Secure Asynchronous Reactive Systems *

Michael Backes, Birgit Pfitzmann, and Michael Waidner
IBM Zurich Research Laboratory, Rüschlikon, Switzerland
{mbc,bpf,wmi}@zurich.ibm.com

Abstract

We present a rigorous model for secure reactive systems in asynchronous networks. It captures both computational aspects of security as needed for cryptography, and abstractions as needed in typical theorem provers and model checkers, with clear refinement relations within and between the layers of abstraction.

The term "reactive" means that the system interacts with its users multiple times, e.g., in many concurrent protocol runs. We define a distributed scheduling scheme that allows both probabilistic executions with realistic adversarial scheduling and the representation of local submachines. Polynomial runtime in the reactive case is defined precisely, to capture polynomially bounded users and adversaries, as well as actions of runtime-restricted machines in the presence of more powerful adversaries. The model captures the strongest possible adversarial behaviors, but is defined in a modular way so that other trust models can be considered, e.g., some secure or authentic channels and static or adaptive adversaries.

Security-preserving refinement is different from normal refinement because it includes confidentiality. We capture it by the notion of *reactive simulatability*, extending existing cryptographic notions from much simpler scenarios to the general reactive case.

Keywords: security, cryptography, simulatability, formal methods, reactive systems

1 Introduction

In the early days of security research, cryptographic protocols were designed in an iterative way: someone proposed a protocol, someone else found an attack, an improved version was proposed, and so on, until no further attacks were found. Today it is commonly accepted that this approach gives no security guarantee. Too many seemingly simple and secure protocols have been found flawed over the years. Moreover, problems like n-party key agreement, fair contract signing, anonymous communication, electronic auctions or payments are too complex for this approach. Secure protocols—or more generally, secure reactive systems—need a proof of security before being acceptable.

One possibility for conducting such a proof is the cryptographic approach. This means reduction proofs between the security of the overall system and the security of the cryptographic primitives: One shows that if one could break the overall system, one could also break one of the underlying cryptographic primitives with respect to its cryptographic definition, e.g., adaptive chosen-message security for signature schemes. This reduction approach was introduced for cryptographic primitives in [18, 30, 63], and has since been used throughout theoretical

*A preliminary version of this paper appeared at IEEE Symposium on Security and Privacy, Oakland, May 2001.

cryptography. The best-known application to protocols is the handling of authentication protocols originating in [17]. In principle, proofs in this approach are as rigorous as typical proofs in mathematics. In practice, however, human beings are extremely fallible with this type of proofs when protocols are concerned. This is mainly due to the distributed-systems aspects of the protocols. It is well-known from non-cryptographic distributed systems that many wrong protocols have been published even for very small problems. Hand-made proofs are highly error-prone because following all the different cases how actions of different machines interleave is extremely tedious. Humans tend to take wrong shortcuts and do not want to proof-read such details in proofs by others. If the protocol contains cryptography, this obstacle is even much worse: Already a rigorous definition of the goals and of the protocol itself gets more complicated, and there is no general framework for this. Often not only trace properties (integrity) have to be proven but also confidentiality (also called secrecy). Further, in principle the complexity-theoretic reduction has to be carried out across all these cases, and it is not at all trivial to do this rigorously. In consequence, there is almost no real cryptographic proof of a larger protocol, and several times supposedly proven, relatively small systems were later broken, e.g., [53, 24, 34].

The other possibility is to use formal methods. There one leaves the tedious parts of proofs to machines, i.e., model checkers or automatic theorem provers. None of these tools, however, is currently able to deal with reduction proofs.[1] Nobody even thought about this for a long time, because one felt that protocol proofs could be based on simpler, idealized abstractions of cryptographic primitives. Almost all these abstractions are variants of the Dolev-Yao model [25], which represents all cryptographic primitives as operators of a term algebra with cancellation rules. For instance, public-key encryption is represented by operators E for encryption and D for decryption with one cancellation rule, $\mathsf{D}(\mathsf{E}(m)) = m$ for all m. Encrypting a message m twice in this model does not yield another message from the basic message space but the term $\mathsf{E}(\mathsf{E}(m))$. Further, the model assumes that two terms whose equality cannot be derived with the cancellation rules are not equal, and every term that cannot be derived is completely secret. This simplifies proofs of larger protocols considerably and gave rise to a large body of literature on analyzing the security of protocols using formal verification techniques (a very partial list includes [47, 44, 36, 19, 45, 37, 41, 52, 60, 1]). However, originally there was no foundation at all for such assumptions about real cryptographic primitives, and thus no guarantee that protocols proved with these tools were still secure when implemented with real cryptography. Although no previously proved protocol has been broken when implemented with standard provably secure cryptosystems, this was clearly an unsatisfactory situation, and artificial counterexamples can be constructed.

