

Deriving Cryptographically Sound Implementations Using Composition and Formally Verified Bisimulation

Michael Backes¹, Christian Jacobi², and Birgit Pfitzmann³

¹ Saarland University, Saarbrücken, Germany
mbackes@cs.uni-sb.de

² IBM Deutschland Entwicklung GmbH, Processor Development 2, Böblingen, Germany
cjacobi@de.ibm.com

³ IBM Zurich Research Laboratory, Rüschlikon, Switzerland
bpf@zurich.ibm.com

Abstract. We consider abstract specifications of cryptographic protocols which are both suitable for formal verification and maintain a sound cryptographic semantics. In this paper, we present the first abstract specification for ordered secure message transmission in reactive systems based on the recently published model of Pfitzmann and Waidner. We use their composition theorem to derive a possible implementation whose correctness additionally involves a classical bisimulation, which we formally verify using the theorem prover PVS. The example serves as the first important case study which shows that this approach is applicable in practice, and it is the first example that combines tool-supported formal proof techniques with the rigorous proofs of cryptography.

Keywords: security, cryptography, formal verification, PVS, simulatability

1 Introduction

Nowadays, security proofs are getting more and more attention both in theory and practice. Some years ago, this field of research only focused on certain cryptographic primitives such as encryption and digital signature schemes. In current research, larger systems like secure channels or fair exchange protocols are to be verified. The main goal researchers are ultimately aiming at is to verify really large systems like whole e-commerce architectures.

If we turn our attention to what already has been done, we can distinguish between two main approaches that unfortunately seem to be rather disjoint. One approach mainly considers the cryptographic aspects of protocols aiming at complete and mathematically rigorous proofs with respect to cryptographic definitions. The other one involves formal methods, so protocols should be verified using formal proof systems or these proofs should even be generated automatically by theorem provers. Usually, these proofs are much trustworthier than hand-made proofs, especially if we consider large protocols using many single steps. The main problem of this approach lies in the necessary abstraction of cryptographic details. This abstraction cannot be completely avoided, since formal methods cannot handle probabilistic behaviours so far, so usually perfect cryptography is assumed (following the approach of Dolev and Yao [4]) in order to make

machine-aided verification possible. However, these abstractions are unfaithful, since no secure implementation is known so far.

Comparing both approaches, we can see that cryptographic proofs are more meaningful in the sense of security but they also have one main disadvantage: cryptographic proofs usually are very long and error-prone even for very small examples like encryption schemes, and moreover have to be done by hand so far. Hence, it seems rather impossible to verify large systems like whole e-commerce architectures by now.

Our approach tries to combine the best of both worlds: We aim at proofs that allow abstractions and the use of verification tools but nevertheless keep a sound cryptographic semantics. For this, we split our system into two layers, the lower one containing cryptographic systems, the higher one hiding all cryptographic details enabling tool-supported proofs. Secure composition with respect to these layers has already been shown by Pfizmann and Waidner in [16], so if we consider a large system and replace a verified abstract subsystem with a cryptographic implementation, we again obtain a secure system if the implementation is proven to be at least as secure as its abstract counterpart.

In this paper we present the first abstract specification for ordered secure message transmission, and we derive a possible implementation serving as the first example of a concrete and secure system derived using the composition theorem from [16]. Moreover, the crucial part of this security proof involves a bisimulation, which we formally verify using the theorem prover PVS [13] yielding a trustworthy proof. Our implementation is based on the scheme for standard secure message transmission presented in [16], but we put a system on top of it to prevent message reordering.

Outline. We recapitulate the underlying model of reactive systems in asynchronous networks in Section 2. Furthermore we briefly review how to express typical trust models and what secure composition of systems means. Sections 3, 4 and 5 contain the main work. In Section 3 we present an abstract specification for ordered secure message transmission, and a possible implementation derived using the composition theorem. In Section 4 we accomplish some preparatory work for proving the security of the implementation, which is performed in Section 5 using the theorem prover PVS. Section 6 summarizes and gives an outlook on future work.

Related Literature. One main goal in the verification of cryptographic protocols is to retain a sound cryptographic semantics and nevertheless provide abstract interfaces in order to make machine-aided verification possible. This goal is pursued by several researchers: our specification for ordered secure message transmission is based on a model recently introduced by Pfizmann and Waidner [16], which we believe to be really close to this goal. Another possible way to achieve this goal has been presented in [7,8]: actual cryptography and security is directly expressed and verified using a formal language (π -calculus), but their approach does neither offer any abstractions nor abstract interfaces that enable tool support. [11] has quite a similar motivation to our underlying model, but it is restricted to the usual equational specifications of cryptographic primitives, the Dolev-Yao model [4], and the semantics is not probabilistic. Moreover, [11] only considers passive adversaries and a restricted class of users, referred to as “environment”. So the abstraction from cryptography is not faithful. This applies also to other formal-methods papers about security, e.g., [9,17,1,14,5]: they are based on intuitive but

unfaithful abstractions, i.e., no secure cryptographic implementation is known. In [2], it is shown that a slight variation of the Dolev-Yao model is cryptographically faithful specifically for symmetric encryption, but only under passive attacks.

As to secure message transmission, several specifications have been proposed, but they are either specific for one concrete protocol or lack abstraction [7]. So far, no model for ordered secure message transmission has been published. Thus, we present the first completely abstract specification and a possible implementation for secure message transmission that prevents message reordering. We furthermore showed that the composition theorem of [16] is in fact applicable in practice. Moreover, our proof contains machine-aided verification, so this paper is the first one that uses formal verification of cryptographic protocols while retaining a sound semantics with respect to the underlying cryptographic primitives.

2 Reactive Systems in Asynchronous Networks

In this section we briefly recapitulate the model for reactive systems in asynchronous networks as introduced in [16]. All details not necessary for understanding are omitted, they can be found in [16]. Machines are represented by probabilistic state-transition machines, similar to probabilistic I/O automata [10]. For complexity we consider every automaton to be implemented as a probabilistic Turing machine; complexity is measured in the length of its initial state, i.e., the initial worktape content (often a security parameter k in unary representation).

2.1 General System Model and Simulatability

Systems are mainly compositions of several machines. Usually we consider real systems that are built by a set \hat{M} of machines $\{M_1, \dots, M_n\}$, and ideal systems built by one machine $\{\text{TH}\}$.

Communication between different machines is done via ports. Inspired by the CSP notation [6], we write output and input ports as $p!$ and $p?$ respectively. The ports of a machine M are denoted by $\text{ports}(M)$. Connections are defined implicitly by naming convention, that is port $p!$ sends messages to $p?$. To achieve asynchronous timing, a message is not directly sent to its recipient, but it is first stored in a special machine \tilde{p} called a buffer and waits to be scheduled. If a machine wants to schedule the i -th message of buffer \tilde{p} (this machine must have the unique clock out-port p^{\triangleleft}) it simply sends i at p^{\triangleleft} . The i -th message is then scheduled by the buffer and removed from its internal list. Usually buffers are scheduled by the adversary, but it is sometimes useful to let other machines schedule certain buffers. This is done by the mentioned clock out-port p^{\triangleleft} .

A *collection* \mathcal{C} of machines is a finite set of machines with pairwise different machine names and disjoint sets of ports. The *completion* $[\mathcal{C}]$ of a collection \mathcal{C} is the union of all machines of \mathcal{C} and the buffers needed for every connection.

A *structure* is a pair (\hat{M}, S) , where \hat{M} is a collection of machines and $S \subseteq \text{free}([\hat{M}])$, the so called *specified ports*, are a subset of the free¹ ports in $[\hat{M}]$. Roughly, the ports

¹A port is called *free* if its corresponding port is not in the collection. These ports will be connected to the users and the adversary.

S guarantee specific services to the honest users. We always describe specified ports by their complements S^c , i.e., the ports honest users should have. A structure can be completed to a *configuration* by adding machines H and A modeling honest users and the adversary. 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. If we now consider a set of structures, we obtain a *system* Sys .

