are specified, i.e., $S_{\mathcal{H}}^{\mathsf{filt}}$ consists of all ports corresponding to $\mathsf{ports}(\hat{M}_{\mathcal{H}}^{\mathsf{filt}})$.

5.1.1 States

Each machine $\mathsf{M}_u^{\mathsf{filt}}$ maintains two arrays and three variables, whose meanings follow closely from the description of the state of $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ introduced in Section 4.2.2:

- $(msg\_in_{u,v}^{\mathsf{filt}})_{v \in \mathcal{M}}$ over $\{0, \ldots, \mathsf{max\_in\_user}(k)\}$,
- $(msg\_out_{v,u}^{\mathsf{filt}})_{v \in \mathcal{M}}$ over $\{0, \ldots, \mathsf{max\_in\_adv}(k)\}$,
- $sc\_in_u^{\mathsf{filt}}$ over $\{0, \ldots, \mathsf{max\_in\_user}(k)\}$,
- $sc\_out_u^{\mathsf{filt}}$ over $\{0, \ldots, \mathsf{max\_in\_adv}(k)\}$,
- $stopped_u^{\mathsf{filt}}$ over $\{0, 1\}$.

Both arrays should be initialized with 0 everywhere, and the three counters should be initially 0.

5.1.2 Inputs and their Evaluation

The length functions for each port of each machine $\mathsf{M}_u^{\mathsf{filt}}$ are defined similarly to Section 4.2.3, i.e., they are determined by the domain specified for each input in the part "for ..." after the parameter list. However, if the counter $sc\_in_u^{\mathsf{filt}}$ reaches the bound $\mathsf{max\_in\_user}(k)$, $sc\_out_u^{\mathsf{filt}}$ reaches the bound $\mathsf{max\_in\_adv}(k)$, or $stopped_u^{\mathsf{filt}} = 1$ holds, we use different bounds to ensure polynomial runtime of the system. These bounds are introduced in Section 5.1.3.

We further assume that encoding of tuples has the following straightforward length property: $\mathsf{len}((m, num)) = \mathsf{len}(m) + c(k)$ for every $num \in \{0, \ldots, \max\{\mathsf{max\_in\_user}(k), \mathsf{max\_in\_adv}(k)\}\}$ and an arbitrary polynomially bounded function c, i.e., $\mathsf{len}(num)$ is constant for each fixed security parameter k. This condition can easily be achieved by padding all values num to a fixed size $\geq \mathsf{len}(\max\{\mathsf{max\_in\_user}(k), \mathsf{max\_in\_adv}(k)\})$. Now the behavior of $\mathsf{M}_u^{\mathsf{filt}}$ is defined as follows.

Initialization.

- *Send initialization:* $(\mathsf{snd\_init})$ at $\mathsf{in}_u?$:

  If $sc\_in_u^{\mathsf{filt}} < \mathsf{max\_in\_user}(k)$, set $sc\_in_u^{\mathsf{filt}} := sc\_in_u^{\mathsf{filt}} + 1$ and output $(\mathsf{snd\_init})$ at $\mathsf{out}_u^{\mathsf{filt}}!$, 1 at $\mathsf{out}_u^{\mathsf{filt}\triangleleft}!$.

- *Receive initialization:* $(\mathsf{rec\_init}, v)$ at $\mathsf{in}_u^{\mathsf{filt}}?$ for $v \in \mathcal{M}$:

  If $stopped_u^{\mathsf{filt}} = 0$ and $sc\_out_u^{\mathsf{filt}} < \mathsf{max\_in\_adv}(k)$, set $sc\_out_u^{\mathsf{filt}} := sc\_out_u^{\mathsf{filt}} + 1$ and output $(\mathsf{rec\_init}, v)$ at $\mathsf{out}_u!$, 1 at $\mathsf{out}_u{}^{\triangleleft}!$.

Sending and receiving messages.

- *Send:* (send, m, v) at $\mathsf{in}_u?$ for $v \in \mathcal{M} \setminus \{u\}$, $m \in \Sigma^*$, $\mathsf{len}(m) \leq \mathsf{max\_len}(k)$:

  If $sc\_in_u^{\mathsf{filt}} < \mathsf{max\_in\_user}(k)$, set $sc\_in_u^{\mathsf{filt}} := sc\_in_u^{\mathsf{filt}} + 1$, $msg\_in_{u,v}^{\mathsf{filt}} := msg\_in_{u,v}^{\mathsf{filt}} + 1$ and output $(\mathsf{send}, (m, msg\_in_{u,v}^{\mathsf{filt}}), v)$ at $\mathsf{out}_u^{\mathsf{filt}}!$, 1 at $\mathsf{out}_u^{\mathsf{filt}\triangleleft}!$.

- *Receive:* $(\mathsf{receive}, v, m')$ at $\mathsf{in}_u^{\mathsf{filt}}?$ for $v \in \mathcal{M}$, $m' \in \Sigma^*$, $\mathsf{len}(m') \leq \mathsf{max\_len}(k) + c(k)$:

  If $stopped_u^{\mathsf{filt}} = 0$ and $sc\_out_u^{\mathsf{filt}} < \mathsf{max\_in\_adv}(k)$, decompose the message m' into (m, num). If $num \geq msg\_out_{v,u}^{\mathsf{filt}}$ (or $num = msg\_out_{v,u}^{\mathsf{filt}}$ in the perfect ordered system), set $sc\_out_u^{\mathsf{filt}} := sc\_out_u^{\mathsf{filt}} + 1$, $msg\_out_{v,u}^{\mathsf{filt}} := num + 1$ and output $(\mathsf{receive}, v, m)$ at $\mathsf{out}_u!$, 1 at $\mathsf{out}_u{}^{\triangleleft}!$.

Stop commands.

- *Stop:* (stop) at $\mathsf{in}_u^{\mathsf{filt}}?$:

  If $stopped_u^{\mathsf{filt}} = 0$ and $sc\_out_u^{\mathsf{filt}} < \mathsf{max\_in\_adv}(k)$, set $stopped_u^{\mathsf{filt}} := 1$ and output (stop) at $\mathsf{out}_u!$, 1 at $\mathsf{out}_u{}^{\triangleleft}!$.

Trash inputs. Finally, if $\mathsf{M}_u^{\mathsf{filt}}$ receives an input at a port $\mathsf{in}_u?$ which is not comprised by the above transitions, it increases the counter $sc\_in_u^{\mathsf{filt}}$. Similarly, if $\mathsf{M}_u^{\mathsf{filt}}$ receives such an input at port $\mathsf{in}_u^{\mathsf{filt}}?$ it increases the counter $sc\_out_u^{\mathsf{filt}}$.

5.1.3 On Polynomial Runtime

In order to apply existing results of the underlying model, in particular the composition theorem, the system $Sys^{\mathsf{filt}}_{n,L}$ must be polynomial-time, i.e., every machine $\mathsf{M}^{\mathsf{filt}}_u$ must be polynomial-time. Note that each input at port $\mathsf{in}_u?$ checks if $sc\_in^{\mathsf{filt}}_u < \mathsf{max\_in\_user}(k)$ holds, doing nothing at failure. In case of a successful check, $\mathsf{M}^{\mathsf{filt}}_u$ increases $sc\_in^{\mathsf{filt}}_u$. Similar reasoning holds for the port $\mathsf{in}^{\mathsf{filt}}_u?$ with $sc\_out^{\mathsf{filt}}_u$ and $\mathsf{max\_in\_adv}(k)$, where additionally $stopped^{\mathsf{filt}}_u = 0$ is checked and maybe $stopped^{\mathsf{filt}}_u = 1$ is set. This means that only a polynomial number of inputs lead to a state change or a non-empty output. However, since the machine still has to read its input to perform the mentioned checks, this is not yet sufficient for polynomial runtime. We therefore use the length functions of the underlying model to "cut off" an input port as soon as a corresponding counter has reached its limit.

More formally, the value 0 for the length function for a port $\mathsf{p}?$ means that no input is accepted (without a Turing step) at $\mathsf{p}?$. This means that whenever the counter $sc\_in^{\mathsf{filt}}_u$ reaches the bound $\mathsf{max\_in\_user}(k)$ or $sc\_out^{\mathsf{filt}}_u$ reaches the bound $\mathsf{max\_in\_adv}(k)$, the length function for the port $\mathsf{in}_u?$ respectively $\mathsf{in}^{\mathsf{filt}}_u?$ is always zero. Similarly, if $stopped^{\mathsf{filt}}_u = 1$ then the length function for $\mathsf{in}^{\mathsf{filt}}_u?$ is zero. Note that this does not affect the functional behavior of the machine $\mathsf{M}^{\mathsf{filt}}_u$ since the port $\mathsf{in}_u?$ is only cut off if no further input at $\mathsf{in}_u?$ can cause $\mathsf{M}^{\mathsf{filt}}_u$ to change its state or produce a non-empty output, similarly for the port $\mathsf{in}^{\mathsf{filt}}_u?$.

Lemma 5.1 *The system $Sys^{\mathsf{filt}}_{n,L}$ is polynomial-time for all parameters n, L.* □

Proof. Each transition of each $\mathsf{M}^{\mathsf{filt}}_u$ can surely be realized in polynomial time, since the length bounds only read a polynomially bounded number of bits in each transition. Moreover, non-empty inputs at $\mathsf{in}_u?$ can only occur if $sc\_in^{\mathsf{filt}}_u < \mathsf{max\_in\_user}(k)$; if this condition does not hold, the length function for $\mathsf{in}_u?$ is explicitly defined to be zero. If the check succeeds, each transition increases the counter $sc\_in^{\mathsf{filt}}_u$, hence there can at most be $\mathsf{max\_in\_user}(k)$ inputs at $\mathsf{in}_u?$. Similarly, non-empty inputs at $\mathsf{in}^{\mathsf{filt}}_u?$ can only occur if $stopped^{\mathsf{filt}}_u = 0$ and $sc\_out^{\mathsf{filt}}_u < \mathsf{max\_in\_adv}(k)$, and each transition in this case either increases the counter $sc\_out^{\mathsf{filt}}_u$ or sets $stopped^{\mathsf{filt}}_u = 1$. Hence there are at most $\max\{\mathsf{max\_in\_adv}(k), 1\}$ inputs at $\mathsf{in}^{\mathsf{filt}}_u?$, i.e., a polynomial number of inputs total, which finishes the proof. ■

5.2 The Ideal System for Unordered Secure Message Transmission

As described above, the system $Sys^{\mathsf{SM,spec}}$ is the ideal system for secure unordered message transmission of [68]. We now describe it in full because we need it for our security proof in Sections 6 and 7. We made a few adaptations (in particular renaming the ports intended for the users), which do not invalidate the proof.

Let n denote the number of participants. Similar to the system for ordered secure message transmission, the system for secure unordered message transmission has a parameter $\mathsf{max\_len}$ bounding the permitted message length. Then the ideal system for secure unordered message transmission is given by

$$Sys^{\mathsf{SM,spec}}_{n,\mathsf{max\_len}} = \{(\mathsf{TH}^{\mathsf{SM}}_{\mathcal{H}}, S^{\mathsf{SM}}_{\mathcal{H}}) \mid \mathcal{H} \subseteq \{1, \ldots, n\}\},$$

with $\mathsf{ports}(\mathsf{TH}^{\mathsf{SM}}_{\mathcal{H}}) := \{\mathsf{out}^{\mathsf{filt}}_u?, \mathsf{in}^{\mathsf{filt}}_u!, \mathsf{in}^{\mathsf{filt}\triangleleft}_u!, \mathsf{from\_adv}_u?, \mathsf{to\_adv}_u!, \mathsf{to\_adv}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\}$. If $\mathcal{H}$ is clear from the context, let again $\mathcal{A} := \mathcal{M} \setminus \mathcal{H}$. The ports of the users which connect to those ports are

$$S^{\mathsf{SM}c}_{\mathcal{H}} := \{\mathsf{in}^{\mathsf{filt}}_u?, \mathsf{out}^{\mathsf{filt}}_u!, \mathsf{out}^{\mathsf{filt}\triangleleft}_u! \mid u \in \mathcal{H}\}.$$

5.2.1 States

The machine $\mathsf{TH}_{\mathcal{H}}^{\mathsf{SM}}$ maintains three arrays, whose meanings should already be clear from the description of the state of $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$.

- $(init_{u,v}^{\mathsf{SM}})_{u,v\in\mathcal{M}}$ over $\{0,1\}$ initialized with 0 everywhere.
- $(stopped_{u}^{\mathsf{SM}})_{u\in\mathcal{H}}$ over $\{0,1\}$ initialized with 0 everywhere,
- $(deliver_{u,v}^{\mathsf{SM}})_{u,v\in\mathcal{H}}$ of lists, all initially empty.

