

# Unifying Simulatability Definitions in Cryptographic Systems under Different Timing Assumptions (Extended Abstract)\*

Michael Backes

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

**Abstract.** The cryptographic concept of simulatability has become a salient technique for faithfully analyzing and proving security properties of arbitrary cryptographic protocols. We investigate the relationship between simulatability in synchronous and asynchronous frameworks by means of the formal models of Pfitzmann et. al., which are seminal in using this concept in order to bridge the gap between the formal-methods and the cryptographic community. We show that the synchronous model can be seen as a special case of the asynchronous one with respect to simulatability, i.e., we present an embedding between both models that we show to preserve simulatability. We show that this result allows for carrying over lemmas and theorems that rely on simulatability from the asynchronous model to its synchronous counterpart without any additional work. Hence future work can concentrate on the more general asynchronous case, without having to neglect the analysis of synchronous protocols.

## 1 Introduction

In recent times, the analysis of cryptographic protocols has been getting more and more attention, and the demand for general frameworks for representing cryptographic protocols and the security requirements of cryptographic tasks has been rising. Existing frameworks are either motivated by the complexity-theoretic view on cryptography, which aims at proving cryptographic protocols with respect to the cryptographic semantics, or they are motivated by the view of the formal-methods community, which aims at capturing abstractions of cryptography in order to make such protocols accessible for formal verification. Frameworks built on abstractions will be further dealt with in the related literature along with a discussion on the cryptographic justification of these abstractions.

For living up to the probabilistic nature of cryptography, a framework for dealing with actual cryptography necessarily has to be able to deal with probabilistic behaviors. The standard understanding in well-known, non security-specific probabilistic frameworks like [31, 33] is that the order of events is fixed by means of a probabilistic scheduler that has full information about the system. In contrast to that, the standard understanding in cryptology (closest to a rigorous definition in [10]) is that the adversary schedules everything, but only with realistic information. This corresponds to making a certain subclass of schedulers explicit for the model from [31]. However, if one splits

---

\* The full version is available from <http://eprint.iacr.org/2003/114>.

a machine into local submachines, or defines intermediate systems for the purposes of proof only, this may introduce many schedules that do not correspond to a schedule of the original system and therefore just complicate the proofs. The typical solution is a distributed definition of scheduling which allows machines that have been scheduled to schedule certain (statically fixed) other machines themselves.

Based on these requirements, several general definitions of secure protocols were developed over the years, e.g. [15, 7, 28, 11, 30, 12], which are all potential candidates for such a framework. To allow for a faithful analysis of cryptographic protocols, it is well-known that such models not only have to capture probabilistic behaviors, but also complexity-theoretically bounded adversaries as well as a reactive environment of the protocol, i.e., continuous interaction with the users and the adversary. Unfortunately, most of the above work does not live up to these requirements in spite of its generality, mainly since it concentrates on the task of secure function evaluation, which does not capture a reactive environment. Currently, the models of Pfitzmann et. al. [28, 30] and Canetti [12], which have been developed concurrently but independently, stand out as the standard models for sound protocol analysis and design.

Regarding the underlying definition of time, such models can be split into synchronous and asynchronous ones. In synchronous models [28], time is assumed to be expressible in rounds, whereas asynchronous scenarios [30, 12] do not impose any assumption on time. This makes asynchronous scenarios attractive since no assumption is made about network delays and the relative execution speed of the parties. Moreover, the notion of rounds is difficult to justify in practice as it seems to be very difficult to establish them for the Internet for example. This attractiveness is substantiated by a large body of literature on asynchronous cryptographic protocols, e.g., [8, 14]. However, time guarantees are sometimes explicitly desired, e.g., on when a process can abort. Hence assumptions have to be made in this case, which induce a certain amount of synchrony again. This sometimes makes a synchronous assumption of time nevertheless necessary in practice, e.g., in Kerberos [23].

Hence researchers usually restrict their attention to one definition of time, or they are driving double-tracked by maintaining two separate models. However, this presupposes proving every theorem for both models. This is not nice. An alternative approach, taken in this work, is to show that the synchronous model can be regarded as a special case of an asynchronous one, and hence does not have to be considered separately, but still can be used to conveniently express synchronous protocols.

Although this idea might not be surprising, it is very difficult to achieve since it turns out that carrying over results from the asynchronous to the synchronous model presupposes the ability of (at least partially) reversing the considered embedding. Recall that suitable frameworks, especially the frameworks of Canetti and Pfitzmann et. al., have a distributed scheduling which significantly complicates this reversion.

Formally, a special case means that there is an embedding into the asynchronous model that preserves a desired property. Which property has to be preserved depends on the goals to strive for. For cryptographic protocols, the property of *simulatability* stands out. Simulatability captures the notion of a cryptographically secure implementation and serves as a link to the formal-methods community, which typically only hold a top-level view of cryptography, where cryptographic primitives are replaced by deter-

ministic abstractions. A more comprehensive discussion of simulatability and its relationship to protocol verification work done by the formal-methods community is given in the paragraph on related literature below.

In the following, we investigate the synchronous and asynchronous models of Pfitzmann et. al. [28, 30], which are seminal in using the concept of simulatability to bridge the gap between the formal-methods and the cryptographic community. We show that the synchronous model can be embedded in the asynchronous model such that simulatability is preserved by this embedding, i.e., if two systems fulfill the simulatability relation in the synchronous model, their respective images fulfill the relation in the asynchronous model and vice versa. We show that this result allows for carrying over lemmas and theorems from the asynchronous case to the synchronous case without proving them twice. We are confident that this result helps to make future protocol analysis in these models more convenient and more efficient.

Moreover, we believe that our approach for establishing the embedding and its properties can be successfully used for other models with only minor changes. Especially the asynchronous model of Canetti is surely worth to be investigated. However, his corresponding synchronous model [11] is still specific for secure function evaluation; hence adopting it to a reactive environment is a necessary prerequisite for this future work. The lack of such a reactive synchronous model was – besides the fact that the models of Pfitzmann et. al. are more rigorously defined than the one of Canetti – our main reason why we decided to base our work on the model of Pfitzmann et. al.