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, a designated master scheduler is scheduled.

Altogether we obtain a probability space of runs (sometimes called *traces* or *executions*) of a configuration $conf$ for each security parameter k . If we restrict these runs to a set \hat{M} of machines, we obtain the *view* of \hat{M} ; this is a random variable denoted by $view_{conf,k}(\hat{M})$.

An important security concept is *simulatability*. Essentially it means that whatever might happen to an honest user H in a real system Sys_{real} can also happen to the same honest user in an ideal System Sys_{id} . Formally speaking, for every configuration $conf_1$ of Sys_{real} there is a configuration $conf_2$ of Sys_{id} yielding indistinguishable views for the same H in both systems [18]. We write this $Sys_{real} \geq_{sec} Sys_{id}$ and say that Sys_{real} is *at least as secure as* Sys_{id} ; indistinguishability of the views of H is denoted by $view_{conf_1}(H) \approx view_{conf_2}(H)$. Usually, only certain “corresponding” structures (\hat{M}_1, S_1) of Sys_{real} and (\hat{M}_2, S_2) of Sys_{id} are compared, in particular we require $S_1 = S_2$. In general, a mapping f may denote this correspondence and one writes \geq_{sec}^f , but if the requirement $S_1 = S_2$ gives a unique one-to-one correspondence, we call the mapping canonical and omit it. This is the case in all our examples.

An important feature of the system model is transitivity of \geq_{sec} , i.e., the preconditions $Sys_1 \geq_{sec} Sys_2$ and $Sys_2 \geq_{sec} Sys_3$ together imply $Sys_1 \geq_{sec} Sys_3$ [16].

2.2 Standard Cryptographic Systems

We now turn our attention to the specific class of standard cryptographic systems with static adversaries. 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 $in_u?$ and $out_u!$, which are specified ports of the system and connect to the user u . A standard cryptographic system Sys can now be derived by a *trust model*, which consists of an access structure ACC and a channel model χ . ACC is a set of subsets \mathcal{H} of $\{1, \dots, n\}$ and denotes the possible sets of correct machines. The channel model classifies every connection as secure (private and authentic), authenticated or insecure. In the given model these changes can easily be done via port renaming [16]. Thus, for each set \mathcal{H} and a fixed channel model, we obtain a modified machine $M_{u,\mathcal{H}}$ for every machine M_u with $u \in \mathcal{H}$. These machines form the structure for the set \mathcal{H} ; the remaining machines are considered part of the adversary.

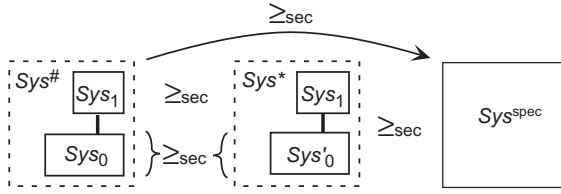


Fig. 1. Composition of Systems.

Ideal systems are typically of the form $Sys_{id} = \{(\{TH_{\mathcal{H}}\}, S_{\mathcal{H}}) \mid \mathcal{H} \in ACC\}$ with the same sets $S_{\mathcal{H}}$ as in the corresponding real system Sys_{real} , i.e., each structure consists of only *one* machine that we usually refer to as *trusted host* $TH_{\mathcal{H}}$, or TH for short.

2.3 Composition

We conclude this section with a briefly 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 real system whereas Sys'_0 is an ideal specification of the real 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^* . 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 1. The composition theorem now states that this replacement maintains security, i.e., $Sys^{\#}$ is at least as secure as Sys^* (see [16] 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 1. Moreover, such specifications are well-suited for formal verification, because single machines are usually much easier to validate. Our specification in Section 3 is of this kind.

3 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. Furthermore, a concrete implementation for this specification is presented according to the composition approach from Section 2.3.

3.1 The Abstract Specification

Our specification is a typical ideal system $Sys^{\text{spec}} = \{(\text{TH}'_{\mathcal{H}}, S_{\mathcal{H}}) \mid \mathcal{H} \in \text{ACC}\}$ as described in Section 2.2 where any number of participants may be dishonest. We start with an intuitive description of how the scheme works.

The ideal machine $\text{TH}'_{\mathcal{H}}$ models initialization, sending and receiving of messages. A user u can initialize communications with other users by inputting a command of the form (snd_init) to the port $\text{in}_u?$ of $\text{TH}'_{\mathcal{H}}$. In the real world, initialization corresponds to key generation and authenticated key exchange. Sending a message to a user v is triggered by a command (send, m, v) . If v is honest, the message is stored in an internal array $\text{deliver}_{u,v}^{\text{spec}}$ of $\text{TH}'_{\mathcal{H}}$ together with a counter indicating the number of the message. After that, the information $(\text{send_blindly}, i, l, v)$ 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 may even see its length. We speak of tolerable imperfections that are explicitly given to the adversary. Because of the asynchronous timing model, $\text{TH}'_{\mathcal{H}}$ has to wait for a special term $(\text{receive_blindly}, v, i)$ or $(\text{rec_init}, u)$ sent by the adversary, signaling that the i th message in $\text{deliver}_{u,v}^{\text{spec}}$ should be delivered to v or that a connection between u and v should be established, respectively. In the first case, $\text{TH}'_{\mathcal{H}}$ reads $(m, j) := \text{deliver}_{u,v}^{\text{spec}}[i]$ and checks whether $j \geq \text{msg_out}_{u,v}^{\text{spec}}$ holds for a message counter $\text{msg_out}_{u,v}^{\text{spec}}$. This test prevents replay and message reordering. If the test is successful the message is delivered and the counter is set to $j + 1$. Otherwise, $\text{TH}'_{\mathcal{H}}$ outputs nothing. The user v receives inputs $(\text{receive}, u, m)$ and $(\text{rec_init}, u)$, respectively.

If v is dishonest, $\text{TH}'_{\mathcal{H}}$ simply outputs (send, m, v) to the adversary. The adversary can also send a message m to a user u by inputting a command $(\text{receive}, v, m)$ to the port $\text{from_adv}_u?$ of $\text{TH}'_{\mathcal{H}}$ for a corrupted user v . Finally, he can stop the machine of any user by sending a command (stop) to $\text{TH}'_{\mathcal{H}}$; this corresponds to exceeding the machine's runtime bounds in the real world.

The length of each message and the number of messages each user may send and receive is bounded by $L(k)$, $s_1(k)$ and $s_2(k)$, respectively, for polynomials L , s_1 , s_2 , and the security parameter k . We furthermore distinguish the *standard ordered system* and the *perfect ordered system*. The standard ordered system only prevents message reordering, but the adversary can still leave out messages. In the perfect ordered system, the adversary can only deliver messages between honest users in exactly the sequence they have been sent. We now give the formal specification of the systems.

Scheme 1 (Specification for Ordered Secure Message Transmission) Let $n \in \mathbb{N}$ and polynomials $L, s_1, s_2 \in \mathbb{N}[x]$ be given, and let Σ denote the message alphabet, len the length of strings, and \downarrow an undefined value. Let $\mathcal{M} := \{1, \dots, n\}$ denote the set of possible participants, and let the access structure ACC be the powerset of \mathcal{M} . Our specification for ordered secure message transmission is a standard ideal system

$$Sys_{n,L,s_1,s_2}^{\text{msg_ord,spec}} = \{(\{\text{TH}'_{\mathcal{H}}\}, S_{\mathcal{H}}) \mid \mathcal{H} \subseteq \mathcal{M}\}$$

with $S_{\mathcal{H}}^c := \{\text{in}_u!, \text{out}_u?, \text{in}_u^{\downarrow}! \mid u \in \mathcal{H}\}$ and $\text{TH}'_{\mathcal{H}}$ defined as follows. When \mathcal{H} is clear from the context, let $\mathcal{A} := \mathcal{M} \setminus \mathcal{H}$ denote the indices of corrupted machines.