Our overall goal is to combine the best of these two worlds: the rigorous treatment of cryptographic operations from the cryptographic approach, and the clearer protocol specifications, availability of abstractions, and automation from the formal-methods approach. The first two things we need for this are a joint, rigorously defined *mathematical model* for representing protocols, and a definition of *security-preserving refinement* that enables rigorous proofs that certain abstractions are sound. This is what we provide in this paper. Section 1.1 surveys next steps that we have already taken towards the overall goal and to validate that the basic definitions presented here are suitable for this purpose.

Prior to this work, no such model existed. Formal methods for security come with rigorous

[1] Efforts are under way to formulate syntactic calculi for probabilism and polynomial-time considerations, in particular [48, 49, 35] and, as a second step, to encode them into proof tools. However, this approach can not yet handle protocols with any degree of automation. It is complementary to, rather than competing with, the approach of making simple deterministic abstractions of cryptography and working with those wherever cryptography is only used in a blackbox way.

system models, but all the classical ones do not allow any representation of actual cryptography. There are no probabilistic behaviors, no way to capture polynomially bounded adversaries (in other terms attackers or intruders), and the adversary models are quite restricted because they immediately relate to the symbolic modeling of cryptographic objects. For instance, partial information cannot be expressed in such models. Further, there is no general notion of security-preserving refinement for protocols. In the general distributed-computing area, probabilistic asynchronous system models exist. However, they do not come with polynomial-time considerations, and we discuss below why we use a new scheduling scheme even on the abstract layer. Further, these models do not come with security definitions. On the cryptography side, there was no rigorous model for general reactive systems at all. In particular there was no detailed model of scheduling in the asynchronous case, no details of the notion of polynomial time in the reactive case, and no clear notion of how a cryptographic system interacts with users at all (and not only an adversary). Modeling users separately from the adversary is important for confidentiality considerations because one has to represent external secrets whose probabilistic choice may be influenced by active attacks and that might be used in multiple interactions of the users with the system and the adversary. Cryptography has a security-preserving refinement notion, called simulatability, but only for the restricted cases that have been modeled so far. We extend this to *reactive simulatability*.

Our model provides all these missing features. We now highlight the main design decisions that we made to allow for such a general model.

Our primary machine model is *probabilistic state-transition machines*, similar to probabilistic I/O automata as in [42, 57]. Other terms for such machines are extended finite-state automata or state-transition machines. For the computational complexity aspects, we define implementations of such machines by probabilistic interactive Turing machines. We do not use Turing machines as the sole or primary model, in contrast to prior cryptographic literature, because the I/O automata enable us to express non-cryptographic protocol parts and abstractions from cryptography in a well-defined way unencumbered with Turing-machine details. This is important for the desired accessibility of the resulting model to existing theorem provers and model checkers. We do not define one specific formal language for expressing the actions of the I/O automata. There are many such languages in formal methods both for security and in general. Most of them have a semantics that is closely related to I/O automata. Thus we hope it will be easy to use our automata variant as the semantics for many formal methods, and to encode it directly into automated tools that are not based on a specific language for reactive systems but, e.g., directly on higher-order logic.

We made one addition to individual machines compared with other I/O automata models, in order to enable machines to have polynomial runtime independent of their environment without being automatically vulnerable to denial-of-service attacks by long messages: We allow state-dependent *length bounds* on the inputs that a machine will read from each channel.