5.2.2 Inputs and their Evaluation

The length functions of the machine $\mathsf{TH}_{\mathcal{H}}^{\mathsf{SM}}$ are defined similarly to Section 4.2.3, i.e., they are determined by the respective domains. The state-transition function of $\mathsf{TH}_{\mathcal{H}}$ is defined by the following rules:

Initialization.

- *Send initialization:* $(\mathsf{snd\_init})$ at $\mathsf{out}_u^{\mathsf{filt}}?$:
  If $init_{u,u}^{\mathsf{SM}} = 0$, set $init_{u,u}^{\mathsf{SM}} := 1$ and output $(\mathsf{snd\_init})$ at $\mathsf{to\_adv}_u!$, 1 at $\mathsf{to\_adv}_u{}^{\triangleleft}!$.
- *Receive initialization:* $(\mathsf{rec\_init}, u)$ at $\mathsf{from\_adv}_v?$ for $u \in \mathcal{M}$:
  If $stopped_v^{\mathsf{SM}} = 0$ and $init_{u,v}^{\mathsf{SM}} = 0$ and $[u \in \mathcal{H} \Rightarrow init_{u,u}^{\mathsf{SM}} = 1]$, set $init_{u,v}^{\mathsf{SM}} := 1$ and output $(\mathsf{rec\_init}, u)$ at $\mathsf{in}_v^{\mathsf{filt}}!$, 1 at $\mathsf{in}_v^{\mathsf{filt}^{\triangleleft}}!$.

Sending and receiving messages.

- *Send:* (send, m, v) at $\mathsf{out}_u^{\mathsf{filt}}?$ for $v \in \mathcal{M} \setminus \{u\}$, $m \in \Sigma^*$, $l := \mathsf{len}(m) \leq \mathsf{max\_len}(k) + c(k)$:
  If $init_{u,u}^{\mathsf{SM}} = 1$, and $init_{v,u}^{\mathsf{SM}} = 1$:
  If $v \in \mathcal{A}$ then { output (send, m, v) at $\mathsf{to\_adv}_u!$, 1 at $\mathsf{to\_adv}_u{}^{\triangleleft}!$ }, else $\{i := \mathsf{size}(deliver_{u,v}^{\mathsf{SM}}) + 1$; $deliver_{u,v}^{\mathsf{SM}}[i] := m$; output $(\mathsf{send\_blindly}, i, l, v)$ at $\mathsf{to\_adv}_u!$, 1 at $\mathsf{to\_adv}_u{}^{\triangleleft}!$ $\}$.
- *Receive from honest party u:* $(\mathsf{receive\_blindly}, u, i)$ at $\mathsf{from\_adv}_v?$ for $u \in \mathcal{H}$, $i \in \mathbb{N}$:
  If $stopped_v^{\mathsf{SM}} = 0$, $init_{v,v}^{\mathsf{SM}} = 1$, $init_{u,v}^{\mathsf{SM}} = 1$, and $m := deliver_{u,v}^{\mathsf{SM}}[i] \neq \downarrow$, then output $(\mathsf{receive}, u, m)$ at $\mathsf{in}_v^{\mathsf{filt}}!$, 1 at $\mathsf{in}_v^{\mathsf{filt}^{\triangleleft}}!$.
- *Receive from dishonest party u:* $(\mathsf{receive}, u, m)$ at $\mathsf{from\_adv}_v?$ for $u \in \mathcal{A}$, $m \in \Sigma^*$, $\mathsf{len}(m) \leq \mathsf{max\_len}(k)$:
  If $stopped_v^{\mathsf{SM}} = 0$, $init_{v,v}^{\mathsf{SM}} = 1$ and $init_{u,v}^{\mathsf{SM}} = 1$, then output $(\mathsf{receive}, u, m)$ at $\mathsf{in}_v^{\mathsf{filt}}!$, 1 at $\mathsf{in}_v^{\mathsf{filt}^{\triangleleft}}!$.

Stop commands.

- *Stop:* (stop) at $\mathsf{from\_adv}_u?$:
  If $stopped^{\mathsf{SM}} = 0$, set $stopped_u^{\mathsf{SM}} = 1$ and output (stop) at $\mathsf{in}_u^{\mathsf{filt}}!$, 1 at $\mathsf{in}_u^{\mathsf{filt}^{\triangleleft}}!$.

Trash inputs. $\mathsf{TH}_{\mathcal{H}}^{\mathsf{SM}}$ simply ignores trash inputs at every port.

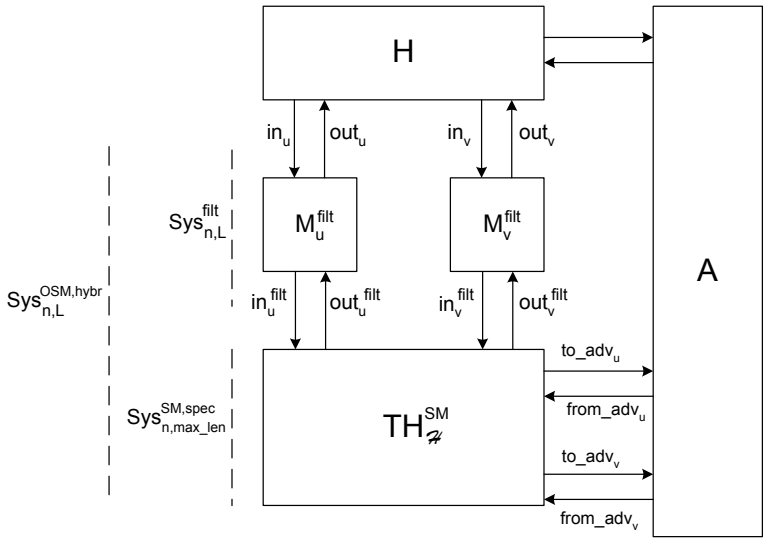

Figure 5: The Hybrid System.

5.3 The Hybrid System

We now combine the two systems $Sys^{\mathsf{SM,spec}}_{n,\mathsf{max\_len}}$ and $Sys^{\mathsf{filt}}_{n,L}$ in the "canonical" way, i.e., we combine those structures with the same index $\mathcal{H}$. We further restrict ourselves to the case, where the parameter $\mathsf{max\_len}$ of $Sys^{\mathsf{SM,spec}}_{n,\mathsf{max\_len}}$ is equal to the respective message length function of the parameter L (cf. Section 4.2.1). This yields a system $Sys^{\mathsf{OSM,hybr}}_{n,L}$, which we call *hybrid system for ordered secure message transmission*. It is depicted in Figure 5. The specified ports of the hybrid system for $\mathcal{H}$ are then given by $\{\mathsf{out}_u?, \mathsf{in}_u!, \mathsf{in}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\}^c$, i.e., they are equal to the specified ports $S^{\mathsf{OSM}}_{\mathcal{H}}$ of the specification. Finally, we define all connections $\{\mathsf{out}^{\mathsf{filt}}_u!, \mathsf{out}^{\mathsf{filt}}_u?\}$ and $\{\mathsf{in}^{\mathsf{filt}}_u!, \mathsf{in}^{\mathsf{filt}}_u?\}$ of $Sys^{\mathsf{OSM,hybr}}_{n,L}$ to be secure, because they correspond to local subroutine calls.

5.4 The Real System

The concrete system $Sys^{\mathsf{OSM,real}}_{n,L,\mathcal{E},\mathcal{S}}$ is derived by replacing $Sys^{\mathsf{SM,spec}}_{n,\mathsf{max\_len}}$ with $Sys^{\mathsf{SM,real}}_{n,\mathsf{max\_len},\mathcal{E},\mathcal{S}}$, which is the concrete implementation of $Sys^{\mathsf{SM,spec}}_{n,\mathsf{max\_len}}$ as introduced in [68]. For understanding it is sufficient to give a brief review of $Sys^{\mathsf{SM,real}}_{n,\mathsf{max\_len},\mathcal{E},\mathcal{S}}$. It is a standard cryptographic system of the form $Sys^{\mathsf{SM,real}}_{n,\mathsf{max\_len},\mathcal{E},\mathcal{S}} = \{(\hat{M}^{\mathsf{SM}}_{\mathcal{H}}, S^{\mathsf{SM}}_{\mathcal{H}}) \mid \mathcal{H} \in \{1,\ldots,n\}\}$, cf. Section 4.1, where n denotes the number of participants, i.e., any subset of participants may be dishonest; $\mathsf{max\_len}$ is the usual bound on the message length, which we defined to be equal to the message length function in L. The system uses an asymmetric encryption scheme $\mathcal{E}$ and a digital signature scheme $\mathcal{S}$ as cryptographic primitives, which are additional parameters of the system. A user u can let his machine create signature and encryption keys that are sent to other users over authenticated channels. Messages sent from user u to user v are signed and encrypted by M_u and sent to M_v over an insecure channel, representing a real network. The adversary can schedule the communication between correct machines and send arbitrary messages m to arbitrary users. He can also replay messages and also change their order, which is prevented in our scheme by the additional filtering system.

We now build the combination of $Sys^{\mathsf{SM,real}}_{n,\mathsf{max\_len},\mathcal{E},\mathcal{S}}$ and $Sys^{\mathsf{filt}}_{n,L}$ again in the canonical way, which yields a new system $Sys^{\mathsf{OSM,real}}_{n,L,\mathcal{E},\mathcal{S}}$ that we refer to as the *real system for ordered secure message transmission*. It is depicted in Figure 6.

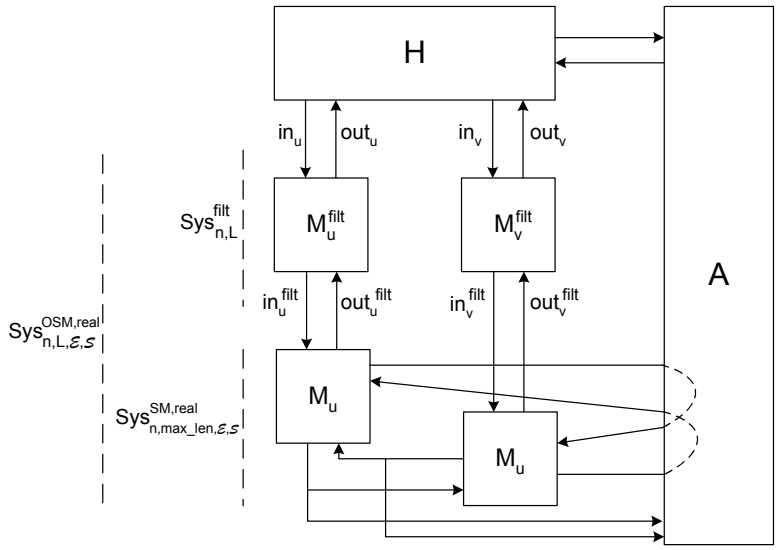

Figure 6: Sketch of the Real System for Ordered Secure Message Transmission.

6 Proving Security of the Real Ordered System

In this section, we start to prove that the real ordered system $Sys^{\mathsf{OSM,real}}_{n,L,\mathcal{E},\mathcal{S}}$ is at least as secure as the specification $Sys^{\mathsf{OSM,spec}}_{n,L}$ provided that the encryption and signature system used are secure. We further show that $Sys^{\mathsf{OSM,real}}_{n,L,\mathcal{E},\mathcal{S}}$ computationally fulfills the integrity property of Definition 4.1.

6.1 The Simulatability Property

We start with the simulatability property, which is captured in the following theorem.

Theorem 6.1 *(Security of Real Ordered Secure Message Transmission)* We have $Sys^{\mathsf{OSM,real}}_{n,L,\mathcal{E},\mathcal{S}} \geq^{\mathsf{poly}}_{\mathsf{sec}} Sys^{\mathsf{OSM}}_{n,L}$ for all parameters $n, L, \mathcal{E}, \mathcal{S}$ (and for the canonical mapping), provided the signature and encryption schemes used are secure. This holds with blackbox simulatability.[1] □

The proof is split into four steps, which can be illustrated in Figure 3:

1. First, [68] contains the result $Sys^{\mathsf{SM,real}}_{n,\mathsf{max\_len},\mathcal{E},\mathcal{S}} \geq^{\mathsf{poly}}_{\mathsf{sec}} Sys^{\mathsf{SM,spec}}_{n,\mathsf{max\_len}}$.
2. Secondly, the composition theorem (cf. Section 2.4) yields the relation $Sys^{\mathsf{OSM,real}}_{n,L,\mathcal{E},\mathcal{S}} \geq^{\mathsf{poly}}_{\mathsf{sec}} Sys^{\mathsf{OSM,hybr}}_{n,L}$. The only remaining task is to check that its preconditions are fulfilled, which is straightforward since we showed that the system $Sys^{\mathsf{filt}}_{n,L}$ is polynomial-time in Lemma 5.1.
3. Thirdly, we prove $Sys^{\mathsf{OSM,hybr}}_{n,L} \geq^{\mathsf{poly}}_{\mathsf{sec}} Sys^{\mathsf{OSM,spec}}_{n,L}$.
4. Finally, $Sys^{\mathsf{OSM,real}}_{n,L,\mathcal{E},\mathcal{S}} \geq^{\mathsf{poly}}_{\mathsf{sec}} Sys^{\mathsf{OSM,spec}}_{n,L}$ follows from the transitivity lemma, cf. Section 2.1.