*Related Literature.* If cryptographic protocols should be verified using formal methods, some kind of abstraction is needed as the underlying reduction proofs of cryptography are still out of scope of current verification techniques. This abstraction is usually based on the so-called Dolev-Yao model [13], which considers cryptographic primitives as operators in a free algebra where only predefined cancellation rules hold. This abstraction simplifies proofs of larger protocols considerably, and it gave rise to a large body of literature on analyzing the security of protocols using techniques for formal verification of computer programs (a very partial list of work includes [22, 9, 20, 26, 1]).

Since this line of work turned out to be very successful, the interesting question arose whether these abstractions are indeed justified from the view of cryptography, i.e., whether properties proved for the abstractions are still valid for the cryptographic implementation. Abadi et. al. showed in [3, 2] that the Dolev-Yao model is cryptographically faithful at least for symmetric encryption and synchronous protocols. There, however, the adversary is restricted to passive eavesdropping. Consequently, it was not necessary to choose a reactive model of a system and its honest users, and the notion of simulatability could be replaced by the weaker notion of indistinguishability [34]. Guttman et. al. showed in [17] that the probability of two executions of the same protocol – either executed in a Dolev-Yao-like framework or using real cryptographic primitives – may deviate from each other at most for a certain bound. However, their results are specific for the Wegman-Carter system so far. Moreover, as this system is information-theoretically secure, its security proof is much easier to handle than primitives with security guarantees only against computationally bounded adversaries since no reduction proofs against underlying number-theoretic assumptions have to be made. Some further approaches for special security goals or primitives are [32, 19]. However, there

is evidence that the original Dolev-Yao model is not justified in the presence of active attacks, even if provably secure cryptographic primitives are used, cf. [27] for an (admittedly constructed) counterexample. This exemplifies the demand for “better” abstractions which the models of Canetti and of Pfitzmann et. al. want to establish using the concept of simulatability.

Simulatability bridges this gap by serving as a cryptographically sufficient relationship between abstract specifications and cryptographic implementations, i.e., abstractions which can be shown to simulate a given implementation in a particular sense are known to be sound with respect to the security definitions of cryptography. Simulatability was first invented for multi-party function evaluation [15, 7, 11], i.e., systems with only one initial input set and only one output set. An extension to a reactive scenario, where participants can make new inputs many times, e.g., start new sessions like key exchanges, was first fully defined in [27], with extensions to asynchronous systems in [30, 12]. Each of the three considered models has already been successfully used to built up sound abstractions of various cryptographic primitives and all of them enjoy a composition theorem, i.e., large protocols can be refined step-wise without destroying the already proven properties.

Comparing the models of Canetti and Pfitzmann et. al., we can state that Canetti’s work enjoys a more general composition theorem and has moreover addressed more cryptographic primitives so far. On the other hand, the models of Pfitzmann et. al. are more rigorously defined and early examples of tool-supported proofs in their models exist [5, 4], using PVS [25]. Moreover, the recently published universally composable cryptographic library [6] may pave the way to formal verification of large security protocols within their models.

## 2 Review of the Reactive Models in Synchronous and Asynchronous Networks

In this section we review the synchronous and the asynchronous model for probabilistic reactive systems as introduced in [28] and [30], respectively. Several definitions are only sketched, whereas those that are essential for understanding our upcoming results are given in full detail.

### 2.1 General System Model

In the following we consider a finite alphabet  $\Sigma$  and some special symbols  $!, ?, \leftarrow, \triangleleft \notin \Sigma$  that will be used to express different ports of machines. For  $s \in \Sigma^*$  and  $l \in \mathbb{N}_0$ , we define  $s \upharpoonright_l$  to be the  $l$ -letter prefix of  $s$ .

Our machine model is *probabilistic state-transition machines*, similar to probabilistic I/O automata as sketched by Lynch [21]. Communication between different machines is done via *ports* which are divided into *input* and *output* ports. Inspired by the CSP-Notation [18] we write input and output ports as  $q?$  and  $q!$ . Let  $q!^C := q?$  and vice versa; for a set  $P$  of input and output ports, let  $P^C := \{q \mid q^C \in P\}$ .

Ports will later be connected by naming convention, i.e., a port  $q!$  always sends messages to  $q?$ . In the asynchronous model, a special machine called a *buffer* will further be inserted in each connection to ensure asynchronous behavior. A buffer stores all of

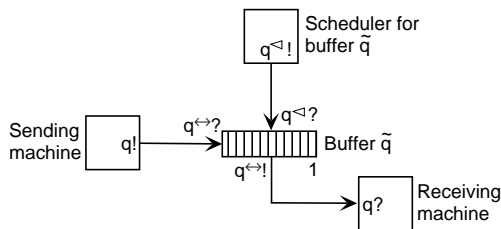
its inputs in an internal list. If a machine wants to schedule the  $i$ -th message of buffer  $\tilde{q}$  (this machine must have the unique clock-out port  $q^{\leftarrow!}$ ), it simply sends  $i$  at  $q^{\leftarrow!}$ , cf. Figure 1. The buffer then outputs the  $i$ -th message and removes it from its internal list. Neither buffers nor clock ports occur in the synchronous model; they are just included to establish a distributed scheduling in the asynchronous case.

A *machine*  $M$  has a *name*  $name_M$ , a sequence  $Ports_M$  of *ports*, containing both input ports and output ports, and a set  $States_M$  of *states*, comprising sets  $Ini_M$  and  $Fin_M$  of *initial* and *final states*, respectively. If a machine is switched, it receives an input tuple at its input ports and performs its *transition function*  $\delta_M$  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. Furthermore, each machine has a bound  $l_M$  on the length of the considered inputs which allows time bounds on the computation time independent of the environment. The parts of an input that are beyond the length bound are ignored, i.e., incoming strings are only processed up to a predefined length. In particular, this is used to ensure polynomial runtime of individual machines.

For a set  $\hat{M}$  of machines, let  $ports(\hat{M})$  denote the set of ports of all machines  $M \in \hat{M}$ . Machines usually start with one initial input, i.e., the starting state is parameterized. Complexity is measured in terms of the length of this initial input, usually a security parameter  $k$  given in unary representation; in particular, polynomial-time is meant in this sense. We only briefly state here, that these machines have a natural realization as a probabilistic interactive Turing machine as introduced in [16]. We call a machine  $M$  a *black-box submachine* of a machine  $M'$  if the machine  $M'$  has access to the state-transition function  $\delta_M$  of  $M$ , i.e., it can execute  $\delta_M$  for the current state of the machine and arbitrary inputs.