The ports of the machine $\text{TH}'_{\mathcal{H}}$ are $\{\text{in}_u?, \text{out}_u!, \text{out}_u^{\triangleleft}! \mid u \in \mathcal{H}\} \cup \{\text{from_adv}_u?, \text{to_adv}_u!, \text{to_adv}_u^{\triangleleft}! \mid u \in \mathcal{H}\}$. Internally, $\text{TH}'_{\mathcal{H}}$ maintains seven arrays:

- $(\text{init}_{u,v}^{\text{spec}})_{u,v \in \mathcal{M}}$ over $\{0, 1\}$ for modeling initialization of users,
- $(\text{sc_in}_{u,v}^{\text{spec}})_{u \in \mathcal{H}, v \in \mathcal{M}}$ over $\{0, \dots, s_1(k)\}$ for counting how often $\text{TH}'_{\mathcal{H}}$ has been switched by user u using messages intended for v ,
- $(\text{msg_out}_{u,v}^{\text{spec}})_{u,v \in \mathcal{H}}$ over $\{0, \dots, s_2(k)\}$ for storing the number of the next expected message (cf. the description above),
- $(\text{sc_out}_{u,v}^{\text{spec}})_{u \in \mathcal{M}, v \in \mathcal{H}}$ over $\{0, \dots, s_2(k)\}$ for counting how often $\text{TH}'_{\mathcal{H}}$ has been switched by the adversary for delivering a message from user u to user v ,
- $(\text{msg_in}_{u,v}^{\text{spec}})_{u \in \mathcal{H}, v \in \mathcal{M}}$ over $\{0, \dots, s_1(k)\}$ for counting the incoming messages from u intended for v ,
- $(\text{stopped}_u^{\text{spec}})_{u \in \mathcal{H}}$ over $\{0, 1\}$ for storing whether the machine of user u has already been stopped, i.e., reached its runtime bounds,
- $(\text{deliver}_{u,v}^{\text{spec}})_{u,v \in \mathcal{H}}$ of lists for storing the actual messages.

The first six arrays are initialized with 0 everywhere, except that $\text{msg_out}_{u,v}^{\text{spec}}$ is initialized with 1 everywhere. The last array should be initialized with empty lists everywhere. Roughly, the five arrays $\text{init}_{u,v}^{\text{spec}}$, $\text{msg_out}_{u,v}^{\text{spec}}$, $\text{msg_in}_{u,v}^{\text{spec}}$, $\text{stopped}_u^{\text{spec}}$, and $\text{deliver}_{u,v}^{\text{spec}}$ ensure functional correctness, whereas the arrays $\text{sc_in}_{u,v}^{\text{spec}}$ and $\text{sc_out}_{u,v}^{\text{spec}}$ help to make the system polynomial-time: the machine $\text{TH}'_{\mathcal{H}}$ ignores certain inputs as soon as these counters reach the given bounds $s_1(k)$ or $s_2(k)$, respectively. The state-transition function of $\text{TH}'_{\mathcal{H}}$ is defined by the following rules, written in a pseudo-code language. For the sake of readability, we exemplarily annotate the “Send initialization” transition, i.e., the key generation in the real world.

- **Send 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 (`snd_init`) to $\text{TH}'_{\mathcal{H}}$. Now, the system checks that the user has not already reached his message bound (which is quite improbable in this case unless he tried to send trash all the time), that the machine itself has not reached its runtime bound, and that no key generation of this user has already occurred in the past. These three checks correspond to $\text{sc_in}_{u,v}^{\text{spec}} < s_1(k)$ for all $v \in \mathcal{M}$, $\text{stopped}_u^{\text{spec}} = 0$, and $\text{init}_{u,u}^{\text{spec}} = 0$, respectively. If at least the check of the message bound (i.e., $\text{sc_in}_{u,v}^{\text{spec}} < s_1(k)$) holds, the counter $\text{sc_in}_{u,v}^{\text{spec}}$ is increased. If all three checks hold, the keys are distributed over authenticated channels, modeled by an output (`snd_init`) to the adversary which either can schedule them immediately, later or even leave them on the channels forever. In our pseudo-code language this is expressed as follows:

On input (`snd_init`) at $\text{in}_u?$: If $\text{sc_in}_{u,v}^{\text{spec}} < s_1(k)$ for all $v \in \mathcal{M}$, set $\text{sc_in}_{u,v}^{\text{spec}} := \text{sc_in}_{u,v}^{\text{spec}} + 1$ for all $v \in \mathcal{M}$, otherwise do nothing. If the test holds check $\text{stopped}_u^{\text{spec}} = 0$ and $\text{init}_{u,u}^{\text{spec}} = 0$. In this case set $\text{init}_{u,u}^{\text{spec}} := 1$ and output (`snd_init`) at $\text{to_adv}_u!$, 1 at $\text{to_adv}_u^{\triangleleft}!$.

The following parts should now be understood similarly:

- **Receive initialization:** On input $(\text{rec_init}, u)$ at $\text{from_adv}_v?$ with $u \in \mathcal{M}, v \in \mathcal{H}$: If $\text{stopped}_v^{\text{spec}} = 0$, $\text{init}_{u,v}^{\text{spec}} = 0$, and $[u \in \mathcal{H} \Rightarrow \text{init}_{u,u}^{\text{spec}} = 1]$, set $\text{init}_{u,v}^{\text{spec}} := 1$. If $\text{sc_out}_{u,v}^{\text{spec}} < s_2(k)$ set $\text{sc_out}_{u,v}^{\text{spec}} := \text{sc_out}_{u,v}^{\text{spec}} + 1$, output $(\text{rec_init}, u)$ at $\text{out}_v!$, 1 at $\text{out}_v^{\triangleleft!}$.
- **Send:** On input (send, m, v) at $\text{in}_u?$: If $\text{sc_in}_{u,v}^{\text{spec}} < s_1(k)$ and $\text{stopped}_u^{\text{spec}} = 0$, set $\text{sc_in}_{u,v}^{\text{spec}} := \text{sc_in}_{u,v}^{\text{spec}} + 1$, otherwise do nothing. If $m \in \Sigma^+, l := \text{len}(m) \leq L(k)$, $v \in \mathcal{M} \setminus \{u\}$, $\text{init}_{u,u}^{\text{spec}} = 1$ and $\text{init}_{v,u}^{\text{spec}} = 1$ holds:
If $v \in \mathcal{A}$ then $\{ \text{set } \text{msg_in}_{u,v}^{\text{spec}} := \text{msg_in}_{u,v}^{\text{spec}} + 1 \text{ and output } (\text{send}, (m, \text{msg_in}_{u,v}^{\text{spec}}), v) \text{ at } \text{to_adv}_u!$, 1 at $\text{to_adv}_u^{\triangleleft!} \}$ else $\{ \text{set } i := \text{size}(\text{deliver}_{u,v}^{\text{spec}}) + 1$, $\text{msg_in}_{u,v}^{\text{spec}} := \text{msg_in}_{u,v}^{\text{spec}} + 1$, $\text{deliver}_{u,v}^{\text{spec}}[i] := (m, \text{msg_in}_{u,v}^{\text{spec}})$ and output $(\text{send_blindly}, i, l, v)$ at $\text{to_adv}_u!$, 1 at $\text{to_adv}_u^{\triangleleft!} \}$.
- **Receive from honest party u :** On input $(\text{receive_blindly}, u, i)$ at $\text{from_adv}_v?$ with $u, v \in \mathcal{H}$: If $\text{stopped}_v^{\text{spec}} = 0$, $\text{init}_{v,v}^{\text{spec}} = 1$, $\text{init}_{u,v}^{\text{spec}} = 1$, $\text{sc_out}_{u,v}^{\text{spec}} < s_2(k)$ and $(m, j) := \text{deliver}_{u,v}^{\text{spec}}[i] \neq \downarrow$, check $j \geq \text{msg_out}_{u,v}^{\text{spec}}$ ($j = \text{msg_out}_{u,v}^{\text{spec}}$ in the perfect ordered system). If this holds set $\text{sc_out}_{u,v}^{\text{spec}} := \text{sc_out}_{u,v}^{\text{spec}} + 1$, $\text{msg_out}_{u,v}^{\text{spec}} := j + 1$ and output $(\text{receive}, u, m)$ at $\text{out}_v!$, 1 at $\text{out}_v^{\triangleleft!}$.
- **Receive from dishonest party u :** On input $(\text{receive}, u, m)$ at $\text{from_adv}_v?$ with $u \in \mathcal{A}, m \in \Sigma^+, \text{len}(m) \leq L(k)$ and $v \in \mathcal{H}$: If $\text{stopped}_v^{\text{spec}} = 0$, $\text{init}_{v,v}^{\text{spec}} = 1$, $\text{init}_{u,v}^{\text{spec}} = 1$ and $\text{sc_out}_{u,v}^{\text{spec}} < s_2(k)$, set $\text{sc_out}_{u,v}^{\text{spec}} := \text{sc_out}_{u,v}^{\text{spec}} + 1$ and output $(\text{receive}, u, m)$ at $\text{out}_v!$, 1 at $\text{out}_v^{\triangleleft!}$.
- **Stop:** On input (stop) at $\text{from_adv}_u?$ with $u \in \mathcal{H}$: If $\text{stopped}_u^{\text{spec}} = 0$, set $\text{stopped}_u^{\text{spec}} := 1$ and output (stop) at $\text{out}_u!$, 1 at $\text{out}_u^{\triangleleft!}$.