Thus, we only have to prove $Sys^{\mathsf{OSM,hybr}}_{n,L} \geq^{\mathsf{poly}}_{\mathsf{sec}} Sys^{\mathsf{OSM,spec}}_{n,L}$. We will even prove the perfect case $Sys^{\mathsf{OSM,hybr}}_{n,L} \geq^{\mathsf{perf}}_{\mathsf{sec}} Sys^{\mathsf{OSM,spec}}_{n,L}$, which is separately captured in the following lemma:

[1] See [68] for further details on canonical mappings and different kinds of simulatability.

Lemma 6.1 *For all parameters n, L, we have $Sys_{n,L}^{\mathsf{OSM,hybr}} \geq_{\mathsf{sec}}^{\mathsf{perf}} Sys_{n,L}^{\mathsf{OSM,spec}}$ (for the canonical mapping), and with blackbox simulatability.* □

In order to prove this, we assume a configuration $conf_{\mathsf{hybr}} := (\{\mathsf{TH}_{\mathcal{H}}^{\mathsf{SM}}\} \cup \hat{M}_{\mathcal{H}}^{\mathsf{filt}}, S_{\mathcal{H}}, \mathsf{H}, \mathsf{A})$ of $Sys_{n,L}^{\mathsf{OSM,hybr}}$ with $\hat{M}_{\mathcal{H}}^{\mathsf{filt}} = \{\mathsf{M}_u^{\mathsf{filt}} \mid u \in \mathcal{H}\}$ to be given, which we call a *hybrid configuration*. We then have to show that there exists a configuration $conf_{\mathsf{spec}} := (\{\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}\}, S_{\mathcal{H}}, \mathsf{H}, \mathsf{A}')$ of $Sys_{n,L}^{\mathsf{OSM,spec}}$, called a *specification configuration*, yielding indistinguishable views for the honest user H.

The adversary A' consists of two machines: a so-called simulator $\mathsf{Sim}_{\mathcal{H}}$, which we define in the following, and the original adversary A. This is exactly the notion of blackbox simulatability. These configurations are shown in Figure 7. We will now first give some preliminaries of the proof of Lemma 6.1, and give a rigorous definition of the simulator afterwards.

6.1.1 Preliminaries for Proving Lemma 6.1

Given a hybrid configuration and a specification configuration as defined above, the ultimate goal is to show that the collections $\hat{M}_{\mathsf{hybr}} := \{\mathsf{TH}_{\mathcal{H}}^{\mathsf{SM}}\} \cup \{\mathsf{M}_u^{\mathsf{filt}} \mid u \in \mathcal{H}\}$ and $\hat{M}_{\mathsf{spec}} := \{\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}, \mathsf{Sim}_{\mathcal{H}}\}$ have the same input-output behavior, i.e., if they obtain the same inputs they produce the same outputs. We do so by proving a deterministic bisimulation, i.e., we define a relation ϕ on the states of the two collections and show that ϕ is maintained in every step of every trace and that the outputs of both systems are always equal. This is exactly the procedure we will perform in the next section using the theorem prover PVS.

Definition 6.1 *(Deterministic Bisimulation) Let two arbitrary collections $\hat{M}_1$ and $\hat{M}_2$ of deterministic machines with identical sets of free ports be given, i.e., $\mathsf{free}([\hat{M}_1]) = \mathsf{free}([\hat{M}_2])$. A deterministic bisimulation between these two collections is a binary relation ϕ on the states of $\hat{M}_1$ and $\hat{M}_2$ such that the following holds.*

- *The initial states of $\hat{M}_1$ and $\hat{M}_2$ satisfy the relation ϕ.*
- *The transition functions δ_1 and δ_2 of $\hat{M}_1$ and $\hat{M}_2$ preserve the relation ϕ and produce identical outputs. I.e., let S_1 and S_2 be two states of $\hat{M}_1$ and $\hat{M}_2$, respectively, with $(S_1, S_2) \in \phi$, let $\mathcal{I}$ be an arbitrary overall input of $\hat{M}_1$ and $\hat{M}_2$, and let $(S_1', \mathcal{O}_1) := \delta_1(S_1, \mathcal{I})$ and $(S_2', \mathcal{O}_2) := \delta_2(S_2, \mathcal{I})$. Then we have $(S_1', S_2') \in \phi$ and $\mathcal{O}_1 = \mathcal{O}_2$.*

We call two collections $\hat{M}_1$ and $\hat{M}_2$ bisimilar if there exists a deterministic bisimulation between them. ◇

We will apply this definition to composed transition functions of each of the two collections $\hat{M}_{\mathsf{hybr}}$ and $\hat{M}_{\mathsf{spec}}$, i.e., the overall transition from an external input (from H or A) to an external output (to H or A). It is quite easy to see that a deterministic bisimulation in this sense implies perfect indistinguishability of the view of H, cf. Figure 7, and even of the joint view of H and the original adversary A. Assume for contradiction that these views are not identical. Thus, there exists a first time where they can be distinguished. This difference has to be produced by the collections. Since we defined this to be the first different step, the prior input of both collections is identical. But thus, both collections also produce identical outputs because they are bisimilar. This yields the desired contradiction.

The next section describes how the machines are expressed in the formal syntax of PVS and partly explains the bisimulation proof, which then finishes the proof of Lemma 6.1, and hence also the simulatability proof of Theorem 6.1.

It is worth mentioning that we used standard paper-and-pencil proofs before we decided to use a formal proof system to validate the desired bisimulation. However, these proofs have turned out to be prone to error since they are straightforward on the one hand, but long and tedious on the other, so they

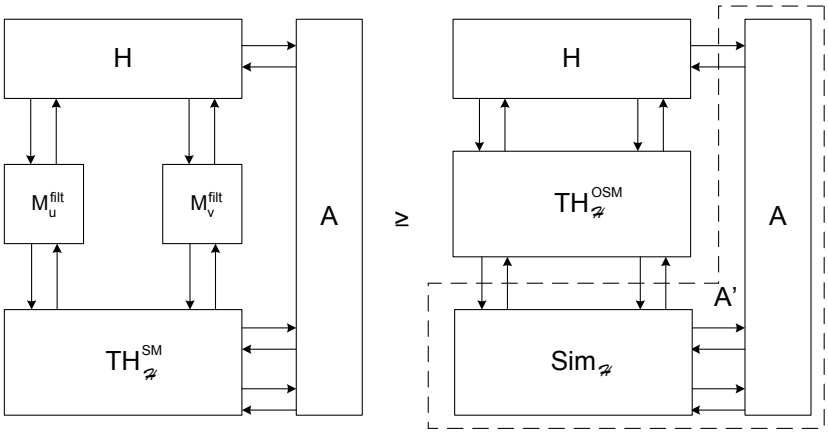

Figure 7: Proof Overview of $Sys_{n,L}^{\mathsf{OSM,hybr}} \geq_{\mathsf{sec}}^{\mathsf{perf}} Sys_{n,L}^{\mathsf{OSM,spec}}$.

are mainly vulnerable to slow-down of concentration. During our formal verification, we in fact found several errors in both our machines and our proofs, which were quite obvious afterwards, but had not been found before. We decided to put the whole paper-and-pencil proof in the web[2], so readers can make up their own minds.

6.1.2 Definition of the Simulator $\mathsf{Sim}_{\mathcal{H}}$

The Simulator $\mathsf{Sim}_{\mathcal{H}}$ is placed between the trusted host $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ and the adversary A, see Figure 7. Its ports are given by $\{\mathsf{to\_adv}_u?, \mathsf{from\_adv}_u!, \mathsf{from\_adv}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\} \cup \{\mathsf{from\_adv}'_u?, \mathsf{to\_adv}'_u!, \mathsf{to\_adv}'_u{}^{\triangleleft}! \mid u \in \mathcal{H}\}$. The first set contains the ports connected to $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$, the ports of the second set are for communication with the adversary. This means that we have to rename the ports $\mathsf{to\_adv}_u?$, $\mathsf{from\_adv}_u!$, and $\mathsf{from\_adv}_u{}^{\triangleleft}!$ of the adversary into $\mathsf{to\_adv}'_u?$, $\mathsf{from\_adv}'_u!$, and $\mathsf{from\_adv}'_u{}^{\triangleleft}!$, respectively. (Port renaming is permitted in simulatability proofs, since the view is defined independently from the port names.)

States.

Internally, $\mathsf{Sim}_{\mathcal{H}}$ maintains four arrays:

- $(init_{u,v}^{\mathsf{sim}})_{u,v\in\mathcal{M}}$ over $\{0,1\}$,
- $(stopped_u^{\mathsf{sim}})_{u\in\mathcal{H}}$ over $\{0,1\}$,
- $(msg\_out_{u,v}^{\mathsf{sim}})_{u\in\mathcal{A},v\in\mathcal{H}}$ over $\{0,\ldots,\mathsf{max\_in\_user}(k)\}$.

All three arrays are initialized with 0 everywhere.

Inputs and their Evaluation.

We now define the behavior of the simulator. The length functions are again determined by the respective domains. In most cases $\mathsf{Sim}_{\mathcal{H}}$ simply forwards inputs to their corresponding outputs, modifying some internal values.

[2] http://www.zurich.ibm.com/∼mbc/PVS/OrdSecMess.tgz

- *Send initialization:* $(\mathsf{snd\_init})$ at $\mathsf{to\_adv}_u?$:

  Set $init^{\mathsf{sim}}_{u,u} := 1$ and output $(\mathsf{snd\_init})$ at $\mathsf{to\_adv}'_u!$, 1 at ${\mathsf{to\_adv}'_u}^{\triangleleft}!$.

- *Receive initialization:* $(\mathsf{rec\_init}, u)$ at $\mathsf{from\_adv}'_v?$ for $u \in \mathcal{M}$:

  If $stopped^{\mathsf{sim}}_u = 0$ and $init^{\mathsf{sim}}_{u,v} = 0$ and $[u \in \mathcal{H} \implies init^{\mathsf{sim}}_{u,u} = 1]$ set $init^{\mathsf{sim}}_{u,v} := 1$ and output $(\mathsf{rec\_init}, u)$ at $\mathsf{from\_adv}_v!$, 1 at ${\mathsf{from\_adv}_v}^{\triangleleft}!$.

- *Send:* $(\mathsf{send\_blindy}, i, l', v)$ at $\mathsf{to\_adv}_u?$ for $v \in \mathcal{H}$, $l' \leq \mathsf{max\_len}(k)$, $i \leq n \cdot \mathsf{max\_in\_user}(k)$:

  Set $l := l' + c(k)$ and output $(\mathsf{send\_blindy}, i, l, v)$ at $\mathsf{to\_adv}'_u!$, 1 at ${\mathsf{to\_adv}'_u}^{\triangleleft}!$.

- *Send 2:* (send, m, v) at $\mathsf{to\_adv}_u?$ for $v \in \mathcal{M}$, $m \in \Sigma^*$, $\mathsf{len}(m) \leq \mathsf{max\_len}(k) + c(k)$:

  Output (send, m, v) at $\mathsf{to\_adv}'_u!$ and 1 at ${\mathsf{to\_adv}'_u}^{\triangleleft}!$.

- *Receive from honest party u:* $(\mathsf{receive\_blindly}, u, i)$ at $\mathsf{from\_adv}'_v?$ for $u \in \mathcal{H}$:

  If $stopped^{\mathsf{sim}}_v = 0$ then output $(\mathsf{receive\_blindly}, u, i)$ at $\mathsf{from\_adv}_v!$ and 1 at ${\mathsf{from\_adv}_v}^{\triangleleft}!$.