A *collection*  $\mathcal{C}$  of machines is a finite set of machines with pairwise different machine names and disjoint sets of ports. In the asynchronous model, the *completion*  $[\mathcal{C}]$  of a collection  $\mathcal{C}$  is the union of all machines of  $\mathcal{C}$  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 collection  $\mathcal{C}$  are denoted as  $free(\mathcal{C})$ . In the asynchronous model, a collection  $\mathcal{C}$  is called *closed* if its completion  $[\mathcal{C}]$  has no free ports except a special master clock-in port  $clk^{\leftarrow?}$ , i.e.,  $free([\mathcal{C}]) = \{clk^{\leftarrow?}\}$ . When we define the interaction of several machines, this port will be used to resolve situations where the interaction cannot proceed. In the synchronous case, we demand  $free(\mathcal{C}) = \emptyset$  instead.

For security purposes, special collections are needed, because an adversary may have taken over parts of the initially intended system, e.g., different situations have to be captured depending on which and how many users are considered as being malicious. Therefore, a system consists of several possible remaining structures.



**Fig. 1.** Ports and buffers.

**Definition 1.** (*Structures and Systems*) A structure is a pair  $struc = (\hat{M}, S)$  where  $\hat{M}$  is a collection of non-buffer machines called correct machines, and  $S \subseteq \text{free}(\hat{M})$  is called specified ports. If  $\hat{M}$  is clear from the context, let  $\bar{S} := \text{free}(\hat{M}) \setminus S$ . We call  $\text{forb}(\hat{M}, S) := \text{ports}(\hat{M}) \cup \bar{S}^C$  the forbidden ports, i.e., those ports that the honest user is forbidden to have. 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 definition. The specified ports are those where a certain service is guaranteed. Note that this definition is valid for both the synchronous and the asynchronous case. In particular, buffers do not have to be explicitly included in the specification of a system, e.g., in the specification of a cryptographic protocol that one wants to analyze. The different timing assumption stem from the different definitions of runs which we will introduce below.

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.** (*Configurations*) A configuration of a system  $Sys$  is a tuple  $conf = (\hat{M}, S, H, A)$  where  $(\hat{M}, S) \in Sys$  is a structure,  $\hat{M} \cup \{H, A\}$  is a closed collection, and  $\text{ports}(H) \cap \text{forb}(\hat{M}, S) = \emptyset$ . The set of configurations is written  $\text{Conf}(Sys)$ . The set of configurations with polynomial-time user  $H$  and adversary  $A$  is called  $\text{Conf}_{\text{poly}}(Sys)$ . The index  $_{\text{poly}}$  is omitted if it is clear from the context. The initial states of all machines in a configuration are a common security parameter  $k$  in unary representation.  $\diamond$

## 2.2 Capturing Asynchronous Runs

For a configuration, both models define a probability space of runs (sometimes called *traces* or *executions*). In the asynchronous model, 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 at the beginning of Section 2.1. If this 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. This is formally captured as follows.

**Definition 3.** (*Asynchronous Runs and Views*) For a given configuration  $conf = (\hat{M}, S, H, A)$  with master scheduler  $X \in \hat{M} \cup \{A\}$ , set  $\hat{C} := [\hat{M} \cup \{H, A\}]$ . 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  $M_{CS} := X$ , and treats each port as a variable over  $\Sigma^*$ , initialized with  $\epsilon$  except for  $\text{clk}^{s?} := 1$ . Probabilistic choices only occur in Phase (1).

1. Switch current scheduler: Switch machine  $M_{CS}$ , i.e., set  $(s', O) \leftarrow \delta_{M_{CS}}(s, I)$  for its current state  $s$  and input port values  $I$ . Then assign  $\epsilon$  to all input ports of  $M_{CS}$ .
2. Termination: If  $X$  is in a final state, the run stops.

3. Buffer messages: For each simple output port  $q!$  of  $M_{CS}$  in the given order, switch buffer  $\tilde{q}$  with input  $q^{\leftrightarrow?} := q!$ , cf. Figure 1. Then assign  $\epsilon$  to all these ports  $q!$  and  $q^{\leftrightarrow?}$ .
4. Clean up scheduling: If at least one clock-out port of  $M_{CS}$  has a value  $\neq \epsilon$ , let  $q^{\triangleleft!}$  denote the first such port and assign  $\epsilon$  to the others. Otherwise let  $\text{clk}^{\triangleleft?} := 1$  and  $M_{CS} := X$  and go back to Phase (1).
5. Scheduled message: Switch  $\tilde{q}$  with input  $q^{\triangleleft?} := q^{\triangleleft!}$  (cf. Figure 1), set  $q^? := q^{\leftrightarrow!}$  and then assign  $\epsilon$  to all ports of  $\tilde{q}$  and to  $q^{\triangleleft!}$ . Let  $M_{CS} := M'$  for the unique machine  $M'$  with  $q^? \in \text{ports}(M')$ . Go back to Phase (1).

Whenever a machine (this may be a buffer) with name  $\text{name}_M$  is switched from  $(s, I)$  to  $(s', O)$ , we add a step  $(\text{name}_M, s, I', s', O)$  with  $I' := I \upharpoonright_{l_M}$  to the run  $r$ , except if  $s$  is final or  $I' = (\epsilon, \dots, \epsilon)$ . For each value of the security parameter, this gives a random variable denoted as  $\text{run}_{\text{conf}, k}$ , hence we obtain a family of random variables

$$\text{run}_{\text{conf}} = (\text{run}_{\text{conf}, k})_{k \in \mathbb{N}}.$$

The view of a subset  $M \subset \hat{C}$  in a run  $r$  is the restriction of  $r$  to  $M$ , i.e., the subsequence of all steps  $(\text{name}_M, s, I, s', O)$ , where  $\text{name}_M$  is the name of a machine  $M \in M$ . This gives a family of random variables

$$\text{view}_{\text{conf}}(M) = (\text{view}_{\text{conf}, k}(M))_{k \in \mathbb{N}}.$$

For a singleton  $M = \{H\}$  we write  $\text{view}_{\text{conf}}(H)$  instead of  $\text{view}_{\text{conf}}(\{H\})$ .  $\diamond$

This rather informal definition of runs can naturally be formalized using transition probabilities, which induce probability spaces over the finite sequences of steps similar to Markov Chains. The extension to infinite sequences can then be achieved using well-established results of measure theory and probability theory, cf. Section 5 of [24]. It is further easy to show that views of polynomial-time machines are of polynomial size.