Finally, if $\text{TH}'_{\mathcal{H}}$ receives an input at a port $\text{in}_u?$ which is not comprised by the above six transitions (i.e., the user sends some kind of trash), it increases the counter $\text{sc_in}_{u,v}^{\text{spec}}$ for all $v \in \mathcal{M}$. Similarly, if $\text{TH}'_{\mathcal{H}}$ receives such an input at a port $\text{from_adv}_v?$ it increases every counter $\text{sc_out}_{u,v}^{\text{spec}}$ for $u \in \mathcal{M}$. \diamond

Thus, at least one counter $\text{sc_in}_{u,v}^{\text{spec}}$ or $\text{sc_out}_{u,v}^{\text{spec}}$ is increased in each transition of $\text{TH}'_{\mathcal{H}}$, and each transition can obviously be realized in polynomial-time, so the machine $\text{TH}'_{\mathcal{H}}$ is polynomial-time.

$\text{Sys}_{n,L,s_1,s_2}^{\text{msg_ord,spec}}$ 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 be expressed in formal languages, e.g., in PVS. In the following we simply write $\text{Sys}_{n,L,s_1,s_2}^{\text{msg_ord,spec}}$ instead of $\text{Sys}_{n,L,s_1,s_2}^{\text{msg_ord,spec}}$ if the parameters n, L, s_1, s_2 are not necessary for understanding.

3.2 The Split Ideal System

This section contains the first step for deriving a real system that is as secure as Scheme 1. If we take a look at Figure 1, the system $\text{Sys}_{n,L,s_1,s_2}^{\text{msg_ord,spec}}$ plays the role of the monolithic specification Sys^{spec} . We now “split” our specification into a system Sys^* such that $\text{Sys}^* \geq_{\text{sec}} \text{Sys}^{\text{spec}}$ holds. Sys^* is the combination of two systems Sys'_0 and Sys_1 . Finally, we replace Sys'_0 with Sys_0 using the composition theorem and obtain a real system that still fulfills our requirements.

The systems Sys'_0 and Sys_0 are the ideal and real systems for secure message transmission presented in [16]. Sys_1 filters messages that are out of order; we define it next, see also Figure 2.

Scheme 2 (Filtering System Sys_1) Let $n, L, s_1, s_2, \mathcal{M}$ be given as in Scheme 1. Furthermore let a polynomial $L_1 := L + c(k)$ be given; the value of $c(k)$ is explained below. Sys_1 is now defined as

$$Sys_1 = \{(\hat{M}'_{\mathcal{H}}, S_{\mathcal{H}}) \mid \mathcal{H} \subseteq \mathcal{M}\},$$

where $\hat{M}'_{\mathcal{H}} = \{M'_u \mid u \in \mathcal{H}\}$ and $\text{ports}(M'_u) = \{\text{in}_u?, \text{out}_u!, \text{out}_u^{\triangleleft!}\} \cup \{\text{in}'_u!, \text{out}'_u?, \text{in}'_u^{\triangleleft!}\}$. All free ports of $[\hat{M}'_{\mathcal{H}}]$ are specified, i.e., $S_{\mathcal{H}}$ consists of all ports corresponding to $\text{ports}(\hat{M}'_{\mathcal{H}})$. Internally, the machine M'_u maintains two arrays $(\text{msg_in}_{u,v}^{\text{id}})_{v \in \mathcal{M}}, (\text{sc_in}_{u,v}^{\text{id}})_{v \in \mathcal{M}}$ over $\{0, \dots, s_1(k)\}$ and two arrays $(\text{msg_out}_{v,u}^{\text{id}})_{v \in \mathcal{M}}, (\text{sc_out}_{v,u}^{\text{id}})_{v \in \mathcal{M}}$ over $\{0, \dots, s_2(k)\}$. All four arrays are initialized with 0 everywhere. Moreover, it contains a flag $(\text{stopped}_u^{\text{id}})$ over $\{0, 1\}$ initialized with 0. We assume that encoding of tuples has the following straightforward length property: $\text{len}((m, \text{num})) = \text{len}(m) + c(k)$ for every $\text{num} \in \{0, \dots, \max\{s_1(k), s_2(k)\}\}$ and an arbitrary function c , i.e., $\text{len}(\text{num})$ is constant for each fixed security parameter k . This condition can easily be achieved by padding all values num to a fixed size $\geq \text{len}(\max\{s_1(k), s_2(k)\})$. The behaviour of M'_u is defined as follows.

- **Send initialization:** On input (snd_init) at $\text{in}_u?$: If $\text{sc_in}_{u,v}^{\text{id}} < s_1(k)$ for every $v \in \mathcal{M}$, set $\text{sc_in}_{u,v}^{\text{id}} := \text{sc_in}_{u,v}^{\text{id}} + 1$ for every $v \in \mathcal{M}$. If $\text{stopped}_u^{\text{id}} = 0$ then output (snd_init) at $\text{in}'_u!, 1$ at $\text{in}'_u^{\triangleleft!}$.
- **Receive initialization:** On input $(\text{rec_init}, v)$ at $\text{out}'_u?$: If $\text{stopped}_u^{\text{id}} = 0$ and $\text{sc_out}_{v,u}^{\text{id}} < s_2(k)$, set $\text{sc_out}_{v,u}^{\text{id}} := \text{sc_out}_{v,u}^{\text{id}} + 1$ and output $(\text{rec_init}, v)$ at $\text{out}_u!, 1$ at $\text{out}_u^{\triangleleft!}$.
- **Send:** On input (send, m, v) at $\text{in}_u?$: If $\text{stopped}_u^{\text{id}} = 0$ and $\text{sc_in}_{u,v}^{\text{id}} < s_1(k)$, set $\text{sc_in}_{u,v}^{\text{id}} := \text{sc_in}_{u,v}^{\text{id}} + 1, \text{msg_in}_{u,v}^{\text{id}} := \text{msg_in}_{u,v}^{\text{id}} + 1$ and output $(\text{send}, (m, \text{msg_in}_{u,v}^{\text{id}}), v)$ at $\text{in}'_u!, 1$ at $\text{in}'_u^{\triangleleft!}$.
- **Receive:** On input $(\text{receive}, v, m')$ at $\text{out}'_u?$: If $\text{stopped}_u^{\text{id}} = 0$ and $\text{sc_out}_{v,u}^{\text{id}} < s_2(k)$, set $\text{sc_out}_{v,u}^{\text{id}} := \text{sc_out}_{v,u}^{\text{id}} + 1$, otherwise do nothing. If the test was true, decompose the message m' into (m, num) . If $\text{num} \geq \text{msg_out}_{v,u}^{\text{id}}$ (or $\text{num} = \text{msg_out}_{v,u}^{\text{id}}$ in the perfect ordered system), $\text{msg_out}_{v,u}^{\text{id}} := \text{num} + 1$ and output $(\text{receive}, v, m)$ at $\text{out}_u!, 1$ at $\text{out}_u^{\triangleleft!}$.
- **Stop:** On input (stop) at $\text{out}'_u?$: If $\text{stopped}_u^{\text{id}} = 0$, set $\text{stopped}_u^{\text{id}} := 1$ and output (stop) at $\text{out}_u!, 1$ at $\text{out}_u^{\triangleleft!}$.