- *Receive from dishonest party u:* $(\mathsf{receive}, u, m')$ at $\mathsf{from\_adv}'_v?$ with $u \in \mathcal{A}$, $\mathsf{len}(m') \leq \mathsf{max\_len}(k) + c(k)$:

  Decompose $m' := (m, num)$: If $stopped^{\mathsf{sim}}_v = 0$, $init^{\mathsf{sim}}_{v,v} = 1$, $init^{\mathsf{sim}}_{u,v} = 1$, $num \geq msg\_out^{\mathsf{sim}}_{u,v}$ ($num = msg\_out^{\mathsf{sim}}_{u,v}$ in the perfect ordered system), set $msg\_out^{\mathsf{sim}}_{u,v} := num + 1$, and output $(\mathsf{receive}, u, m)$ at $\mathsf{from\_adv}_v!$, 1 at ${\mathsf{from\_adv}_v}^{\triangleleft}!$.

- *Stop:* (stop) at $\mathsf{from\_adv}'_u?$:

  If $stopped^{\mathsf{sim}}_u = 0$, set $stopped^{\mathsf{sim}}_u := 1$ and output (stop) at $\mathsf{from\_adv}_u!$, 1 at ${\mathsf{from\_adv}_u}^{\triangleleft}!$.

If a trash input occurs at $\mathsf{to\_adv}_u?$, $\mathsf{Sim}_{\mathcal{H}}$ forwards this input to $\mathsf{to\_adv}'_u!$; trash inputs at $\mathsf{from\_adv}'_u?$ are ignored.

The simulator essentially recalculates the length of message m into $\mathsf{len}((m, num))$ to achieve indistinguishability. Furthermore it decomposes messages sent by the adversary, maybe sorting them out, in order to achieve identical outputs in both systems. Now the overall adversary A' is defined by combining A and $\mathsf{Sim}_{\mathcal{H}}$.

It is easy to see that this combination is polynomial-time in case of a polynomial-time adversary: Each transition of $\mathsf{Sim}_{\mathcal{H}}$ is surely polynomial-time and $\mathsf{Sim}_{\mathcal{H}}$ only accepts inputs of polynomial length at the ports $\mathsf{to\_adv}_u?$. By construction, every such input (either "send initialization", "send", or "trash") will cause the simulator to schedule the adversary subsequently. Since the remaining ports of the simulator are connected to the adversary, there has to at least one step of the adversary after a polynomially bounded number of steps of the simulator. However, since the adversary is polynomial-time, it will enter a final state after a polynomial number of steps, which implies that the steps of the combined machine are also polynomially bounded at the time the adversary halts. Since the definition of combination (cf. [68] ensures that a combined machine enters final state as soon as a contained master scheduler enters final state, we conclude that the combination of a polynomial-time adversary (which is a master scheduler) and the simulator $\mathsf{Sim}_{\mathcal{H}}$ is polynomial-time.

6.1.3 The Ordering Property of the Real Ordered System

We finally address the ordering property of the real ordered system. If the ordering property Req^{OSM} for the specification (Theorem 4.1) and the simulatability property between the specification and the real ordered system (Theorem 6.1) has been proved, it follows easily that the real ordered system also fulfills the property Req^{OSM}, which is captured in the following theorem.

Theorem 6.2 (Ordering Property of the Real System for Ordered Secure Message Transmission) *Let $Sys^{\mathsf{OSM,real}}_{n,L,\mathcal{E},\mathcal{S}}$ be the real system for ordered secure message transmission defined in Section 5.4 for arbitrary parameters $n, L, \mathcal{E}, \mathcal{S}$, and let Req^{OSM} be the integrity property of Definition 4.1. Then $Sys^{\mathsf{OSM,real}}_{n,L,\mathcal{E},\mathcal{S}} \models^{\mathsf{poly}} Req^{\mathsf{OSM}}$ provided that the encryption and signature schemes used are secure.* □

Proof. Theorem 6.1 yields the relation $Sys^{\mathsf{OSM,real}}_{n,L,\mathcal{E},\mathcal{S}} \geq^{\mathsf{poly}}_{\mathsf{sec}} Sys^{\mathsf{OSM,spec}}_{n,L}$, and Theorem 4.1 gives $Sys^{\mathsf{OSM,spec}}_{n,L} \models^{\mathsf{perf}} Req^{\mathsf{OSM}}$, which implies $Sys^{\mathsf{OSM,spec}}_{n,L} \models^{\mathsf{poly}} Req^{\mathsf{OSM}}$. Now Theorem 3.1 implies $Sys^{\mathsf{OSM,real}}_{n,L,\mathcal{E},\mathcal{S}} \models^{\mathsf{poly}} Req^{\mathsf{OSM}}$, since membership in $Req^{\mathsf{OSM}}(S^{\mathsf{OSM}}_{\mathcal{H}})$ is decidable in polynomial time for all $S^{\mathsf{OSM}}_{\mathcal{H}}$, since the send-list and the receive-list are of polynomial length in a polynomial-time configuration. ■

7 Formal Verification of the Bisimulation

In this section, we describe how Theorem 6.1 is formally verified in the theorem proving system PVS [63]. As we already showed in the previous section, it is sufficient to prove that the two collections $\hat{M}_{\mathsf{hybr}}$ and $\hat{M}_{\mathsf{spec}}$ are contained in a deterministic bisimulation.

7.1 Defining the Machines in PVS

In order to do so, we first describe how the machines are formalized in PVS. We subsequently made minor adaptations in the definition of the machines to deal with polynomial runtime more concisely, which do not invalidate the proof.[3]

Since the formal machine descriptions are too large to be given here completely, we use the machine $\mathsf{TH}^{\mathsf{OSM}}_{\mathcal{H}}$ as an example. The complete machine descriptions and the proof are available online.[4]

We denote the number of participating machines by N, and for a given subset $\mathcal{H} \in \{1, \ldots, N\}$, we denote the number of honest users by $M := \#\mathcal{H}$. As defined in Section 4.2, the machine $\mathsf{TH}^{\mathsf{OSM}}_{\mathcal{H}}$ has $2M$ input ports $\{\mathsf{in}_u?, \mathsf{from\_adv}_u? \mid u \in \mathcal{H}\}$. In PVS, we number these input ports $1, \ldots, 2M$, where we identify $1, \ldots, M$ with the user ports and $M+1, \ldots, 2M$ with the adversary ports. Similarly, $\mathsf{TH}^{\mathsf{OSM}}_{\mathcal{H}}$ has output ports $\{\mathsf{out}_u!, \mathsf{to\_adv}_u! \mid u \in \mathcal{H}\}$, which also are numbered $1, \ldots, 2M$. In PVS, we define the following types to denote machines, honest users, and ports:

```
MACH:     TYPE = subrange(1,N)      %% machines
USERS:    TYPE = subrange(1,M)      %% honest users
PORTS:    TYPE = subrange(1,2*M)    %% port numbers
```

The `subrange(i,j)` type is a PVS built-in type denoting the integers $i, \ldots, j$. We further define a type `STRING` to represent messages.

In Section 4.2.3, the different possible inputs to machine $\mathsf{TH}^{\mathsf{OSM}}_{\mathcal{H}}$ are listed, e.g., $(\mathsf{snd\_init})$, $(\mathsf{rec\_init}, u), \ldots$ In PVS, the type of input ports is defined using a PVS abstract datatype [62]. The prefix `m1i` in the following stands for "inputs of machine 1", which is $\mathsf{TH}^{\mathsf{OSM}}_{\mathcal{H}}$, and is used to distinguish between inputs and outputs of the different machines.

```
m1_in_port: DATATYPE
BEGIN
  m1i_snd_init:                              m1i_snd_init?
```

[3] Unfortunately, we are currently not able to incorporate these changes in the PVS proof, since PVS is not freely accessible for commercial use, which prohibits us from using it as we are currently affiliated with IBM. The existing proof was developed when the authors were affiliated with Saarland University.

[4] http://www.zurich.ibm.com/∼mbc/PVS/OrdSecMess.tgz

```
    m1i_rec_init(u: MACH):                      m1i_rec_init?
    m1i_send(m: STRING, v: MACH):               m1i_send?
    m1i_receive_blindly(u: USERS, i: posnat): m1i_receive_blindly?
    m1i_receive(u: MACH, m: STRING):            m1i_receive?
    m1i_stop:                                   m1i_stop?
  END m1_in_port
```

This defines an abstract datatype with *constructors* `m1i_snd_init`, `m1i_rec_init` etc. For example, for given u, i, `m1i_receive_blindly(u,i)` constructs an instance of the above datatype, which we identify with $(\text{receive\_blindly}, u, i)$. Given an instance p of this datatype, we can use the *recognizers* on the right side of the definition to distinguish between the different forms. For example, `m1i_receive_blindly?(p)` checks whether the instance p of the `m1i_in_port` datatype was constructed from the `m1i_receive_blindly` constructor. If it was, the components u and i can be restored using the *accessor functions* $u(\cdot)$ and $i(\cdot)$; for example, $u(p)$ returns the u component of p. The accessor functions may be overloaded for different constructors (e.g., u is overloaded in `m1i_rec_init`, `m1i_receive_blindly` and `m1i_receive`).

The machine $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ performs a step iff exactly one of the input ports is active. In this case, we call the input *ok*, otherwise *garbage*. The type of the complete inputs to $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ comprising all $2M$ input ports is therefore either garbage, or the number u of the active port together with the input p on port u. This is formalized in the following PVS datatype:

```
  M1_INP: DATATYPE
  BEGIN
    m1i_garbage:                                  m1i_garbage?
    m1i_ok(u: PORTS, p: m1_in_port):              m1i_ok?
  END M1_INP
```

Similar datatypes `m1_out_port` and `M1_OUT` are defined to denote the type of individual outputs, and the type of the complete output of $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$, respectively.

Next we define the state type of $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$. As defined in Section 4.2.2, this state consists of seven one- or two-dimensional arrays. In PVS, arrays are modeled as functions mapping the indices to the contents of the array. For example `[MACH,USERS -> nat]` defines a two-dimensional array of natural numbers, where the first index ranges over $\mathcal{M}$, and the second ranges over $\mathcal{H}$. The state type of $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ is defined as a record of such arrays. There is only one small exception: the array $deliver_{u,v}^{\mathsf{OSM}}$ stores lists of tuples (m, i) (e.g., see the "Send" transition), where m is a string and $i \in \mathbb{N}$. It is convenient in PVS to decompose this array of lists of tuples into two arrays of lists, where the first array $deliver_{u,v}^{\mathsf{OSM}}$ stores lists of messages m, and the second array $deliv\_i_{u,v}^{\mathsf{OSM}}$ stores lists of naturals i. Lists are defined as a recursive algebraic abstract datatype in the PVS library [62]. Altogether, this yields a state type of eight arrays:

```
M1_STATE: TYPE = [# init_spec: [MACH,MACH -> bool],
                    sc_in_spec: [USERS -> nat],
                    msg_in_spec: [USERS,MACH -> nat],
                    msg_out_spec: [USERS,USERS -> posnat],
                    sc_out_spec: [USERS -> nat],
                    deliver_spec: [USERS,USERS -> list[STRING]],
                    deliv_i_spec: [USERS,USERS -> list[posnat]],
                    stopped_spec: [USERS -> bool] #]
```

The initial state `m1_init` is defined as a constant of type `M1_STATE`:

```
  M1_init: M1_STATE = (#
    init_spec := LAMBDA (w1,w2: MACH): FALSE,
    ...
```

```
    deliv_i_spec := LAMBDA (u1,u2: USERS): null,
    stopped_spec := LAMBDA (u1: USERS): FALSE #)
```

The constructor `null` denotes the empty list. The machine $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ is now formalized in PVS as a next-state/output function mapping current state and inputs to the next state and outputs. We exemplarily give the first few lines of the PVS code:

```
M1_ns(S: M1_STATE, I: M1_INP): [# ns: M1_STATE, O: M1_OUT #] =
  IF m1i_garbage?(I) THEN
     (# ns:=S, O:=m1o_garbage #)
        %% do not change the state, output nothing
  ELSE
    LET ua1=ua(I), p=p(I) IN
      %% ua1 is the active port number,
      %% p is the input on this port
    IF ua1<=M AND m1i_snd_init?(p) THEN
      %% we have a send-init on a user port (<=M);
      IF S‘sc_in_spec(ua1)<s1k  AND NOT S‘stopped_spec(ua1) THEN
        IF S‘init_spec(ua1,ua1) THEN
  (# ns:=S WITH [ ‘sc_in_spec(ua1) := sc_in_spec(ua1)+1,
             O:=m1o_garbage #)
          %% increment sc_in_spec, but do not send any output
ELSE
  (# ns:=S WITH [ ‘sc_in_spec(ua1) := sc_in_spec(ua1)+1,
                          ‘init_spec(ua1,ua1) := TRUE ],
             O := m1o_ok(M+ua1, m1o_snd_init) #)
          %% increment sc_in_spec, set init_spec(ua1,ua1):=true
          %% send m1o_snd_init to adversary port M+ua1
        ENDIF
      ELSE %% otherwise do nothing
(# ns:=S, Out:=m1o_garbage #)
      ENDIF
    ELSIF ua1>M AND m1i_rec_init?(p) THEN
    ...
```

In a similar way we have formalized the machines $\mathsf{TH}_{\mathcal{H}}^{\mathsf{SM}}$, $\{\mathsf{M}_u^{\mathsf{filt}} \mid u \in \mathcal{H}\}$, and $\mathsf{Sim}_{\mathcal{H}}$. The M machines $\mathsf{M}_u^{\mathsf{filt}}$ in the left part of Figure 7 have been combined into a single machine in PVS; however, this is only syntactic and does not change the semantics. The combination of the machines $\mathsf{TH}_{\mathcal{H}}^{\mathsf{SM}}$ and $\{\mathsf{M}_u^{\mathsf{filt}} \mid u \in \mathcal{H}\}$ respectively $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ and $\mathsf{Sim}_{\mathcal{H}}$ is straightforward by composition of the corresponding state transition functions: An input from H is always first handled by a machine $\mathsf{M}_u^{\mathsf{filt}}$ and $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$, and then by $\mathsf{TH}_{\mathcal{H}}^{\mathsf{SM}}$ and $\mathsf{Sim}_{\mathcal{H}}$, respectively, and vice versa. This saves us from implementing the full asynchronous scheduling algorithm in PVS for this example.