### 2.3 Capturing Synchronous Runs

In the synchronous model, ports, machines, collections, structures, and systems are defined similar to the asynchronous model. The only exception is that there are no clock ports and no buffers, which have only been included to model asynchronous timing, i.e., corresponding ports  $p^?$  and  $p!$  are directly connected. The main difference is the definition of runs. Instead of our asynchronous run algorithm (cf. Definition 3), runs are defined using *rounds* which is the usual concept in synchronous scenarios. Every global round is again divided into  $n$  so-called subrounds, and there is a mapping  $\kappa$ , called *clocking scheme*, from the set  $\{1, \dots, n\}$  into the powerset of considered machines, i.e., the machines of the structure, the user, and the adversary. Here  $\kappa(i)$  denotes the machines that switch in subround  $i$ . We call a clocking scheme *valid* if every machine is clocked at most once between two successive clockings of the adversary.

**Definition 4.** (*Synchronous Runs and Views*) Given a configuration  $\text{conf} = (\hat{M}, S, H, A)$  along with a clocking scheme  $\kappa$  for  $n \in \mathbb{N}$ , runs are defined as follows: Each global round  $i$  has  $n$  subrounds. In subround  $[i.j]$  all machines  $M \in \kappa(j)$  switch simultaneously, i.e., each state-transition function  $\delta_M$  is applied to  $M$ 's current input yielding

a new state and output (probabilistically). The output at a port  $p!$  is available as input at  $p?$  until the machine with port  $p?$  is switched. If several inputs arrive until that time, they are concatenated. This gives a family of random variables

$$run_{conf} = (run_{conf,k})_{k \in \mathbb{N}}.$$

More precisely, each run is a function mapping each triple  $(M, i, j) \in \hat{M} \cup \{H, A\} \times \mathbb{N} \times \{1, \dots, n\}$  to a quadruple  $(s, I', s', O)$  of the old and new state, inputs (with  $I' := I \upharpoonright_{l_M}$  again), and outputs of machine  $M$  in subround  $[i..j]$ , with  $I' \equiv \epsilon$ ,  $O \equiv \epsilon$ , and  $s = s'$  if  $M$  is not switched in this subround. The view of  $M \subset \hat{M} \cup \{H, A\}$  in a run  $r$  is the restriction of  $r$  to  $M \times \mathbb{N} \times \{1, \dots, n\}$ . This gives a family of random variables

$$view_{conf}(M) = (view_{conf,k}(M))_{k \in \mathbb{N}}.$$

◇

Again, the view of a polynomial-time machine can easily be shown to be of polynomial size. Alternatively, we can consider runs as a sequence of seven-tuples  $(M, i, j, s, I', s', O)$  for ascending values of  $i$  and  $j$ , where tuples with the same values  $i$  and  $j$  can be ordered arbitrarily since they switch simultaneously and do not influence each other. This characterization of runs is equivalent to the original one (we just expanded the function), but it is better suited for our upcoming proofs.

Instead of arbitrary clocking schemes as in the above definition of runs, the authors of [28] focus on only one special clocking scheme  $\kappa$ , given by  $(\hat{M} \cup \{H\}, \{A\}, \{H\}, \{A\})$ . Clocking the adversary between the correct machines is the well-known model of “rushing adversaries”. In [28], it has been shown that this clocking scheme does not restrict the possibilities of the adversary. Since our upcoming results hold for all valid clocking schemes, they in particular hold for rushing adversaries.

## 2.4 Simulatability

The definition of one system securely implementing another one is based on the common concept of *simulatability*. Simulatability essentially means that whatever might happen to an honest user in a real system  $Sys_1$  can also happen in an ideal system  $Sys_2$ : For every structure  $struc_1 \in Sys_1$ , every user  $H$ , and every adversary  $A_1$ , there exists an adversary  $A_2$  on a corresponding ideal structure  $struc_2$  such that the view of  $H$  is indistinguishable in the two configurations. Indistinguishability is a well-defined cryptographic notion from [34].

**Definition 5.** (*Computational Indistinguishability*) Two families  $(var_k)_{k \in \mathbb{N}}$  and  $(var'_k)_{k \in \mathbb{N}}$  of random variables (or probability distributions) on common domains  $D_k$  are computationally indistinguishable (“ $\approx$ ”) if for every algorithm  $Dis$  (the distinguisher) that is probabilistic polynomial-time in its first input,

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

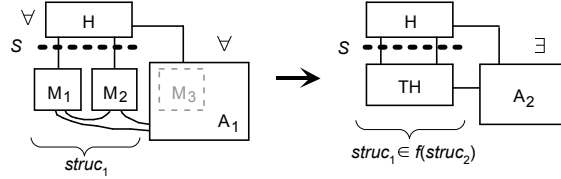
Intuitively, given the security parameter and an element chosen according to either  $var_k$  or  $var'_k$ ,  $Dis$  tries to guess which distribution the element came from. ◇

<sup>1</sup> The class *NEGL* denotes the set of all negligible functions, i.e.,  $g: \mathbb{N} \rightarrow \mathbb{R}_{\geq 0} \in NEGL$  if for all positive polynomials  $Q$ ,  $\exists k_0 \forall k \geq k_0: g(k) \leq 1/Q(k)$ .

Corresponding structures in the simulatability definition are designated by a function  $f$  from  $Sys_1$  to the powerset of  $Sys_2$ . The function  $f$  is called *valid* if it maps structures with the same set of specified ports.

**Definition 6. (Simulatability)** *Let systems  $Sys_1$  and  $Sys_2$  with a valid mapping  $f$  be given. We say  $Sys_1 \geq^f Sys_2$  if for every configuration  $conf_1 = (\hat{M}_1, S, H, A_1) \in \text{Conf}_{\text{poly}}(Sys_1)$ , there exists a configuration  $conf_2 = (\hat{M}_2, S, H, A_2) \in \text{Conf}_{\text{poly}}(Sys_2)$  with  $(\hat{M}_2, S) \in f(\hat{M}_1, S)$  such that  $\text{view}_{conf_1}(H) \approx \text{view}_{conf_2}(H)$ .  $\diamond$*

This is shown in Figure 2. In the following, we augment  $\geq$  with a subscript  $\text{sync}$  or  $\text{async}$  to distinguish the definition of the synchronous and asynchronous case. In a typical ideal system, each structure contains only one machine TH called trusted host, which serves as an ideal functionality of the real system. The machine TH is usually deterministic with a very simple transition function and hence in scope of current verification techniques.