Finally, if M'_u receives an input at a port $\text{in}_u?$ which is not comprised by the above five transitions, it increases the counter $\text{sc_in}_{u,v}^{\text{spec}}$ for all $v \in \mathcal{M}$. Similarly, if M'_u receives such an input at port $\text{out}'_u?$ it increases every counter $\text{sc_out}_{v,u}^{\text{spec}}$ for $v \in \mathcal{M}$. \diamond

Obviously, Sys'_0 is polynomial-time for the same reason as $\text{TH}'_{\mathcal{H}}$.

As described above, the system Sys_0 is the ideal system for secure message transmission of [16]. We now describe it in full because we need it for our security proof in Sections 4 and 5. We made a few adaptations, which do not invalidate the proof.

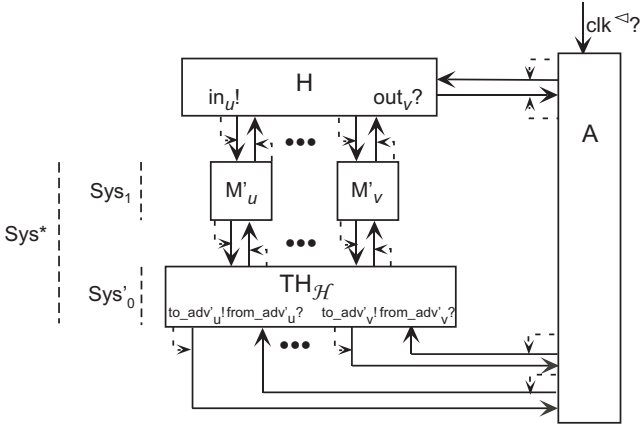


Fig. 2. The Split Ideal System.

Scheme 3 (Ideal System for Unordered Secure Message Transmission) Let n, L_1, \mathcal{M} be given as above. ACC is the powerset of \mathcal{M} . Then

$$Sys'_0 := \{(\{TH_{\mathcal{H}}\}, S_{\mathcal{H}}) \mid \mathcal{H} \subseteq \mathcal{M}\}$$

with $S_{\mathcal{H}}^c := \{in'_u^!, out'_u^?, in'_u^{\triangleleft} \mid u \in \mathcal{H}\}$ and $TH_{\mathcal{H}}$ defined as follows. The ports of $TH_{\mathcal{H}}$ are $\{in'_u^!, out'_u^!, out'_u^{\triangleleft}, from_adv'_u^?, to_adv'_u^!, to_adv'_u^{\triangleleft} \mid u \in \mathcal{H}\}$. $TH_{\mathcal{H}}$ maintains arrays $(init_{u,v}^*)_{u,v \in \mathcal{M}}$ and $(stopped_{u,v}^*)_{u,v \in \mathcal{H}}$ over $\{0, 1\}$, both initialized with 0 everywhere, and an array $(deliver_{u,v}^*)_{u,v \in \mathcal{H}}$ of lists, all initially empty. The state-transition function of $TH_{\mathcal{H}}$ is defined by the following rules:

- **Send initialization.** On input (snd_init) at $in'_u^?$: If $stopped_{u,u}^* = 0$ and $init_{u,u}^* = 0$, set $init_{u,u}^* := 1$ and output (snd_init) at $to_adv'_u^!$, 1 at $to_adv'_u^{\triangleleft}$.
- **Receive initialization.** On input (rec_init, u) at $from_adv'_v^?$ with $u \in \mathcal{M}, v \in \mathcal{H}$: If $stopped_{u,v}^* = 0$ and $init_{u,v}^* = 0$ and $[u \in \mathcal{H} \Rightarrow init_{u,u}^* = 1]$, set $init_{u,v}^* := 1$ and output (rec_init, u) at $out'_v^!$, 1 at out'_v^{\triangleleft} .
- **Send.** On input (send, m, v) at $in'_u^?$ with $m \in \Sigma^+, l := \text{len}(m) \leq L_1(k)$, and $v \in \mathcal{M} \setminus \{u\}$: If $stopped_{u,u}^* = 0$, $init_{u,u}^* = 1$, and $init_{v,u}^* = 1$: If $v \in \mathcal{A}$ then { output (send, m, v) at $to_adv'_u^!$, 1 at $to_adv'_u^{\triangleleft}$ }, else { $i := \text{size}(deliver_{u,v}^*) + 1$; $deliver_{u,v}^*[i] := m$; output (send_blindly, i, l, v) at $to_adv'_u^!$, 1 at $to_adv'_u^{\triangleleft}$ }.
- **Receive from honest party u .** On input (receive_blindly, u, i) at $from_adv'_v^?$ with $u, v \in \mathcal{H}$: If $stopped_{u,v}^* = 0$, $init_{v,v}^* = 1$, $init_{u,v}^* = 1$, and $m := deliver_{u,v}^*[i] \neq \downarrow$, then output (receive, u, m) at $out'_v^!$, 1 at out'_v^{\triangleleft} .
- **Receive from dishonest party u .** On input (receive, u, m) at $from_adv'_v^?$ with $u \in \mathcal{A}, m \in \Sigma^+, \text{len}(m) \leq L_1(k)$, and $v \in \mathcal{H}$: If $stopped_{u,v}^* = 0$, $init_{v,v}^* = 1$ and $init_{u,v}^* = 1$, then output (receive, u, m) at $out'_v^!$, 1 at out'_v^{\triangleleft} .
- **Stop.** On input (stop) at $from_adv'_u^?$ with $u \in \mathcal{H}$, set $stopped_{u,u}^* = 1$ and output (stop) at $out'_u^!$, 1 at out'_u^{\triangleleft} .

◇

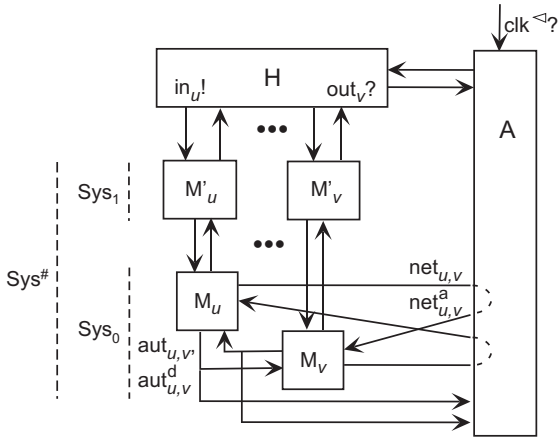


Fig. 3. Sketch of the Real System for Ordered Secure Message Transmission.

If we now combine the two systems Sys'_0 and Sys_1 in the “canonical” way, i.e., we combine those structures with the same index \mathcal{H} , we obtain the system Sys^* , which we call split ideal system (Figure 2). Finally, we define all connections $\{out'_u!, out'_u?\}$ and $\{in'_u!, in'_u?\}$ of Sys^* to be secure, because they correspond to local subroutine calls.