The only non-trivial choice we have made in the transliteration of the machines to PVS is the type of the input- and output-ports. In a previous attempt, we did not use the abstract datatype definition of `M1_INP`, but defined `M1_INP` as an array of $2M$ individual input ports; in order to model non-active ports, we added an `m1i_inactive` form to the input port type `m1i_in_port`. An input from `M1_INP` was defined to be *ok* iff exactly one of the ports is different from `m1i_inactive`. This obviously models the same valid inputs as the definition of `M1_INP` above. The problem with the array definition is that extracting the active port number u involves an application of the choice-function ε in order to choose the index u of the array for which the port is active. The application of the choice-function considerably complicates the proofs in PVS, since the definition of ε is not constructive in PVS. In contrast, in the definition using the abstract datatype, the active port number u can be constructively extracted from the input by applying the accessor function of the abstract datatype. Due to constructiveness, the proofs in

PVS become much simpler. This problem in the port definition also applies to the output ports of the machines.

The rest of the transliteration of the machine definitions to PVS is straightforward. In the following, we revert to standard mathematical notation for the sake of brevity and readability.

7.2 Proving the Bisimulation

In order to prove Lemma 6.1, we consider the following predicates on the states of the collections $\hat{M}_{\mathsf{hybr}}$ and $\hat{M}_{\mathsf{spec}}$ and show them to be invariant.

- *Stop Flags:* This invariants consists of two subparts:
  - $\forall u \in \mathcal{H}\colon stopped_u^{\mathsf{OSM}} = stopped_u^{\mathsf{filt}} \wedge stopped_u^{\mathsf{SM}} = stopped_u^{\mathsf{sim}}$,
  - $\forall u \in \mathcal{H}\colon stopped_u^{\mathsf{SM}} = 0 \Rightarrow stopped_u^{\mathsf{OSM}} = 0$,
- *Inputs Counters:* $\forall u \in \mathcal{H}\colon sc\_in_u^{\mathsf{filt}} = sc\_in_u^{\mathsf{OSM}}$,
- *Output Counters:* $\forall u \in \mathcal{H}\colon sc\_out_u^{\mathsf{filt}} = sc\_out_u^{\mathsf{OSM}}$,
- *Initialization Arrays:* $\forall u \in \mathcal{H}, w \in \mathcal{M}\colon\ init_{w,u}^{\mathsf{SM}} = init_{w,u}^{\mathsf{sim}} = init_{w,u}^{\mathsf{OSM}}$,
- *User Messages:* $\forall u \in \mathcal{H}, w \in \mathcal{M}\colon msg\_in_{u,w}^{\mathsf{filt}} = msg\_in_{u,w}^{\mathsf{OSM}}$,
- *Network Messages:* This invariant consists of two subparts:
  - $\forall u, w \in \mathcal{H}\colon msg\_out_{w,u}^{\mathsf{filt}} = msg\_out_{w,u}^{\mathsf{OSM}}$,
  - $\forall u \in \mathcal{H}$ with $sc\_out_u^{\mathsf{OSM}} < \mathsf{max\_in\_adv}(k)$, $w \in \mathcal{M} \setminus \mathcal{H}\colon msg\_out_{w,u}^{\mathsf{filt}} = msg\_out_{w,u}^{\mathsf{sim}}$,
- *Message Array Content:* $\forall u, v \in \mathcal{H}\colon deliver_{u,v}^{\mathsf{SM}} = deliver_{u,v}^{\mathsf{OSM}} \wedge deliv\_i_{u,v}^{\mathsf{SM}} = deliv\_i_{u,v}^{\mathsf{OSM}}$,
- *Message Array Length:* $\forall u, v \in \mathcal{H}\colon\ length(deliver_{u,v}^{\mathsf{SM}}) = length(deliv\_i_{u,v}^{\mathsf{SM}})$, where $length$ is the PVS function delivering the length of lists,
- *Message Array Length 2:* $\forall u, v \in \mathcal{H}\colon\ length(deliver_{u,v}^{\mathsf{OSM}}) = length(deliv\_i_{u,v}^{\mathsf{OSM}})$.

Each of the 9 invariants is formalized as a predicate $\phi_i(S_{\mathsf{hybr}}, S_{\mathsf{spec}})$ on the current states of the two collections $\hat{M}_{\mathsf{hybr}}$ and $\hat{M}_{\mathsf{spec}}$. The conjunction of all the ϕ_i yields the bisimulation relation ϕ. Let δ_{hybr} and δ_{spec} denote the overall transition function of the machine collections $\hat{M}_{\mathsf{hybr}}$ and $\hat{M}_{\mathsf{spec}}$, respectively. The following theorem asserts that the invariants indeed are invariants of these collections:

Theorem 7.1 *Let S_{hybr} and S_{spec} be states of the two collections $\hat{M}_{\mathsf{hybr}}$ and $\hat{M}_{\mathsf{spec}}$ such that all invariants $\phi_i(S_{\mathsf{hybr}}, S_{\mathsf{spec}})$, $1 \leq i \leq 9$ hold. The transition functions $\delta_{\mathsf{hybr}}, \delta_{\mathsf{spec}}$ preserve the invariants, i.e., for an arbitrary overall input $\mathcal{I}$ of $\hat{M}_{\mathsf{hybr}}$ and $\hat{M}_{\mathsf{spec}}$ we have*

$$\phi_i(S'_{\mathsf{hybr}}, S'_{\mathsf{spec}})\ \forall i, 1 \leq i \leq 9$$

with $(S'_{\mathsf{hybr}}, \mathcal{O}_{\mathsf{hybr}}) := \delta_{\mathsf{hybr}}(S_{\mathsf{hybr}}, \mathcal{I})$ and $(S'_{\mathsf{spec}}, \mathcal{O}_{\mathsf{spec}}) := \delta_{\mathsf{spec}}(S_{\mathsf{spec}}, \mathcal{I})$. Furthermore, the initial states $initial_{\mathsf{hybr}}$ and $initial_{\mathsf{spec}}$ satisfy all 9 invariants. □

In PVS, this theorem is split into 9 lemmas, one for each invariant. Using the invariants ϕ_i, we prove the following theorem:

Theorem 7.2 *Let S_{hybr} and S_{spec} be states satisfying all invariants $\phi_i(S_{\mathsf{hybr}}, S_{\mathsf{spec}})$, $1 \leq i \leq 9$, and let $\mathcal{I}$ be an overall input of the collections $\hat{M}_{\mathsf{hybr}}$ and $\hat{M}_{\mathsf{spec}}$. Then both collections make the same outputs on all ports to the users and the adversary.* □

Together, Theorems 7.1 and 7.2 prove that the two systems are bisimilar, which finishes the proof of Lemma 6.1, and hence also the proof of Theorem 6.1.

7.3 Verification Effort

The manual proof effort in PVS is rather small. The proofs make heavy use of the built-in PVS strategy `(grind)`, which expands definitions and performs automatic case-splitting. The main effort was to figure out the correct parameters for the `(grind)` command. The proof goals not resolved by `(grind)` were proved with little manual assistance. However, looking for errors and thinking about the necessary modifications of the machines was a time-consuming task. During our proof attempts, we simultaneously debugged the machines until we finally found the correct specifications of all machines. After that, the proof itself turned out to be quite easy. Altogether, the formalization of the machines in PVS took 2 weeks, and the development of the proofs took another week (given prior familiarity with PVS). A complete checking of the proof takes about one hour on a 600 MHz Athlon processor.

8 Verification of the Ordered Channel Specification

In this section, we formally verify Theorem 4.1, i.e., that message reordering in our specification of Section 4 is in fact prevented. the property seems to hold by construction, but experience shows that such proofs made by 'simply looking' are often flawed. Even if proofs of this kind are made by hand in a rigorous way, they often turn out to be apparently straightforward and dull which yields proofs with faults and imperfections. Following our approach of the previous section, we formally verify the integrity property in PVS. This will be described in the following. For reasons of readability and brevity, we again use standard mathematical notation instead of PVS syntax. The PVS sources are available online.[4]

According to Definition 2.1, we assume that the machine $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ operates on an input set $\mathcal{I}_{\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}}$ (short $\mathcal{I}$), a state set $States_{\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}}$ (short S), and an output set $\mathcal{O}_{\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}}$ (short $\mathcal{O}$). For convenience, the (deterministic) transition function $\delta_{\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}} : \mathcal{I} \times S \to S \times \mathcal{O}$ is split into $\delta : \mathcal{I} \times S \to S$ and $\omega : \mathcal{I} \times S \to \mathcal{O}$, which denote the next-state and output part of $\delta_{\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}}$, respectively.

In order to formulate the property, we need a PVS-suited, formal notation of (infinite) runs of a machine, of lists, of what it means that a list l_1 is a sublist of a list l_2, and we need formalizations of the *receive-list* and *send-list*.

Definition 8.1 (Input sequence, state trace, output sequence) *Let M be a machine with input set $\mathcal{I}_{\mathsf{M}}$, state set $States_{\mathsf{M}}$, output set $\mathcal{O}_{\mathsf{M}}$, state transition function δ, and output transition function ω. Call $s_{init} \in States_{\mathsf{M}}$ the initial state. An* input sequence *$i : \mathbb{N} \to \mathcal{I}_{\mathsf{M}}$ for machine M is a function mapping the time (modeled as the set $\mathbb{N}$) to inputs $i(t) \in \mathcal{I}_{\mathsf{M}}$. A given input sequence i defines a sequence of states $s^i : \mathbb{N} \to States_{\mathsf{M}}$ of the machine M by the following recursive construction:*

$$\begin{aligned} s^i(0) &:= s_{init}, \\ s^i(t+1) &:= \delta(i(t), s^i(t)). \end{aligned}$$

The sequence s^i is called state-trace of M under i. *The* output sequence *$o^i : \mathbb{N} \to O$ of the run is defined as*

$$o^i(t) := \omega(i(t), s^i(t)).$$

We omit the index i if the input sequence is clear from the context. For components x of the state type, we write $x(t)$ for the content of x in $s(t)$. For example, we write $deliver_{u,v}^{\mathsf{OSM}}(t)$ to denote the content at time t of the list $deliver_{u,v}^{\mathsf{OSM}}$, which is part of the state of $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$. $\diamond$

In the context of $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$, the input sequence i consists of the messages that the honest users and the adversary send to $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$.

As our upcoming definitions uses the PVS-intern terminology of lists, we restate the definition from [62], and further give the definition of sublists.