**Fig. 2.** Overview of the simulatability definition.

### 3 Idea and Definition of the Embedding

The informal idea of the embedding  $\varphi_{Sys}$  is to add an explicit master scheduler that should simulate the synchronous run induced by the given clocking scheme. However, due to the general distributed scheduling (cf. Definition 3), leaving the actual machines unmodified leads to non-simulatable situations, as these machines can clock themselves without ever giving control to this explicit master scheduler.

Hence, we first define a mapping  $\varphi_M$  that surrounds single synchronous machines (i.e., machines that are designed for a synchronous environment) with an “asynchronous coat”. More precisely, if a synchronous machine  $M_{\text{sync}}$  makes a transition, it obtains all inputs at once that arrived since its last scheduling, whereas in asynchronous scenarios, these inputs come one by one and have to be processed in several transitions. Thus, the surrounding asynchronous machine  $\varphi_M(M_{\text{sync}})$  stores all inputs internally, until it is asked to perform the transition of its synchronous submachine. This is modeled by a special port  $p_{M_{\text{sync}}}$  of the machine  $\varphi_M(M_{\text{sync}})$ . An input at this port causes  $\varphi_M(M_{\text{sync}})$  to schedule its submachine with the collected inputs and forward its outputs. As such embedded machines do not produce any clock outputs, the ports  $p_{M_{\text{sync}}}$  can be used by the master scheduler to simulate the synchronous time by a suitable scheduling strategy.

The formal definition of  $\varphi_M$  can be found in the full version. We only briefly state that  $\varphi_M(M_{\text{sync}})$  is polynomial-time by construction iff  $M_{\text{sync}}$  is polynomial-time. For a set  $\hat{M}$  of synchronous machines, we further define  $\varphi_M(\hat{M}) := \bigcup_{M_{\text{sync}} \in \hat{M}} \varphi_M(M_{\text{sync}})$ .

The desired mapping  $\varphi_{Sys}$  on synchronous systems then simply embeds every machine and adds a specific master scheduler to each structure, i.e., for an arbitrary structure  $(M_{\text{sync}}, S_{\text{sync}})$  and clocking scheme  $\kappa$ , we obtain a structure  $(\varphi_M(\hat{M}_{\text{sync}}) \cup \{X_{\text{sync}, \kappa}\}, S_{\text{sync}})$ , where  $X_{\text{sync}, \kappa}$  is an explicit master scheduler defined as follows. Besides the master clock-in port, it has clock-out ports for clocking inputs and outputs

of the given structure, for clocking the connection between A and H, and finally ports  $p_{M_{\text{sync}}}!$ ,  $p_{M_{\text{sync}}}^{\leftarrow}$  for clocking, i.e., giving control to, each machine  $\varphi_M(M_{\text{sync}})$ . Internally, it maintains a variable  $local\_rnd$  over  $\{1, \dots, n\}$  and a variable  $global\_rnd$  over  $\mathbb{N}$  both initialized with 1. For the sake of readability, we describe the behavior of  $X_{\text{sync},\kappa}$  using “for”-loops. This is just a notational convention that should be understood as follows: every time  $X_{\text{sync},\kappa}$  is scheduled, it performs the next step of the loop.

1. **Schedule Current Machines:** For all machines  $M_{\text{sync}} \in \kappa(local\_rnd)$  output  $(global\_rnd, local\_rnd)$  at  $p_{M_{\text{sync}}}!$ , 1 at  $p_{M_{\text{sync}}}^{\leftarrow}$ . The order of the switched machines can be chosen arbitrary.
2. **Schedule Outgoing Buffers:** For all  $M_{\text{sync}} \in \kappa(local\_rnd)$  output 1 at every port  $p^{\leftarrow}$  with  $p \in Ports_{M_{\text{sync}}}$ . The order of the switched machine can be chosen arbitrary except that output ports of the adversary are scheduled first if  $A \in \kappa(local\_rnd)$ .<sup>2</sup>
3. **Switch to next Round:** Set  $local\_rnd := local\_rnd + 1$ . If  $local\_rnd > n$ , set  $global\_rnd := global\_rnd + 1$  and  $local\_rnd := 1$ . Go to Phase (1).

We finally define a mapping  $\varphi_{conf}$  on synchronous configurations of a system  $Sys$ , i.e., configurations which consist of synchronous machines only, by

$$\varphi_{conf}(\hat{M}_{\text{sync}}, S_{\text{sync}}, H, A) := (\varphi_M(\hat{M}_{\text{sync}}) \cup \{X_{\text{sync},\kappa}\}, S_{\text{sync}}, \varphi_M(H), \varphi_M(A)),$$

with  $X_{\text{sync},\kappa}$  given as in  $\varphi_{Sys}$  for the particular structure. We will in the following simply write  $S$  instead of  $S_{\text{sync}}$ ; we also write  $\varphi$  instead of  $\varphi_{Sys}$ ,  $\varphi_M$ , and  $\varphi_{conf}$  if its meaning is clear from the context.

## 4 The Embedding Theorems

We now have to prove that the function  $\varphi$  has the desired properties with respect to simulatability, i.e., we have

$$\varphi_{Sys}(Sys_{\text{sync},1}) \geq_{\text{async}} \varphi_{Sys}(Sys_{\text{sync},2}) \Rightarrow Sys_{\text{sync},1} \geq_{\text{sync}} Sys_{\text{sync},2},$$

where we omitted the mappings, i.e., the superscript of both relations for the sake of readability. This captures the content of our first embedding theorem. Unfortunately, the converse direction does not hold, but our second embedding theorem will state a weaker version that is still sufficient for our purpose.

We first take a look at the runs in a synchronous system  $Sys_{\text{sync}}$  and in its asynchronous counterpart  $\varphi(Sys_{\text{sync}})$ . We define a mapping  $\phi$  on the runs of the asynchronous system yielding runs of the synchronous system. Intuitively,  $\phi$  “compresses” an asynchronous run to its synchronous counterpart, which consists of much less steps.