3.3 The Real System

Our real system $Sys^\#$ is derived by replacing Sys'_0 with Sys_0 . For understanding it is sufficient to give a brief review of Sys_0 from [16]. It is a standard cryptographic system of the form $Sys_0 = \{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}}) \mid \mathcal{H} \in ACC\}$, see Figure 3, where $\hat{M}_{\mathcal{H}} = \{M_u \mid u \in \mathcal{H}\}$ and ACC is the powerset of \mathcal{M} , i.e., any subset of participants may be dishonest. It uses asymmetric encryption and digital signatures as cryptographic primitives. 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 machines² and send arbitrary messages m to arbitrary users.

We now build the combination of Sys_1 and Sys_0 in the canonical way, which yields a new system $Sys^\#$ that we refer to as real ordered system.

4 Proving Security of the Real Ordered System

We now start to prove that the real ordered system is at least as secure as the specification. This is captured by the following theorem.

Theorem 1. (*Security of Real Ordered Secure Message Transmission*) For all $n \in \mathbb{N}$ and $s_1, s_2, L \in \mathbb{N}[x]$, $Sys^\# \succeq_{sec}^{poly} Sys^{spec}$ holds (for the canonical mapping), provided

² He can therefore replay messages and also change their order. This is prevented in our scheme by the additional filtering system Sys_1 .

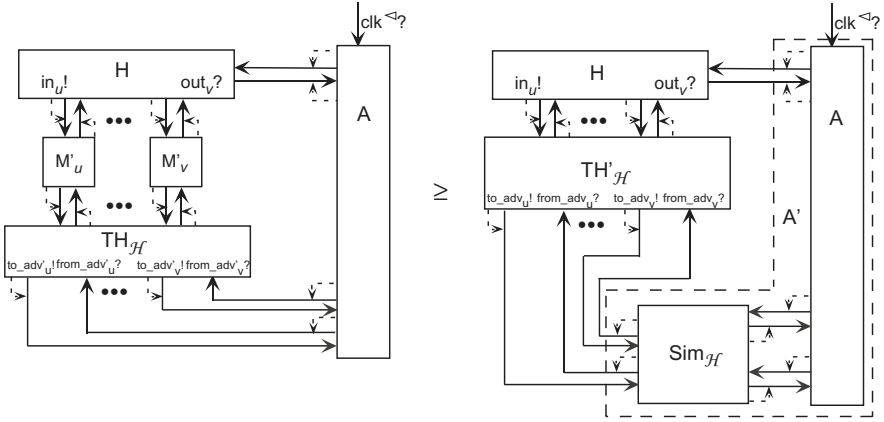


Fig. 4. Proof Overview of $Sys^* \geq_{sec}^{perf} Sys^{spec}$.

the signature and encryption schemes used are secure. This holds with blackbox simulatability.³ \square

Our proof contains the already described four steps, illustrated in Figure 1. First, [16] contains the result $Sys_0 \geq_{sec} Sys'_0$. Secondly, the composition theorem (cf. Section 2.3) yields the relation $Sys^\# \geq_{sec} Sys^*$. The only remaining task is to check that its preconditions are fulfilled, which is straightforward since we showed that the system Sys_1 is polynomial-time. If we have proven $Sys^* \geq_{sec} Sys^{spec}$, then $Sys^\# \geq_{sec} Sys^{spec}$ follows from the transitivity lemma, cf. Section 2.1. Thus, we only have to prove $Sys^* \geq_{sec}^{poly} Sys^{spec}$. We will even prove the perfect case $Sys^* \geq_{sec}^{perf} Sys^{spec}$.

Lemma 1. For all $n \in \mathbb{N}$ and $s_1, s_2, L \in \mathbb{N}[x]$, $Sys^* \geq_{sec}^{perf} Sys^{spec}$ holds (for the canonical mapping), and with blackbox simulatability. \square

In order to prove this, we assume a configuration $conf_{si} := (\{TH_{\mathcal{H}}\} \cup \hat{M}'_u, S_{\mathcal{H}}, H, A)$ of Sys^* with $\hat{M}'_u = \{M'_u \mid u \in \mathcal{H}\}$ to be given, which we call split-ideal configuration. We then have to show that there exists a configuration $conf_{sp} := (\{TH'_{\mathcal{H}}\}, S_{\mathcal{H}}, H, A')$ of Sys^{spec} , called specification configuration, yielding indistinguishable views for the honest user H .

The adversary A' consists of two machines: a so-called simulator $Sim_{\mathcal{H}}$, 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 4.

Definition of the Simulator $Sim_{\mathcal{H}}$. The Simulator $Sim_{\mathcal{H}}$ is placed between the trusted host $TH'_{\mathcal{H}}$ and the adversary A , see Figure 4. Its ports are given by $\{to_adv_u?, from_adv_u!, from_adv_u^{\triangleleft}! \mid u \in \mathcal{H}\} \cup \{from_adv'_u?, to_adv'_u!, to_adv'_u^{\triangleleft}! \mid u \in \mathcal{H}\}$. The first set contains the ports connected to $TH'_{\mathcal{H}}$, the ports of the second set are for communication with the adversary. Internally, $Sim_{\mathcal{H}}$ maintains two arrays $(init_{u,v}^{sim})_{u,v \in \mathcal{M}}$, $(stopped_u^{sim})_{u \in \mathcal{H}}$ over $\{0, 1\}$, an array $(msg_out_{u,v}^{sim})_{u \in A, v \in \mathcal{H}}$

³ See [16] for further details on valid and canonical mappings and different kinds of simulatability.

over $\{0, \dots, s_1(k)\}$, and an array $(sc_out_{u,v}^{sim})_{u \in \mathcal{M}, v \in \mathcal{H}}$ over $\{0, \dots, s_2(k)\}$. All four arrays are initialized with 0 everywhere. They match the arrays in the ideal system, except that $msg_out_{u,v}^{sim}$ corresponds to $msg_out_{u,v}^{id}$ of M'_v for dishonest v only. We now define the behaviour of the simulator. In most cases $Sim_{\mathcal{H}}$ simply forwards inputs to their corresponding outputs, modifying some internal values.

- **Send initialization:** Upon input (snd_init) at $to_adv_u?$, $Sim_{\mathcal{H}}$ sets $init_{u,u}^{sim} := 1$ and outputs (snd_init) at $to_adv_u!$, 1 at $to_adv_u^{\triangleleft!}$.
- **Receive initialization:** Upon input (rec_init, u) at $from_adv_v?$: If $stopped_u^{sim} = 0$ and $init_{u,v}^{sim} = 0$ and $[u \in \mathcal{H} \implies init_{u,u}^{sim} = 1]$ $Sim_{\mathcal{H}}$ sets $init_{u,v}^{sim} := 1$. If additionally $sc_out_{u,v}^{sim} < s_2(k)$ holds, it sets $sc_out_{u,v}^{sim} := sc_out_{u,v}^{sim} + 1$ and outputs (rec_init, u) at $from_adv_v!$, 1 at $from_adv_v^{\triangleleft!}$.
- **Send:** Upon input (send_blindy, i, l', v) at $to_adv_u?$, $Sim_{\mathcal{H}}$ determines $l := l' + c(k)$ and outputs (send_blindy, i, l, v) at $to_adv_u!$, 1 at $to_adv_u^{\triangleleft!}$. Upon input (send, m, v) at $to_adv_u?$, $Sim_{\mathcal{H}}$ simply forwards the input to $to_adv_u!$ and schedules it.
- **Receive from honest party u :** Upon input (receive_blindy, u, i) at $from_adv_v?$, $Sim_{\mathcal{H}}$ forwards this input to port $from_adv_v!$ and schedules it.
- **Receive from dishonest party u :** Upon input (receive, u, m') at $from_adv_v?$ with $u \in \mathcal{A}$, $Sim_{\mathcal{H}}$ decomposes $m' = (m, num)$: If $stopped_v^{sim} = 0$, $init_{v,v}^{sim} = 1$, $init_{u,v}^{sim} = 1$, $len(m') \leq L_1(k)$, $num \geq msg_out_{u,v}^{sim}$ ($num = msg_out_{u,v}^{sim}$ in the perfect ordered system) and $sc_out_{u,v}^{sim} < s_2(k)$, set $msg_out_{u,v}^{sim} := num + 1$, $sc_out_{u,v}^{sim} := sc_out_{u,v}^{sim} + 1$ and output (receive, u, m) at $from_adv_v!$, 1 at $from_adv_v^{\triangleleft!}$.
- **Stop:** On input (stop) at $from_adv_u?$: If $stopped_u^{sim} = 0$, $Sim_{\mathcal{H}}$ sets $stopped_u^{sim} := 1$ and outputs (stop) at $from_adv_u!$, 1 at $from_adv_u^{\triangleleft!}$.