Definition 8.2 (Lists) *A* list over type T *is the closure of applications of the constructor $null$ yielding an empty list, and the constructor $cons(car\colon T, cdr\colon list[T])$ yielding a list with head car and tail cdr. It holds $car(cons(t,l)) = t$ and $cdr(cons(t,l)) = l$. The predicates $null?(l)$ and $cons?(l)$ are used to test whether l is empty or non-empty, respectively. PVS provides functions $length(l)$, $append(t,l)$, and $nth(l,i)$ to measure the length of a list l, to append an element t at the end of the list l, and to access the i^{th} element of l (counted from 0).* $\diamond$

Definition 8.3 (Sublists) *A list l_1 is called sublist of a list l_2 (written $l_1 \subseteq l_2$) iff the following recursive predicate is satisfied:*

$$\begin{aligned} l_1 \subseteq l_2 : \Longleftrightarrow \quad & null?(l_1) \vee \\ & cons?(l_1) \wedge \quad \big(car(l_1) = car(l_2) \wedge cdr(l_1) \subseteq cdr(l_2) \\ & \qquad\qquad \vee\, l_1 \subseteq cdr(l_2)\big). \end{aligned}$$

Let $k \in \mathbb{N}_0$. The list l_1 is called sublist of the k-prefix of l_2 (written $l_1 \subseteq^k l_2$) iff the following recursive predicate is satisfied:

$$\begin{aligned} l_1 \subseteq^k l_2 : \Longleftrightarrow \quad & null?(l_1) \vee \\ & cons?(l_1) \wedge k \geq 1 \wedge \quad \big(car(l_1) = car(l_2) \wedge cdr(l_1) \subseteq^{k-1} cdr(l_2) \\ & \qquad\qquad \vee\, l_1 \subseteq^{k-1} cdr(l_2)\big). \end{aligned}$$

$\diamond$

The following lemma summarizes some facts on lists and sublists:

Lemma 8.1 *Let l_1, l_2, l_3 be lists over some type T, let $t \in T$, and $k, k' \in \mathbb{N}_0$. It holds:*

1. $k \leq length(l_1) \implies nth(append(t, l_1), k) = \begin{cases} nth(l_1, k) & \text{if } k < length(l_1) \\ t & \text{otherwise} \end{cases}$
2. $l_1 \subseteq l_2 \implies l_1 \subseteq append(t, l_2)$
3. $l_1 \subseteq l_2 \implies append(t, l_1) \subseteq append(t, l_2)$
4. $l_1 \subseteq^k l_2 \implies l_1 \subseteq^k append(t, l_2)$
5. $k < length(l_2) \wedge l_1 \subseteq^k l_2 \implies append(nth(l_2, k), l_1) \subseteq^{k+1} l_2$,
   that is, one may append the k^{th} element (counted from 0) of l_2 to l_1 while preserving the prefix-sublist property.
6. $k' \geq k \wedge l_1 \subseteq^k l_2 \implies l_1 \subseteq^{k'} l_2$
7. $l_1 \subseteq^k l_2 \implies l_1 \subseteq l_2$

8. $l_1 \subseteq l_2 \wedge l_2 \subseteq l_3 \implies l_1 \subseteq l_3$

□

All claims are proved by induction on the recursive structure of the lists.

Definition 8.4 (Receive- and send-list) *Let i be an input sequence for machine $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$, and let s and o be the corresponding state-trace of $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ and the output sequence, respectively. Let $u, v \in \mathcal{H}$. The receive-list is obtained by appending a new element m whenever v receives a message $(\mathsf{receive}, m, u)$ from $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$. The send-list is obtained by appending m whenever u sends a message (send, m, v) to $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$. Formally, this is captured in the following recursive definitions:*

$$\textit{recvlist}_{u,v}^{i}(t) := \begin{cases} null & \textit{if } t = -1, \\ append(m, \textit{recvlist}_{u,v}^{i}(t-1)) & \textit{if } t \geq 0 \wedge o^i(t) = (\mathsf{receive}, m, u) \\ & \quad \textit{at } \mathsf{out}_v!. \\ \textit{recvlist}_{u,v}^{i}(t-1) & \textit{otherwise} \end{cases}$$

$$\textit{sendlist}_{u,v}^{i}(t) := \begin{cases} null & \textit{if } t = -1, \\ append(m, \textit{sendlist}_{u,v}^{i}(t-1)) & \textit{if } t \geq 0 \wedge i(t) = (\mathsf{send}, m, v) \\ & \quad \textit{at } \mathsf{in}_u?. \\ \textit{sendlist}_{u,v}^{i}(t-1) & \textit{otherwise} \end{cases}$$

◇

We now are ready to give a precise, PVS-suited formulation of Theorem 4.1, i.e., the integrity property we are aiming to prove:

Theorem 8.1 *For any $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ input sequence i, for any $u, v \in \mathcal{H}$, $u \neq v$, and any point in time $t \in \mathbb{N}$, it holds*

$$\textit{recvlist}_{u,v}^{i}(t) \subseteq \textit{sendlist}_{u,v}^{i}(t). \tag{1}$$

In the following, we omit the index i. □

Proof (sketch). The proof is split into two parts: we prove $\textit{recvlist}_{u,v}(t-1) \subseteq deliver_{u,v}^{\mathsf{OSM}}(t)$ and $deliver_{u,v}^{\mathsf{OSM}}(t) \subseteq \textit{sendlist}_{u,v}(t-1)$. The claim of the theorem then follows from Lemma 8.1.8.

The second claim $deliver_{u,v}^{\mathsf{OSM}}(t) \subseteq \textit{sendlist}_{u,v}(t-1)$ is proved by induction on t. Both induction base and step are proved in PVS by the built-in strategy `(grind)`, which performs automatic definition expanding and rewriting with Lemma 8.1.

The first claim $\textit{recvlist}_{u,v}(t-1) \subseteq deliver_{u,v}^{\mathsf{OSM}}(t)$ is more complicated. The claim is also proved by induction on t. However, it is easy to see that the claim is not inductive: in case of a $(\mathsf{receive\_blindly}, u, i)$ at $\mathsf{from\_adv}_v?$, $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ outputs $(\mathsf{receive}, m, u)$ to $\mathsf{out}_v!$, where $(m, j) := deliver_{u,v}^{\mathsf{OSM}}[i]$, i.e., m is the ith message of the $deliver_{u,v}^{\mathsf{OSM}}$ list. By the definition of the receive-list, the message m is appended to $\textit{recvlist}_{u,v}$. In order to prove that $\textit{recvlist}_{u,v} \subseteq deliver_{u,v}^{\mathsf{OSM}}$ is preserved during this transition, it is necessary to know that the receive list was a sublist of the prefix of the $deliver_{u,v}^{\mathsf{OSM}}$ list that does not reach to m. It would suffice to know that

$$\textit{recvlist}_{u,v}(t-1) \subseteq^{i} deliver_{u,v}^{\mathsf{OSM}}(t).$$

Then the claim follows from Lemma 8.1.5.

We therefore strengthen the invariant to comprise the prefix-sublist property. However, the i in the above prefix-sublist relation stems from the input $(\mathsf{receive\_blindly}, u, i)$, and hence is not suited to state the invariant. To circumvent this problem, we recursively construct a sequence $\mathit{last\_rcv\_blindly}_{u,v}(t)$ which holds the parameter i of the last valid $(\mathsf{receive\_blindly}, u, i)$ received by $\mathsf{TH}_{\mathcal{H}}^{\mathsf{OSM}}$ at $\mathsf{from\_adv}_v$?; then

$$\mathit{recvlist}_{u,v}(t-1) \subseteq^{l} \mathit{deliver}_{u,v}^{\mathsf{OSM}}(t) \text{ with } l = \mathit{last\_rcv\_blindly}_{u,v}(t)$$

is an invariant of the system. We further strengthen this invariant by asserting that $\mathit{last\_rcv\_blindly}_{u,v}(t)$ and the j's stored in the $\mathit{deliver}_{u,v}^{\mathsf{OSM}}$ list grow monotonically. Together this yields the inductive invariant. We omit the details and again refer the to the PVS files available online. ∎

8.1 Verification Effort

Together, the development of the inductive invariant and its proof took 2 weeks, which included some failed approaches in strengthening the invariant to become inductive. The proof of the invariant takes 500 proof commands. A further week and 350 proof commands were needed for the development of the sublist theory, which can be reused in future verification projects. The main difficulty during the verification of the invariant was finding the stronger inductive invariant. Once the correct invariant was found, its proof was quite easy. Before we started the formal verification, we had a hand-written proof of Theorem 8.1. However, the proof was incomplete in the sense that we did not prove some needed invariants; in fact, we did not even notice that we used these invariants in our hand-made proofs, because of our intuitive understanding of the system.

9 Conclusion and Outlook

In this paper, we have addressed the problem how cryptographic protocols in asynchronous networks can be verified both machine-aided and sound with respect to the definitions of cryptography. We have established a preservation theorem for integrity properties stating that the verification of integrity properties of abstract specifications automatically carries over to the concrete implementations if the implementation is secure in the sense of simulatability. Moreover, we have shown that logic derivations among integrity properties are valid for the concrete systems in the cryptographic sense, which makes them accessible to theorem provers. As an example, we have presented a specification of secure message transmission with ordered channels, which we formally validated using the theorem proving system PVS. Furthermore, we used formally verified bisimulations to derive a secure implementation. Together with the preservation theorem these results imply that the correctness of the verified property is equivalent to the security of the underlying cryptographic primitives, i.e., if the primitives for encryption and digital signatures are secure with respect to their respective security definitions, the integrity property holds for the concrete implementation. This yields the first formal verification of a cryptographic protocol that is sound with respect to the cryptographic definitions. We hope that our work paves the way for the actual use of automatic proof tools for many similar cryptographically faithful proofs of security protocols.

Acknowledgments

We thank *Michael Steiner* and *Michael Waidner* for interesting discussions.

References

[1] M. Abadi and J. Jürjens. Formal eavesdropping and its computational interpretation. In *Proc. 4th International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 82–94, 2001.

[2] M. Abadi and P. Rogaway. Reconciling two views of cryptography: The computational soundness of formal encryption. In *Proc. 1st IFIP International Conference on Theoretical Computer Science*, volume 1872 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2000.

[3] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

[4] M. Backes. A cryptographically sound dolev-yao style security proof of the Otway-Rees protocol. In *Proceedings of 9th European Symposium on Research in Computer Security (ESORICS)*, volume 3193 of *Lecture Notes in Computer Science*, pages 89–108. Springer, 2004.

[5] M. Backes. Quantifying probabilistic information flow in computational reactive systems. In *Proceedings of 10th European Symposium on Research in Computer Security (ESORICS)*, volume 3679 of *Lecture Notes in Computer Science*, pages 336–354. Springer, 2005.

[6] M. Backes. Unifying simulatability definitions in cryptographic systems under different timing assumptions. *Journal of Logic and Algebraic Programming (JLAP)*, 2:157–188, 2005.

[7] M. Backes and M. Duermuth. A cryptographically sound Dolev-Yao style security proof of an electronic payment system. In *Proceedings of 18th IEEE Computer Security Foundations Workshop (CSFW)*, pages 78–93, 2005.

[8] M. Backes, M. Duermuth, D. Hofheinz, and R. Kuesters. Conditional reactive simulatability. In *Proceedings of 11th European Symposium on Research in Computer Security (ESORICS)*, volume 4189 of *Lecture Notes in Computer Science*, pages 424–443. Springer, 2006. Preprint on IACR ePrint 2006/132.

[9] M. Backes and D. Hofheinz. How to break and repair a universally composable signature functionality. In *Proceedings of 7th Information Security Conference (ISC)*, volume 3225 of *Lecture Notes in Computer Science*, pages 61–72. Springer, 2004. Preprint on IACR ePrint 2003/240.

[10] M. Backes and C. Jacobi. Cryptographically sound and machine-assisted verification of security protocols. In *Proc. 20th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 2607 of *Lecture Notes in Computer Science*, pages 675–686. Springer, 2003.

[11] M. Backes, C. Jacobi, and B. Pfitzmann. Deriving cryptographically sound implementations using composition and formally verified bisimulation. In *Proc. 11th Symposium on Formal Methods Europe (FME 2002)*, volume 2391 of *Lecture Notes in Computer Science*, pages 310–329. Springer, 2002.

[12] M. Backes and P. Laud. Computationally sound secrecy proofs by mechanized flow analysis. In *Proceedings of 13th ACM Conference on Computer and Communications Security (CCS)*, pages 370–379, 2006.

[13] M. Backes, S. Moedersheim, B. Pfitzmann, and L. Vigano. Symbolic and cryptographic analysis of the secure WS-ReliableMessaging Scenario. In *Proceedings of Foundations of Software Science*

and Computational Structures (FOSSACS), volume 3921 of *Lecture Notes in Computer Science*, pages 428–445. Springer, 2006.

[14] M. Backes and B. Pfitzmann. Computational probabilistic non-interference. *International Journal of Information Security (IJIS)*, 3(1):42–60, 2004.

[15] M. Backes and B. Pfitzmann. A cryptographically sound security proof of the Needham-Schroeder-Lowe public-key protocol. *IEEE Journal on Selected Areas of Computing (JSAC)*, 22(10):2075–2086, 2004.