### 4.1 Relating Asynchronous Runs and Synchronous Runs

In the following, let an arbitrary synchronous system  $Sys_{\text{sync}}$  with a clocking scheme  $\kappa$  and an arbitrary configuration  $conf_{\text{sync}} = (\hat{M}_{\text{sync}}, S, H_{\text{sync}}, A_{\text{sync}}) \in \text{Conf}(Sys_{\text{sync}})$

<sup>2</sup> Without this restriction, the behavior of the adversary could depend on outputs of machines scheduled in the same subround, which would lead to non-simulatable situations.

be given. Moreover, let  $conf_{async} = (\varphi(\hat{M}_{sync}) \cup \{X_{sync,\kappa}\}, S, \varphi(H_{sync}), A') \in \text{Conf}(\varphi(Sys_{sync}))$  be given (i.e.,  $\varphi(conf_{sync})$  but with an arbitrary adversary). We call such a configuration an *embedded synchronous configuration with arbitrary adversary*.

Note that runs of  $conf_{async}$  always have a prescribed structure induced by the behavior of the master scheduler  $X_{sync,\kappa}$ : they are built by “blocks”. The steps  $(M_{sync}, i, j, s, \mathcal{I}, s', \mathcal{O})$  of the machines  $M_{sync} \in \hat{M}_{sync} \cup \{H_{sync}\}$  switched in round  $[i..j]$  in the synchronous run are represented by two blocks in the asynchronous run. The first block corresponds to Phase (1) in the definition of  $X_{sync,\kappa}$  and hence consists of those steps induced by clocking the machines  $\varphi(M_{sync})$  with  $M_{sync} \in \kappa(j)$  and  $A'$  if  $A_{sync} \in \kappa(j)$ . More precisely, it consists of  $|\kappa(j)|$  sub-blocks, one for each switched machine, where a sub-block for  $M_{sync}$  comprises the step of the master scheduler, the step of the scheduled buffer, the step of the switched machine, and steps for the receiving buffers.<sup>3</sup> The second block corresponds to Phase (2) in the definition of  $X_{sync,\kappa}$  and hence consists of the steps of the buffers connected to the output ports of the switched machines as well as steps of the machines receiving the scheduled message.

Informally, the mapping  $\phi$  combines the blocks of all machines  $M_{sync} \in \kappa(j)$  yielding the synchronous step of every machine  $M_{sync}$  that switches in the  $j$ -th subround of the particular global round. Note that all necessary information (e.g.,  $M_{sync}, i, j, s, s'$  etc.) is already given by the blocks except for the gathered inputs  $I$ . The mapping  $\phi$  overcomes this absence by collecting all “partial” inputs itself. It now also becomes clear why we defined the master scheduler to schedule each machine specifically with a tuple  $(i, j)$  indicating the current global and local round, since this information would otherwise not be contained in the asynchronous run. The precise definition of the mapping  $\phi$  is omitted due to lack of space. We only state that  $\phi$  is also well-defined on the view of arbitrary subsets of machines. The following lemma establishes a computational bound on the mapping  $\phi$  in polynomial-time configurations:

**Lemma 1.** *Let  $conf_{async}$  be an embedded synchronous configuration with arbitrary adversary. If  $conf_{async}$  is polynomial-time, then  $\phi$  applied to the view of the honest user and the adversary is computable in polynomial-time.*  $\square$

We can moreover show that the mapping  $\phi$  compresses the view of any machine in an embedded synchronous configuration to the view of the original machine in the synchronous configuration again. Formally, this is captured in the following theorem.

**Theorem 1.** *Let a synchronous system  $Sys_{sync}$ , a clocking scheme  $\kappa$ , and a configuration  $conf_{sync} = (\hat{M}_{sync}, S, H_{sync}, A_{sync}) \in \text{Conf}(Sys_{sync})$  be given, and set  $conf_{async} := \varphi(conf_{sync})$ . Then for every  $M_{sync} \in \hat{M}_{sync} \cup \{H_{sync}, A_{sync}\}$ , we have*

$$view_{conf_{sync}}(M_{sync}) = \phi(view_{conf_{async}}(\varphi(M_{sync}))).$$

*Moreover,  $conf_{async}$  is polynomial-time iff  $conf_{sync}$  is polynomial-time.*  $\square$

The following theorem finally states that the runs obtained by applying the mapping  $\phi$  to an embedded synchronous configuration with arbitrary adversary are equal to the runs in the synchronous system for a suitable synchronous adversary.

<sup>3</sup> For  $A'$ , we might have additionally steps because of clocked self-loops.

**Theorem 2.** Let a synchronous system  $Sys_{\text{sync}}$ , a valid clocking scheme  $\kappa$ , and a configuration  $conf_{\text{async}} = (\varphi(\hat{M}_{\text{sync}}) \cup \{X_{\text{sync},\kappa}\}, S, \varphi(H_{\text{sync}}), A') \in \text{Conf}(\varphi(Sys_{\text{sync}}))$  be given. Then there exists an adversary  $A_{\text{sync}}$  using  $A'$  as a blackbox such that for  $conf_{\text{sync}} := (\hat{M}_{\text{sync}}, S, H_{\text{sync}}, A_{\text{sync}})$  and every  $M_{\text{sync}} \in (\hat{M}_{\text{sync}} \cup \{H_{\text{sync}}\})$ , we have

$$view_{conf_{\text{sync}}}(M_{\text{sync}}) = \phi(view_{conf_{\text{async}}}(\varphi(M_{\text{sync}})))$$

Moreover,  $conf_{\text{async}}$  is polynomial-time iff  $conf_{\text{sync}}$  is polynomial-time.  $\square$

## 4.2 The Actual Embedding Theorems

This section contains our two main theorems. We start with a well-known lemma capturing some basic properties of indistinguishable random variables.