What the simulator does is recalculating the length of message m into $len((m, num))$ 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 $Sim_{\mathcal{H}}$.

Now the ultimate goal is to show that the collections $\hat{M}_* := \{TH_{\mathcal{H}}\} \cup \{M_u \mid u \in \mathcal{H}\}$ and $\hat{M}_{spec} := \{TH'_{\mathcal{H}}, Sim_{\mathcal{H}}\}$ have the same input-output behaviour, i.e., if they obtain the same inputs they produce the same outputs. We do so by proving a classical 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 using the theorem prover PVS.

Definition 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., $free([\hat{M}_1]) = free([\hat{M}_2])$. 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 bisimulation between them. \diamond

We will apply this definition to composed transition functions of each of the two collections \hat{M}_* and \hat{M}_{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 4, 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.

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 very error-prone since they are straightforward on the one hand, but long and tedious on the other, so they 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 web⁴, so readers can make up their own minds.

5 Formal Verification of the Bisimulation

5.1 Defining the Machines in PVS

In this section, we describe how Lemma 1 is formally verified in the theorem proving system PVS [13]. As we already showed in the previous section, it is sufficient to prove that the two collections \hat{M}_* and \hat{M}_{spec} are contained in a deterministic bisimulation. In order to do so, we first describe how the machines are formalized in PVS. Since the formal machine descriptions are too large to be given here completely, we use the machine $\text{TH}'_{\mathcal{H}}$ as an example. The complete machine descriptions and the proof are available online⁴.

We denote the number of participating machines by N , and for a given subset $\mathcal{H} \in \mathcal{ACC}$, we denote the number of honest users by $M := \#\mathcal{H}$. As defined in Scheme 1, the machine $\text{TH}'_{\mathcal{H}}$ has $2M$ input ports $\{\text{in}_u?, \text{from_adv}_u? \mid u \in \mathcal{H}\}$. In

⁴ <http://www-krypt.cs.uni-sb.de/~mbackes/PVS/FME2002/>

PVS, we number these input ports $1, \dots, 2M$, where we identify $1, \dots, M$ with the user ports and $M + 1, \dots, 2M$ with the adversary ports. Similarly, $\text{TH}'_{\mathcal{H}}$ has output ports $\{\text{out}_u!, \text{to_adv}_u! \mid u \in \mathcal{H}\}$, which also are numbered $1, \dots, 2M$. 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, \dots, j . We further define a type `STRING` to represent messages.

In Scheme 1, the different possible inputs to machine $\text{TH}'_{\mathcal{H}}$ are listed, e.g., $(\text{snd_init}), (\text{rec_init}, u), \dots$. In PVS, the type of input ports is defined using a PVS abstract datatype [12]. The prefix `m1i` in the following stands for “inputs of machine 1”, which is $\text{TH}'_{\mathcal{H}}$, and is used to distinguish between inputs and outputs of the different machines.

```
m1_in_port: DATATYPE
BEGIN
  m1i_snd_init:                               m1i_snd_init?
  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 $\text{TH}'_{\mathcal{H}}$ performs a step iff exactly one of the input ports is active. In this case, we call the input *ok*, otherwise *garbage*. Because of our underlying scheduling definition, an input with several active input ports cannot occur so *garbage* naturally correspond to an all-empty input. The type of the complete inputs to $\text{TH}'_{\mathcal{H}}$ 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 $TH'_{\mathcal{H}}$, respectively.

Next we define the state type of $TH'_{\mathcal{H}}$. As defined in Scheme 1, 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 $TH'_{\mathcal{H}}$ is defined as a record of such arrays. There is only one small exception: the array $deliver_{u,v}^{spec}$ 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 $deliver_{u,v}^{spec}$ stores lists of messages m , and the second array $deliv_i_{u,v}^{spec}$ stores lists of naturals i . Altogether, this yields a state type of eight arrays:

```
M1_STATE: TYPE = [# init_spec: [MACH,MACH -> bool],
                  sc_in_spec: [USERS,MACH -> nat],
                  msg_in_spec: [USERS,MACH -> nat],
                  msg_out_spec: [USERS,USERS -> posnat],
                  sc_out_spec: [MACH,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. In the definition of machine $TH'_{\mathcal{H}, sc_{in}_{u,v}^{spec}}$ is incremented for all machines v during the “Send initialization” part. This is encapsulated in the following PVS function:

```
incr_sc_in_spec(S: M1_STATE, u: USERS): M1_STATE =
  S WITH [ 'sc_in_spec := LAMBDA (w: USERS, v: MACH):
    IF w=u THEN S'sc_in_spec(w,v)+1 ELSE
      S'sc_in_spec(w,v) ENDF ];
```