[16] M. Backes and B. Pfitzmann. Symmetric encryption in a simulatable Dolev-Yao style cryptographic library. In *Proceedings of 17th IEEE Computer Security Foundations Workshop (CSFW)*, pages 204–218, 2004.

[17] M. Backes and B. Pfitzmann. Limits of the cryptographic realization of Dolev-Yao-style XOR. In *Proceedings of 10th European Symposium on Research in Computer Security (ESORICS)*, volume 3679 of *Lecture Notes in Computer Science*, pages 178–196. Springer, 2005.

[18] M. Backes and B. Pfitzmann. Relating cryptographic und symbolic secrecy. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2(2):109–123, 2005.

[19] M. Backes, B. Pfitzmann, M. Steiner, and M. Waidner. Polynomial liveness. *Journal of Computer Security*, 12(3-4):589–617, 2004.

[20] M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations (extended abstract). In *Proc. 10th ACM Conference on Computer and Communications Security*, pages 220–230, 2003.

[21] M. Backes, B. Pfitzmann, and M. Waidner. A universally composable cryptographic library. *IACR Cryptology ePrint Archive*, 2003:15, 2003.

[22] M. Backes, B. Pfitzmann, and M. Waidner. A general composition theorem for secure reactive system. In *Proceedings of 1st Theory of Cryptography Conference (TCC)*, volume 2951 of *Lecture Notes in Computer Science*, pages 336–354. Springer, 2004.

[23] M. Backes, B. Pfitzmann, and M. Waidner. Low-level ideal signatures and general integrity idealization. In *Proceedings of 7th Information Security Conference (ISC)*, volume 3225 of *Lecture Notes in Computer Science*, pages 39–51. Springer, 2004.

[24] M. Backes, B. Pfitzmann, and M. Waidner. Secure asynchronous reactive systems. *IACR Cryptology ePrint Archive*, 2004:82, 2004.

[25] M. Backes, B. Pfitzmann, and M. Waidner. Reactively secure signature schemes. *International Journal of Information Security (IJIS)*, 4(4):242–252, 2005.

[26] M. Backes, B. Pfitzmann, and M. Waidner. Symmetric authentication within a simulatable cryptographic library. *International Journal of Information Security (IJIS)*, 4(3):135–154, 2005.

[27] D. Beaver. Secure multiparty protocols and zero knowledge proof systems tolerating a faulty minority. *Journal of Cryptology*, 4(2):75–122, 1991.

[28] G. Bella, F. Massacci, and L. C. Paulson. The verification of an industrial payment protocol: The set purchase phase. In *Proc. 9th ACM Conference on Computer and Communications Security*, pages 12–20, 2002.

[29] M. Bellare, T. Kohno, and C. Namprempre. Authenticated encryption in ssh: Provably fixing the ssh binary packet protocol. In *Proc. 9th ACM Conference on Computer and Communications Security*, pages 1–11, 2002.

[30] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Advances in Cryptology: CRYPTO '93*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249. Springer, 1994.

[31] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS. In *Advances in Cryptology: CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1998.

[32] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 3(1):143–202, 2000.

[33] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proc. 42nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 136–145, 2001. Extended version in Cryptology ePrint Archive, Report 2000/67, `http://eprint.iacr.org/`.

[34] Z. Dang and R. Kemmerer. Using the ASTRAL model checker for cryptographic protocol analysis. In *Proc. DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997. `http://dimacs.rutgers.edu/Workshops/Security/`.

[35] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. Secure protocol composition (extended abstract). In *Proc. 1st ACM Workshop on Formal Methods in Security Engineering (FMSE)*, pages 11–23, 2003.

[36] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, 1981.

[37] Y. Desmedt and K. Kurosawa. How to break a practical mix and design a new one. In *Advances in Cryptology: EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 557–572. Springer, 2000.

[38] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[39] B. Dutertre and S. Schneider. Using a PVS embedding of CSP to verify authentication protocols. In *Proc. International Conference on Theorem Proving in Higher Order Logics (TPHOL)*, volume 1275 of *Lecture Notes in Computer Science*, pages 121–136. Springer, 1997.

[40] D. Fisher. Millions of .Net Passport accounts put at risk. *eWeek*, May 2003. (Flaw detected by Muhammad Faisal Rauf Danka).

[41] S. Goldwasser and L. Levin. Fair computation of general functions in presence of immoral majority. In *Advances in Cryptology: CRYPTO '90*, volume 537 of *Lecture Notes in Computer Science*, pages 77–93. Springer, 1990.

[42] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28:270–299, 1984.

[43] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[44] J. D. Guttman, F. J. Thayer Fabrega, and L. Zuck. The faithfulness of abstract protocol analysis: Message authentication. In *Proc. 8th ACM Conference on Computer and Communications Security*, pages 186–195, 2001.

[45] M. Hirt and U. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology*, 13(1):31–60, 2000.

[46] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science, Prentice Hall, Hemel Hempstead, 1985.

[47] D. M. Johnson and F. Javier Thayer. Security and the composition of machines. In *Proc. 1st IEEE Computer Security Foundations Workshop (CSFW)*, pages 72–89, 1988.

[48] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, 1994.

[49] P. Laud. Semantics and program analysis of computationally secure information flow. In *Proc. 10th European Symposium on Programming (ESOP)*, pages 77–91, 2001.

[50] P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *Proc. 5th ACM Conference on Computer and Communications Security*, pages 112–121, 1998.

[51] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. 2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.

[52] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Francisco, 1996.

[53] H. Mantel. On the composition of secure systems. In *Proc. 23rd IEEE Symposium on Security & Privacy*, pages 88–101, 2002.

[54] D. McCullough. Specifications for multi-level security and a hook-up property. In *Proc. 8th IEEE Symposium on Security & Privacy*, pages 161–166, 1987.

[55] D. McCullough. A hookup theorem for multilevel security. *IEEE Transactions on Software Engineering*, 16(6):563–568, 1990.

[56] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. 15th IEEE Symposium on Security & Privacy*, pages 79–93, 1994.

[57] J. McLean. A general theory of composition for a class of "possibilistic" security properties. *IEEE Transactions on Software Engineering*, 22(1):53–67, 1996.

[58] S. Micali and P. Rogaway. Secure computation. In *Advances in Cryptology: CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 392–404. Springer, 1991.

[59] D. Micciancio and B. Warinschi. Soundness of formal encryption in the presence of active adversaries. In *Proc. 1st Theory of Cryptography Conference (TCC)*, volume 2951 of *Lecture Notes in Computer Science*, pages 133–151. Springer, 2004.

[60] J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murϕ. In *Proc. 18th IEEE Symposium on Security & Privacy*, pages 141–151, 1997.

[61] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 12(21):993–999, 1978.

[62] S. Owre and N. Shankar. Abstract datatypes in PVS. Technical report, Computer Science Laboratory, SRI International, 1993.

[63] S. Owre, N. Shankar, and J. M. Rushby. PVS: A prototype verification system. In *Proc. 11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[64] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Cryptology*, 6(1):85–128, 1998.

[65] B. Pfitzmann, M. Schunter, and M. Waidner. Cryptographic security of reactive systems. Presented at the *DERA/RHUL Workshop on Secure Architectures and Information Flow*, 1999, Electronic Notes in Theoretical Computer Science (ENTCS), March 2000. `http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/menu.htm`.

[66] B. Pfitzmann and M. Waidner. How to break and repair a "provably secure" untraceable payment system. In *Advances in Cryptology: CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 338–350. Springer, 1992.

[67] B. Pfitzmann and M. Waidner. Composition and integrity preservation of secure reactive systems. In *Proc. 7th ACM Conference on Computer and Communications Security*, pages 245–254, 2000. Extended version (with Matthias Schunter) IBM Research Report RZ 3206, May 2000, `http://www.semper.org/sirene/publ/PfSW1_00ReactSimulIBM.ps.gz`.

[68] B. Pfitzmann and M. Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *Proc. 22nd IEEE Symposium on Security & Privacy*, pages 184–200, 2001.

[69] P. Rogaway. Authenticated-encryption with associated-data. In *Proc. 9th ACM Conference on Computer and Communications Security*, pages 98–107, 2002.

[70] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *Proc. 2nd USENIX Workshop on Electronic Commerce*, pages 29–40, 1996.

[71] A. C. Yao. Theory and applications of trapdoor functions. In *Proc. 23rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 80–91, 1982.

A Postponed Proofs

Proof. (Lemma 3.1) Let S_{A} denote the set of specified ports the adversary connects to, i.e.,

$$S_{\mathsf{A}} := \{p \mid p \in S \setminus \mathsf{ports}(\mathsf{H})^c\}.$$

Roughly speaking, we will define a new machine H_1 which is inserted between the system and the adversary such that H_1 uses all ports of S_{A}. Combination of H_1 and H will yield the new honest user H_{s}. However, we will at first concentrate on the machine A_{s}.

If the configuration $conf$ is polynomial-time, let the adversary A be bounded by $L(k)$ for a polynomial L and the security parameter k. We now define the new adversary A_{s} of $conf_{\mathsf{s}}$ starting with its ports.

- First of all, every port $p \in \mathsf{ports}(\mathsf{A})$ that does not connect to a specified port, i.e. $p^c \notin S_{\mathsf{A}}$, is also a port of $\mathsf{A_s}$.
- For every simple port $p \in \mathsf{ports}(\mathsf{A})$ with $p^c \in S_{\mathsf{A}}$, $\mathsf{A_s}$ has a port p' of the same kind.
- For every clock-out port $\mathsf{p}^{\triangleleft}! \in \mathsf{ports}(\mathsf{A})$ that connects to the specified ports, i.e. $\mathsf{p}^{\triangleleft}!^c \in S_{\mathsf{A}}$, $\mathsf{A_s}$ has a clock-out port $\mathsf{p_{cr}}^{\triangleleft}!$ and an additional output port $\mathsf{p_{cr}}!$.[5]
- $\mathsf{A_s}$ has additional ports $\mathsf{p}_{\mathsf{A_s}}?, \mathsf{p}_{\mathsf{H}_1}!, \mathsf{p}_{\mathsf{H}_1}{}^{\triangleleft}!$ which will be needed for synchronizing the communication with H_1 and ports $\mathsf{p_{mask\_back}}?, \mathsf{p_{mask}}!, \mathsf{p_{mask}}^{\triangleleft}!$ needed to make the machine H_1 polynomial-time in case of a polynomial-time configuration $conf$.

We assume without loss of generality that all these primed and additional ports are new ports of the configuration. Internally, $\mathsf{A_s}$ maintains an array $\mathcal{O}'\_save = (O'\_save_{\mathsf{p}^{\triangleleft}!})_{\mathsf{p}^{\triangleleft}! \in \mathsf{ports}(\mathsf{A_s})}$ over Σ^+ initialized with ϵ everywhere and two arrays $(out\_buff_{\mathsf{p}!})_{\mathsf{p}! \in S^c_{\mathsf{A}}}$ and $(masked_{\mathsf{p}?})_{\mathsf{p}? \in S^c_{\mathsf{A}}}$ over $\{0,1\}$ initialized with 0 everywhere.

The array out_buff will be used to indicate the buffers between A and the corresponding specified ports of the system which have nonempty contents. The array $masked$ will be used to explicitly tell H_1 which input ports it has to "cut off", i.e., which ports it has to mask with a length bound 0.

The behavior of $\mathsf{A_s}$ is now defined as follows. On an arbitrary given output $\mathcal{O} = (O_p)_{p \in \mathsf{ports}(\mathsf{A})}$ and the given state s' of the blackbox A, the corresponding output $\mathcal{O}' = (O'_p)_{p \in \mathsf{ports}(\mathsf{A_s})}$ of $\mathsf{A_s}$ is derived by the following algorithm. Initially, all components of $\mathcal{O}'$ are set to ϵ. They are automatically reset to ϵ at every call of δ_{A}.