**Lemma 2 (Indistinguishability).** *Indistinguishability of two families of random variables implies indistinguishability of any function  $\delta$  of them if  $\delta$  is polynomial-time computable. Moreover, identically distributed variables are indistinguishable and indistinguishability is an equivalence relation.*  $\square$

**Theorem 3. (First Embedding Theorem)** Let two arbitrary synchronous systems  $Sys_{\text{sync},1}$  and  $Sys_{\text{sync},2}$  with clocking schemes  $\kappa_1$  and  $\kappa_2$  be given such that  $\kappa_2$  is valid, and let  $\varphi(Sys_{\text{sync},1}) \geq_{\text{async}}^f \varphi(Sys_{\text{sync},2})$  for a valid mapping  $f$ . Then

$$Sys_{\text{sync},1} \geq_{\text{sync}}^{f'} Sys_{\text{sync},2},$$

where  $f'$  is defined as  $(\hat{M}_2, S_2) \in f'(\hat{M}_1, S_1) :\Leftrightarrow \varphi(\hat{M}_2, S_2) \in f(\varphi(\hat{M}_1, S_1))$ .  $\square$

*Proof.* Let an arbitrary configuration  $conf_{\text{sync},1} = (\hat{M}_{\text{sync},1}, S, H_{\text{sync}}, A_{\text{sync},1}) \in \text{Conf}_{\text{poly}}(Sys_{\text{sync},1})$  be given. We first apply  $\varphi_{conf}$  on  $conf_{\text{sync},1}$  yielding a configuration  $conf_{\text{async},1} = (\varphi(\hat{M}_{\text{sync},1}) \cup \{X_{\text{sync},1,\kappa_1}\}, S, \varphi(H_{\text{sync}}), \varphi(A_{\text{sync},1})) \in \text{Conf}(\varphi(Sys_{\text{sync},1}))$ . According to Theorem 1,  $conf_{\text{async},1}$  is also polynomial-time and we have

$$view_{conf_{\text{sync},1}}(H_{\text{sync}}) = \phi(view_{conf_{\text{async},1}}(\varphi(H_{\text{sync}}))). \quad (1)$$

Thus, the precondition  $\varphi(Sys_{\text{sync},1}) \geq_{\text{async}}^f \varphi(Sys_{\text{sync},2})$  can be applied yielding a configuration  $conf_{\text{async},2} = (\varphi(\hat{M}_{\text{sync},2}) \cup \{X_{\text{sync},2,\kappa_2}\}, S, \varphi(H_{\text{sync}}), A_{\text{sync},2}) \in \text{Conf}_{\text{poly}}(\varphi(Sys_{\text{sync},2}))$  with  $view_{conf_{\text{async},1}}(\varphi(H_{\text{sync}})) \approx view_{conf_{\text{async},2}}(\varphi(H_{\text{sync}}))$  and  $\varphi(\hat{M}_{\text{sync},2}, S) \in f(\varphi(\hat{M}_{\text{sync},1}, S))$ . As both  $conf_{\text{async},1}$  and  $conf_{\text{async},2}$  are polynomial-time, Lemma 1 and Lemma 2 together yield

$$\phi(view_{conf_{\text{async},1}}(\varphi(H_{\text{sync}}))) \approx \phi(view_{conf_{\text{async},2}}(\varphi(H_{\text{sync}}))). \quad (2)$$

We now apply Theorem 2 to the configuration  $conf_{\text{async},2}$ , which yields a configuration  $conf_{\text{sync},2} = (\hat{M}_{\text{sync}}, S, H_{\text{sync}}, A_{\text{sync},2}) \in \text{Conf}_{\text{poly}}(Sys_{\text{sync},2})$  with

$$\phi(view_{conf_{\text{async},2}}(\varphi(H_{\text{sync}}))) = view_{conf_{\text{sync},2}}(H_{\text{sync}}). \quad (3)$$

Now Equation 1, 2, and 3 together with Lemma 2 finally yield  $view_{conf_{\text{sync},1}}(H_{\text{sync}}) \approx view_{conf_{\text{sync},2}}(H_{\text{sync}})$ . Moreover, we have  $\varphi(\hat{M}_{\text{sync},2}, S) \in f(\varphi(\hat{M}_{\text{sync},1}, S))$ , i.e.,  $(\hat{M}_{\text{sync},2}, S) \in f'(\hat{M}_{\text{sync},1}, S)$  which finally yields  $Sys_{\text{sync},1} \geq_{\text{sync}}^{f'} Sys_{\text{sync},2}$ .  $\blacksquare$

So far, we have shown that asynchronous simulatability among the asynchronous representations implies synchronous simulatability, i.e.,

$$\varphi_{Sys}(Sys_{\text{sync},1}) \geq_{\text{async}} \varphi_{Sys}(Sys_{\text{sync},2}) \Rightarrow Sys_{\text{sync},1} \geq_{\text{sync}} Sys_{\text{sync},2}.$$

We already briefly stated that the converse implication does not hold in general. We had to show that for each configuration  $conf_{\text{async},1} \in \text{Conf}_{\text{poly}}(\varphi_{Sys}(Sys_{\text{sync},1}))$  there exists an indistinguishable configuration  $conf_{\text{async},2} \in \text{Conf}_{\text{poly}}(\varphi_{Sys}(Sys_{\text{sync},2}))$  provided that  $Sys_{\text{sync},1} \geq_{\text{sync}} Sys_{\text{sync},2}$ . However, both the honest user and the adversary may have clock-out ports and they can alternately schedule each other (and also the system erratically), which cannot be captured by a fixed synchronous clocking scheme, so we cannot exploit our assumption  $Sys_{\text{sync},1} \geq_{\text{sync}} Sys_{\text{sync},2}$ .

Anyhow, it is sufficient for our purpose to show that the claim holds for at least those configurations where the honest user  $H_{\text{async}}$  fits the form  $\varphi_M(H_{\text{sync}})$  for a synchronous machine  $H_{\text{sync}}$ . We denote this version of simulatability for the restricted class of users by  $\geq_{\text{async,H}}$  in the sequel. It is immediately clear that the first embedding theorem also holds for the weaker precondition  $\varphi_{Sys}(Sys_{\text{sync},1}) \geq_{\text{async,H}} \varphi_{Sys}(Sys_{\text{sync},2})$ , since we only have to derive an indistinguishable configuration for users of the special form  $\varphi(H_{\text{sync}})$ , and the user remains unchanged at simulatability. We can now present the second embedding theorem.