The `WITH` construct leaves the record S unchanged except for the `sc_in_spec` component, which is replaced by the λ -expression. The machine $TH'_{\mathcal{H}}$ 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 (FORALL w1: S'sc_in_spec(ua1,w1)<s1k) THEN
    IF S'init_spec(ua1,ua1) OR S'stopped_spec(ua1) THEN
      (# ns:=incr_sc_in_spec(S,ua1),0:=m1o_garbage #)
      %% increment sc_in_spec, but do not send any output
    ELSE
      (# ns:=incr_sc_in_spec(S,ua1)
        WITH [ 'init_spec(ua1,ua1) := TRUE ],
        0 := 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, 0:=m1o_garbage #)
  ENDIF
ELSIF ua1>M AND m1i_rec_init?(p) THEN
  ...

```

In a similar way we have formalized the machines $\text{TH}_{\mathcal{H}}$, $\{M'_u \mid u \in \mathcal{H}\}$, and $\text{Sim}_{\mathcal{H}}$. The M machines M'_u in the left part of Figure 4 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 $\text{TH}_{\mathcal{H}}$ and $\{M'_u \mid u \in \mathcal{H}\}$ respectively $\text{TH}'_{\mathcal{H}}$ and $\text{Sim}_{\mathcal{H}}$ is straightforward by composition of the corresponding state transition functions: An input from \mathcal{H} is always first handled by a machine M'_u and $\text{TH}'_{\mathcal{H}}$, and then by $\text{TH}_{\mathcal{H}}$ and $\text{Sim}_{\mathcal{H}}$, 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. However, it should be noted once more that all the definitions and claims in this section have been formalized and verified in PVS.

5.2 Proving the Bisimulation

In order to prove Lemma 1, we prove the following predicates to be invariants of the collections \hat{M}_* and \hat{M}_{spec} when they obtain the same inputs.

$$- \text{stopped}^* = \text{stopped}^{\text{id}} = \text{stopped}^{\text{sim}} = \text{stopped}^{\text{spec}}.$$

Note that we compare whole arrays in this predicate, i.e., we make use of the higher-order capabilities of PVS. One could also write $\forall u : \text{stopped}_u^* = \text{stopped}_u^{\text{id}} = \dots$, but the equality of the whole arrays is more concise and easier to use in the proofs.

$$- \text{sc_in}^{\text{id}} = \text{sc_in}^{\text{spec}}.$$

$$- \text{init}^* = \text{init}^{\text{sim}} = \text{init}^{\text{spec}}.$$

$$- \text{msg_in}^{\text{id}} = \text{msg_in}^{\text{spec}}.$$

$$- \forall u, v \in \mathcal{H} : \text{length}(\text{deliver}_{u,v}^*) = \text{length}(\text{deliv_i}_{u,v}^*).$$

length is the PVS function delivering the length of lists. We use the quantified form of the invariant here instead of the higher-order form, since otherwise we would have to ‘lift’ the length function to arrays of lists.

$$- \forall u, v \in \mathcal{H} : \text{length}(\text{deliver}_{u,v}^{\text{spec}}) = \text{length}(\text{deliv_i}_{u,v}^{\text{spec}}).$$

$$- \text{deliver}^* = \text{deliver}^{\text{spec}} \text{ and } \text{deliv_i}^* = \text{deliv_i}^{\text{spec}}.$$

$$- \text{sc_out}^{\text{id}} = \text{sc_out}^{\text{spec}}.$$

$$- \forall w \in \mathcal{M}, u \in \mathcal{H} : \text{sc_out}_{w,u}^{\text{sim}} \leq \text{sc_out}_{w,u}^{\text{spec}}.$$

Again we use the quantified form, since otherwise we had to lift “ \leq ” to arrays.

$$- \forall w \in \mathcal{M}, u \in \mathcal{H} : ((w \in \mathcal{H} \implies \text{msg_out}_{w,u}^{\text{id}} = \text{msg_out}_{w,u}^{\text{spec}}) \text{ and}$$

$$(w \in \mathcal{A} \wedge \text{sc_out}_{w,u}^{\text{id}} < s_2(k) \implies \text{msg_out}_{w,u}^{\text{id}} = \text{msg_out}_{w,u}^{\text{sim}})).$$

Each of the 10 invariants is formalized as a predicate $\phi_i(S_{\text{si}}, S_{\text{sp}})$ on the current states of the two collections \hat{M}_* and \hat{M}_{spec} . The conjunction of all the ϕ_i yields the bisimulation relation ϕ . Let δ_{si} and δ_{sp} denote the overall transition function of the machine collections \hat{M}_* and \hat{M}_{spec} , respectively. The following theorem asserts that the invariants indeed are invariants of these collections:

Theorem 2. *Let S_{si} and S_{sp} be states of the two collections \hat{M}_* and \hat{M}_{spec} such that all invariants $\phi_i(S_{\text{si}}, S_{\text{sp}})$, $1 \leq i \leq 10$ hold. The transition functions $\delta_{\text{si}}, \delta_{\text{sp}}$ preserve the invariants, i.e., for an arbitrary overall input \mathcal{I} of \hat{M}_* and \hat{M}_{spec} we have*

$$\phi_i(S'_{\text{si}}, S'_{\text{sp}}) \forall i, 1 \leq i \leq 10$$

with $(S'_{\text{si}}, \mathcal{O}_{\text{si}}) := \delta_{\text{si}}(S_{\text{si}}, \mathcal{I})$ and $(S'_{\text{sp}}, \mathcal{O}_{\text{sp}}) := \delta_{\text{sp}}(S_{\text{sp}}, \mathcal{I})$. Furthermore, the initial states $\text{initial}_{\text{si}}$ and $\text{initial}_{\text{sp}}$ satisfy all 10 invariants. \square

In PVS, this theorem is split into 10 lemmas, one for each invariant. Using the invariants ϕ_i , we prove the following theorem:

Theorem 3. *Let S_{si} and S_{sp} be states satisfying all invariants $\phi_i(S_{\text{si}}, S_{\text{sp}})$, $1 \leq i \leq 10$, and let \mathcal{I} be an overall input of the collections \hat{M}_* and \hat{M}_{spec} . Then both collections make the same outputs on all ports to the users and the adversary. \square*

Together, Theorems 2 and 3 prove that the two systems are bisimilar, which finishes our proof of Theorem 1.

5.3 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.

6 Summary and Future Work

We have presented the first abstract specification for secure message transmission preventing message reordering, together with a secure implementation. Its proof of security involved a recently proven composition theorem [16] and a bisimulation which we formally verified using the theorem prover PVS. Our approach furthermore presents a general strategy how to derive real implementations by splitting specifications into smaller systems that can then be refined stepwise using the composition theorem and formal proof systems.

One next step is to verify the claimed integrity property of the systems, i.e., a formula that messages are output in correct order. This requires more theoretical work, e.g., we have to show that integrity properties are in fact preserved under simulatability also in the asynchronous case (this is not trivial even though the synchronous case was already shown in [15]). Also the PVS proof becomes more complicated than the one presented here. A preliminary version of that work can already be seen in [3]. Putting our current paper and those results together, we are confident that our underlying model is well suited for future analysis of larger protocols including real cryptographic primitives, since it supports commonly accepted machine-aided proofs (like the one of [14]) without losing its sound cryptographic semantics.

Concerning further future work, there are innumerable things to do. Obviously, the security of the system presented in this paper is still based on paper-and-pencil proofs such as the composition theorem, the transitivity lemma or the security proof in [16]. Hence, one future step could be the verification of those theorems using formal proof systems. However, we are aware of the difficulty of this task, mostly because of the occurrence of probabilism. In the shorter term, we are turning our attention to a library which should provide sound abstractions of a set of common cryptographic primitives. The library may naturally serve as a construction kit for designing large protocols whose security properties can then easily be validated again by formal proof systems.

References

1. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation* 148/1 (1999) 1-70.
2. M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *IFIP Intern. Conf. on Theoretical Computer Science (TCS 2000)*, LNCS 1872, Springer-Verlag, 2000, 3–22.
3. M. Backes. Cryptographically sound analysis of security protocols. Ph.D thesis, Computer Science Department, Saarland University, 2002.
4. D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory* 29/2 (1983) 198-208.
5. F. J. T. Fabrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Why is a security protocol correct? *1998 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, Los Alamitos 1998, 160-171.
6. C. A. R. Hoare. *Communicating sequential processes*. International Series in Computer Science, Prentice Hall, Hemel Hempstead 1985.
7. P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. *5th ACM Conference on Computer and Communications Security*, San Francisco, November 1998, 112–121.
8. P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. Probabilistic polynomial-time equivalence and security analysis. *Formal Methods '99*, LNCS 1708, Springer-Verlag, 1999, 776–793.
9. G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 1055, Springer-Verlag, Berlin 1996, 147-166.
10. N. Lynch. *Distributed algorithms*. Morgan Kaufmann Publishers, San Francisco 1996.
11. N. Lynch. I/O automaton models and proofs for shared-key communication systems. *12th Computer Security Foundations Workshop (CSFW)*, IEEE, 1999, 14–29.
12. S. Owre and N. Shankar. Abstract datatypes in PVS. Technical report, Computer Science Laboratory, SRI International, 1993.
13. S. Owre, N. Shankar, and J. M. Rushby. PVS: A prototype verification system. In *CADE 11*, volume 607 of *LNAI*, pages 748–752. Springer, 1992.
14. L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1):85-128, 1998.
15. B. Pfizmann and M. Waidner. Composition and integrity preservation of secure reactive systems. *7th ACM Conference on Computer and Communications Security*, Athens, November 2000, 245-254.
16. B. Pfizmann and M. Waidner. A model for asynchronous reactive systems and its application to secure message transmission. *IEEE Symposium on Security and Privacy*, Oakland, May 2001, 184-202.
17. S. Schneider. Security properties and CSP. *1996 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, Washington 1996, 174-187.
18. A. C. Yao. Protocols for secure computations. *23rd Symposium on Foundations of Computer Science (FOCS) 1982*, IEEE Computer Society, 1982, 160-164.