- *Preliminary Step:* First of all, $\mathsf{A_s}$ checks whether A masked one of its own input ports connected to the specified ports using a zero length bound, which it can easily do, because it knows the current state s' of the blackbox. For every masked input port $\mathsf{p}? \notin S^c_{\mathsf{A}}$, i.e., a port connected to an unspecified port of the system, it masks this input too; for every non-masked input port of this kind, it sets the length bound to the runtime of A in the polynomial case and to infinity in the remaining cases. For every masked input port $\mathsf{p}? \in S^c_{\mathsf{A}}$, it sets $masked_{\mathsf{p}?} := 1$. Afterwards, it sets $\mathcal{O}\_save := \mathcal{O}$ (i.e., it saves the whole output tuple, since it will need the tuple after it has been scheduled by H_1 again) and encodes the whole array $masked$ into $c \in \Sigma^*$. Finally, it outputs c at $\mathsf{p_{mask}}!$, 1 at $\mathsf{p_{mask}}^{\triangleleft}!$. Informally speaking, $\mathsf{A_s}$ has to tell H_1 which ports it should mask, so it stores them in the array $masked$ and sends the whole array to H_1.
- *Step 1:* $\mathsf{p}! \notin S^c_{\mathsf{A}}$: At first, $\mathsf{A_s}$ simply goes through the tuple and sets $O'_{\mathsf{p}!} = O_{\mathsf{p}!}$ for every port $\mathsf{p}!$ with $\mathsf{p}! \notin S^c_{\mathsf{A}}$. This case ensures that outputs to itself, to the system, and to the original honest user H will simply be forwarded.
- *Step 2:* $\mathsf{p}! \in S^c_{\mathsf{A}}$: Then, $\mathsf{A_s}$ goes through the tuple and sets $O'_{\mathsf{p}'!} = O_{\mathsf{p}!}$ for every port $\mathsf{p}!$ with $\mathsf{p}! \in S^c_{\mathsf{A}}$. If $O_{\mathsf{p}!} \neq \epsilon$, $\mathsf{A_s}$ additionally sets $out\_buff_{\mathsf{p}'!} := 1$, i.e., it stores which buffers between $\mathsf{A_s}$ and H_1 have nonempty content.

So far we have considered outputs at the simple ports of A. Now $\mathsf{A_s}$ goes through the tuple and searches for the first nonempty output at a clock-out port $\mathsf{p}^{\triangleleft}!$.

- *Step 3:* $\mathsf{p}^{\triangleleft}! \in S^c_{\mathsf{A}}$: If A outputs c at a clock-out port $\mathsf{p}^{\triangleleft}! \in S^c_{\mathsf{A}}$, A' encodes c and the whole array out_buff_p into $c' \in \Sigma^+$. It then sets $O'_{\mathsf{p_{cr}}!} = c'$, $O'_{\mathsf{p_{cr}}^{\triangleleft}!} = 1$, and $out\_buff_p = 0$ for all elements of

[5] The index $_{\mathsf{cr}}$ serves as an abbreviation for "clocking request". These ports will later be used to tell H_1 which buffer it has to schedule.

the array and outputs $\mathcal{O}'$. Informally speaking, $\mathsf{A_s}$ tells H_1 what buffer have nonempty contents at the moment, and that it should schedule the c-th message of buffer $\widetilde{\mathsf{p}}$ afterwards.

- *Step 4:* $\mathsf{p}^{\triangleleft}! \notin S_{\mathsf{A}}^c$ *or no non-empty clock output at all* : If A outputs c at $\mathsf{p}^{\triangleleft}!$ with ${\mathsf{p}^{\triangleleft}!}^c \notin S_{\mathsf{A}}$, $\mathsf{A_s}$ encodes the whole array out_buff_p into $c' \in \Sigma^+$ as in the previous step but containing the number 0 instead of the number c. It then sets $O'_{\mathsf{p}_{\mathsf{H}_1}!} := c'$, $O'_{{\mathsf{p}_{\mathsf{H}_1}}^{\triangleleft}!} := 1$, $O'\_save_{\mathsf{p}^{\triangleleft}!} := c$, and $out\_buff_p = 0$ for all elements of the array and outputs $\mathcal{O}'$.

  We again briefly sketch the intuition behind this case. Messages intended for the system are directly output, but no message is immediately scheduled. Again, $\mathsf{A_s}$ tells H_1 all necessary information for delivering messages to the specified ports, but additionally, it stores which buffer it has to schedule afterwards. Anticipating, H_1 will give back control to $\mathsf{A_s}$ by construction after he delivered the messages to the specified ports, so $\mathsf{A_s}$ will be able to schedule the desired buffer $\widetilde{\mathsf{p}}$.

  If there is no nonempty clock output, $\mathsf{A_s}$ acts identically but sets $O'\_save_{\mathsf{p}^{\triangleleft}!} := \epsilon$ instead. This ensures that no buffer will be scheduled after the control comes back from H_1 to $\mathsf{A_s}$, so the master scheduler will be scheduled just as in the original configuration $conf$.

The behavior of $\mathsf{A_s}$ on external inputs can be described quite simply.

- If $\mathsf{A_s}$ receives an input 1 at $\mathsf{p}_{\mathsf{A_s}}?$ (i.e., the machine H_1 gives back the control), it simply outputs $\mathcal{O}'\_save$ and sets $O'\_save_{\mathsf{p}^{\triangleleft}!} = \epsilon$ afterwards for all elements of the array. This case can only occur as a direct consequence of Step 4 of the above algorithm. Inputs at other ports are simply forwarded to their corresponding ports of A.

- If $\mathsf{A_s}$ receives an input 1 at $\mathsf{p}_{\mathsf{mask\_back}}?$ it sets all components of $masked$ back to 0 and $\mathcal{O} := \mathcal{O}\_save$ and proceeds with Step 1.

- If A enters final state, we define that $\mathsf{A_s}$ finishes the delivering of messages and enters final state too. More precisely, it outputs its tuple derived by the above algorithm and stops. If Step 4 applies, it additionally waits for a nonempty input at $\mathsf{p}_{\mathsf{A_s}}?$, outputs the tuple $\mathcal{O}'\_save$, i.e., the scheduling of the desired buffer, and enters final state after that.

Note, that $\mathsf{A_s}$ obviously can only do a polynomial number of steps between two successive calls of δ_{A} by construction which yields a polynomial-time adversary $\mathsf{A_s}$ again if A is polynomial.

We can now turn our attention to the machine H_1 which is defined as follows. Its ports are given by

- $\{\mathsf{p} \mid \mathsf{p}^c \in S_{\mathsf{A}}\}$: Ports for connecting to the specified ports S_{A}.
- $\{\mathsf{p}'?, {\mathsf{p}'}^{\triangleleft}! \mid \mathsf{p}?^c \in S_{\mathsf{A}}\}$: Input ports for connecting to $\mathsf{A_s}$.
- $\{\mathsf{p}'!, {\mathsf{p}'}^{\triangleleft}! \mid \mathsf{p}!^c \in S_{\mathsf{A}}\}$: Output ports for connecting to $\mathsf{A_s}$.
- $\{\mathsf{p}_{\mathsf{cr}}? \mid {\mathsf{p}^{\triangleleft}!}^c \in S_{\mathsf{A}}\}$: Input ports for clocking requests of $\mathsf{A_s}$.
- $\{\mathsf{p}_{\mathsf{H}_1}?, \mathsf{p}_{\mathsf{A_s}}!, {\mathsf{p}_{\mathsf{A_s}}}^{\triangleleft}!\}$: Ports for synchronization with $\mathsf{A_s}$.
- $\{\mathsf{p}_{\mathsf{mask}}?, \mathsf{p}_{\mathsf{mask\_back}}!, {\mathsf{p}_{\mathsf{mask\_back}}}^{\triangleleft}!\}$: Ports for making explicit changes of length bounds. As already described above, these ports will be used for masking certain inputs.

Internally, H_1 maintains an array $(buff\_coll_{\mathsf{p}?})_{\mathsf{p}? \in S_{\mathsf{A}}}$ over Σ^+ initialized with ϵ everywhere. The behavior of H_1 is defined as follows.

- If H_1 receives an input c at $\mathsf{p}_{\mathsf{mask}}?$ it decomposes c into the array $masked$ again. For every $masked_{\mathsf{p}?} = 1$ it masks the input port $\mathsf{p}?$ using a zero length bound. For every $masked_{\mathsf{p}?} = 0$ it sets the length bound of $\mathsf{p}?$ to the runtime of A in the polynomial case; otherwise, it sets it to infinity.

- If H_1 receives an input c at a port $\mathsf{p}?$, it outputs c at $\mathsf{p}'!$, 1 at ${\mathsf{p}'}^{\triangleleft}!$. This case ensures that outputs made by system are simply forwarded to the adversary.

- If H_1 receives an input c' at $\mathsf{p}_{\mathsf{cr}}?$, it decomposes c' into its original form $c' = c, (out\_buff_{\mathsf{p}!})_{\mathsf{p}! \in S^c_{\mathsf{A}}}$.

  - In case $c \neq 0$, it does the following: For every element $out\_buff_{\mathsf{p}!} \neq 0$ it schedules the message stored in $\widetilde{\mathsf{p}'}$ and saves them in $buff\_coll_{\mathsf{p}?}$.[6] After that, H_1 outputs the array $buff\_coll_{\mathsf{p}?}$ to the corresponding output ports $\mathsf{p}!$ and removes these elements from the array (which yields an empty array again). Additionally, it outputs c at ${\mathsf{p}}^{\triangleleft}!$ (the corresponding clocking port for requests at $\mathsf{p}_{\mathsf{cr}}?$).
  - In case $c = 0$, it collects all messages stored in the buffers $\widetilde{\mathsf{p}'}$ in $buff\_coll_{\mathsf{p}?}$ again as in the previous step. Finally, it outputs these messages at their corresponding ports and 1 at $\mathsf{p}_{\mathsf{A_s}}!$, 1 at ${\mathsf{p}_{\mathsf{A_s}}}^{\triangleleft}!$. This case ensures that the adversary A will be scheduled again, so he can eventually schedule its desired buffer (cf. Step 4 of the description of $\mathsf{A_s}$).

If the configuration $conf$ is polynomial-time, we let H_1 also stop after a polynomial number of steps. A possible polynomial bound can simply be derived if you consider that H_1 has to make less than $|\mathsf{ports}(\mathsf{A_s})|$ outputs for collecting messages from the nonempty buffer. These messages are stored in the corresponding arrays and finally output as a tuple. The number of ports is finite and does not depend on the security parameter k, so the number of steps which H_1 performs between two successive clockings of itself in every run is constant, because masking of input ports is done not only by A but also H_1. Moreover, H_1 can only be clocked either by the system or by the adversary. If it is clocked by the system it immediately clocks $\mathsf{A_s}$ which has to be polynomial-time if A is polynomial-time as we showed above. Thus, H_1 can only perform a constant number of steps between two successive clockings of $\mathsf{A_s}$. If we denote this constant by cst, H_1 simply stops after $\mathsf{cst} \cdot L_{\mathsf{A_s}}(k)$ steps where the polynomial $L_{\mathsf{A_s}}(k)$ bounds the number of steps $\mathsf{A_s}$ can perform.

Putting it all together, H_1 and $\mathsf{A_s}$ simply forward every message between the system Sys and the original adversary A which is represented as a blackbox submachine of the newly defined adversary $\mathsf{A_s}$. Thus, we obtain identical views of the original adversary A, the system Sys, and the honest user H in both configurations. To prove this more formally we could simply go through all possible cases of outputs of A, H, and machines of the system and show that we obtain identical behaviors with respect to the original machines H, A, and the machines of the system in both configurations. We omit it here because it is a rather simple but tedious proof, and we believe that it is already clear by construction of H_1 and $\mathsf{A_s}$ and our above explanations.

As a direct consequence we obtain that the probability of the runs restricted to S does not change, because H_1 always outputs exactly the same tuple to the specified ports as the original A and the view of all machines of the system and the view of H is identical in both configurations. We now combine H and H_1 into one machine $\mathsf{H_s}$. This combination is well-defined in the underlying model and yields a closed collection $\hat{M} \cup \{\mathsf{H_s}, \mathsf{A_s}\}$ again. Moreover, if $conf$ is polynomial-time, H and A are polynomial-time by precondition which implies that $\mathsf{A_s}$ and H_1 are polynomial-time as shown above. Using the combination

[6] This is indeed possible, because the scheduled buffer will schedule H_1 again by construction if it has a nonempty output. This will always be the case, since H_1 will only schedule buffers which he knows to be nonempty.

of two polynomial-time machine yields a polynomial-time machine again (formally proved in [68], we know that $\mathsf{H_s}$ also has to be polynomial-time yielding a polynomial-time configuration

$$conf_{\mathsf{s}} = (\hat{M}_1, S_1, \mathsf{H_a}, \mathsf{A_s}) \in \mathsf{Conf}(Sys)$$

in this case. The view of any set of submachines of $\mathsf{H_s}$ and the probability of the runs restricted to S does not change at combination of machines, which yields

$$view_{conf}(\mathsf{H}) = view_{conf_{\mathsf{s}}}(\mathsf{H}) \text{ and } run_{conf} \lceil_S = run_{conf_{\mathsf{s}}} \lceil_S.$$

Finally, $S^c \subseteq \mathsf{ports}(\mathsf{H_s})$ holds by construction, which finishes our proof. ■