**Theorem 4.** (*Second Embedding Theorem*) *Let two arbitrary synchronous systems  $Sys_{\text{sync},1}$  and  $Sys_{\text{sync},2}$  with clocking schemes  $\kappa_1$  and  $\kappa_2$  be given such that  $\kappa_1$  is valid, and let  $Sys_{\text{sync},1} \geq_{\text{sync}}^f Sys_{\text{sync},2}$  for a valid mapping  $f$ . Then*

$$\varphi(Sys_{\text{sync},1}) \geq_{\text{async,H}}^{f'} \varphi(Sys_{\text{sync},2})$$

where  $f'$  is defined as  $\varphi(\hat{M}_2, S_2) \in f'(\varphi(\hat{M}_1, S_1)) :\Leftrightarrow (\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$ .  $\square$

## 5 Deriving Synchronous Theorems from Asynchronous Ones

Recall that our long-term goal is to avoid proving theorems for both models. We now briefly show how our two embedding theorems can be used to circumvent this problem. One of the most important theorems of both models is transitivity of the relation  $\geq$ .

**Lemma 3.** (*Transitivity*) *If  $Sys_1 \geq^{f_1} Sys_2$  and  $Sys_2 \geq^{f_2} Sys_3$ , then  $Sys_1 \geq^{f_3} Sys_3$  with  $f_3(\hat{M}_1, S)$  being the union of the sets  $f_2(\hat{M}_2, S)$  with  $(\hat{M}_2, S) \in f_1(\hat{M}_1, S)$ .*  $\square$

This has been proven in [28] for the synchronous and in [30] for the asynchronous model. We now exemplarily show how to derive the synchronous version from the asynchronous one using our previous results.

**Lemma 4.** *Assume that the asynchronous version of the transitivity lemma (Lemma 3) has already been proven, then the synchronous version holds as well.*  $\square$

*Proof.* We omit the superscripts  $f_i$  for the sake of readability. Let synchronous systems  $Sys_1$ ,  $Sys_2$ , and  $Sys_3$  be given such that  $Sys_1 \geq_{\text{sync}} Sys_2$  and  $Sys_2 \geq_{\text{sync}} Sys_3$ . We

have to show that  $Sys_1 \geq_{sync} Sys_3$  holds, provided that asynchronous transitivity has already been proven. Our second embedding theorem implies  $\varphi(Sys_1) \geq_{async,H} \varphi(Sys_2)$  and  $\varphi(Sys_2) \geq_{async,H} \varphi(Sys_3)$ . Obviously, the asynchronous version of transitivity is applicable to the relation  $\geq_{async,H}$  instead of  $\geq_{async}$  as well, since it is a special case only, and the honest user remains unchanged at simulatability. Thus, we can apply our (already proven) asynchronous version of the transitivity lemma, which yields  $\varphi(Sys_1) \geq_{async,H} \varphi(Sys_3)$ . Now, we use our first embedding theorem in conjunction with its subsequent remarks (stating that the theorem holds as well for the restricted version  $\geq_{async,H}$  of simulatability) yielding  $Sys_1 \geq_{sync} Sys_3$ . ■

This proof technique is applicable to almost all theorems that rely on simulatability. As the most important example, we name the preservation theorem [29, 4], which states that integrity properties expressed in linear-time logic are preserved under simulatability. The proof of this theorem is difficult and comprises several pages for both models. Using our work, the synchronous proof could as well be omitted.

## References

1. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.
2. 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.
3. 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 *LNCS*, pages 3–22. Springer, 2000.
4. 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 *LNCS*, pages 675–686. Springer, 2003.
5. 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 *LNCS*, pages 310–329. Springer, 2002.
6. M. Backes, B. Pfitzmann, and M. Waidner. A universally composable cryptographic library. IACR Cryptology ePrint Archive 2003/015, Jan. 2003. <http://eprint.iacr.org/>.
7. D. Beaver. Secure multiparty protocols and zero knowledge proof systems tolerating a faulty minority. *Journal of Cryptology*, 4(2):75–122, 1991.
8. M. Bellare, R. Canetti, and H. Krawczyk. A modular approach to the design and analysis of authentication and key exchange protocols. In *Proc. 30th Annual ACM Symposium on Theory of Computing (STOC)*, pages 419–428, 1998.
9. M. Burrows, M. Abadi, and R. Needham. A logic for authentication. Technical Report 39, SRC DIGITAL, 1990.
10. R. Canetti. Studies in secure multiparty computation and applications. Department of Computer Science and Applied Mathematics, The Weizmann Institute of Science, June 1995, revised March 1996, 1995.
11. R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 3(1):143–202, 2000.
12. 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.

13. D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
14. C. Dwork, M. Naor, and A. Sahai. Concurrent zero-knowledge. In *Proc. 30th Annual ACM Symposium on Theory of Computing (STOC)*, pages 409–418, 1998.
15. S. Goldwasser and L. Levin. Fair computation of general functions in presence of immoral majority. In *Advances in Cryptology: CRYPTO '90*, volume 537 of *LNCS*, pages 77–93. Springer, 1990.
16. S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–207, 1989.
17. 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.
18. C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science, Prentice Hall, Hemel Hempstead, 1985.
19. P. Laud. Semantics and program analysis of computationally secure information flow. In *Proc. 10th European Symposium on Programming (ESOP)*, pages 77–91, 2001.
20. 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 *LNCS*, pages 147–166. Springer, 1996.
21. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Francisco, 1996.
22. J. K. Millen. The interrogator: A tool for cryptographic protocol security. In *Proc. 5th IEEE Symposium on Security & Privacy*, pages 134–141, 1984.
23. B. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, 1994.
24. J. Neveu. *Mathematical Foundations of the Calculus of Probability*. Holden-Day, 1965.
25. 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 *LNCS*, pages 748–752. Springer, 1992.
26. L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Cryptology*, 6(1):85–128, 1998.
27. 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>.
28. B. Pfitzmann, M. Schunter, and M. Waidner. Secure reactive systems. Research Report RZ 3206, IBM Research, 2000.
29. 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.
30. 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.
31. R. Segala and N. Lynch. Probabilistic simulation for probabilistic processes. *Nordic Journal of Computing*, 2(2):250–273, 1995.
32. D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Proc. 27th Symposium on Principles of Programming Languages (POPL)*, pages 268–276, 2000.
33. S.-H. Wu, S. A. Smolka, and E. W. Stark. Composition and behaviors of probabilistic I/O automata. *Theoretical Computer Science*, 176(1–2):1–38, 1997.
34. A. C. Yao. Theory and applications of trapdoor functions. In *Proc. 23rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 80–91, 1982.