The second distinctive aspect of our general machine and execution model is a *distributed scheduling* scheme that enables us to represent all realistic scenarios. In well-known, non security-specific probabilistic frameworks like [58, 61], the order of events is chosen by a probabilistic scheduler that has full information about the system. However, this can give the scheduler too much power in a cryptographic scenario. In cryptology, the typical understanding (closest to a rigorous definition in [20]) is that the adversary schedules everything, but only with realistic information, in terms of both observations and computational capabilities. This corresponds to making a certain subclass of schedulers explicit for the model from [58] and joining it with the functional adversary. However, if one splits a machine into local submachines, or defines intermediate systems for the purpose 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. Our solution is a distributed definition of scheduling. It allows a machine that has been scheduled to schedule certain (statically fixed) other machines itself.

As the first specialization of the model for security, we define how a set of specified machines interacts with arbitrary users and an arbitrary adversary. We introduce *service ports* to distinguish, from the point of view of a system specification, where honest users can connect (and then get a certain service guarantee by the system) and what possibilities are only available for adversaries. This enables us to model in a realistic way that users are on the one hand arbitrary and unknown at the system-design time, and on the other hand not arbitrarily stupid, e.g., in the sense of breaking into their own machines. Secondly, we define important cases of how one can derive actual system structures from intended system structures (i.e., from the design of the totally correct system) for different *trust models*, e.g., how adversaries can corrupt machines and modify messages on channels.

The notion of *reactive simulatability* captures the idea of refinement that preserves not only integrity properties, but also confidentiality properties. Intuitively it can be stated as follows, when applied to the relation between an implementation and a specification. (Other terms are real and ideal system, or in special cases cryptographic and abstract system.) Everything that can happen to users of the implementation in the presence of arbitrary adversaries can also happen to the same users with the specification, where attack capabilities are usually much more restricted. In particular, it comprises confidentiality because the notion of what happens to users, called their *view*, not only includes their in- and outputs to the system, but also their communication with the adversary. This includes whether the adversary can guess secrets of the users or partial information about them. The view is actually a probability distribution, so that the definition captures guessing probabilities as well as probabilities of other events like forgeries. Composition and preservation theorems from our subsequent work (see Section 1.1) based on this model show that reactive simulatability has the properties expected from a refinement notion: First, if we design a larger system based on a specification of a subsystem, and later plug the implementation of the subsystem in, the entire implementation of the larger systems is as secure as its design in the same sense of reactive simulatability. Secondly, if we prove specific security properties for the specification, they also hold for the implementation.

1.1 Our Subsequent Work

We already validated the model presented here in several ways. As mentioned in the introduction, we validated that reactive simulatability has the properties expected of a refinement notion. First, it is indeed transitive. Secondly, it has a composition theorem that states that substituting a refined system (an implementation) for the original system (a specification) within a larger system is permitted [56]. The composition theorem can be extended to securely composing a polynomial number of systems [13]. (The idea to do that is from [22].) Thirdly, we defined computational versions of various security property classes and proved preservation theorems for them under reactive simulatability. We did this for integrity [6], transitive and non-transitive non-interference [8, 10], i.e., absence of information flow, and a class of liveness properties [11].

We showed that concrete systems can be proven secure in the sense of reactive simulatability. Our focus is on providing abstract specifications of cryptographic systems and proving real cryptographic implementations secure with respect to them. By *abstract* we mean simple deterministic systems without any cryptographic objects, so that these abstractions should be easily usable in typical design tools for larger systems and in tool-supported verification of those

larger systems. We did this for secure message transmission [56, 7], group key agreement [59], and most importantly a cryptographic library with nested terms like the Dolev-Yao model [12]. A proof of the Needham-Schroeder-Lowe protocol based on this library [9] shows that one can rigorously prove protocols based on this library in much the same way as with the Dolev-Yao model. Reactive simulatability is also useful for lower-layer proofs, e.g., of reactive encryption security from normal encryption security within [56] and of reactive Diffie-Hellman security within [59].

These papers on concrete systems contain detailed proof techniques for the necessary hand-proofs of the security of cryptographic primitives with respect to first abstractions, e.g., so-called cryptographic bisimulations. These are probabilistic bisimulations with imperfections and, in the case of the nested cryptographic library, an embedded static information-flow analysis. We emphasize that it is quite easy to guess almost correct suitable abstractions of cryptography; the major part of the work lies in getting the details right and making a rigorous proof. Several abstractions presented by others with less detailed proofs have been broken [34, 5] (where the first paper attributes another such attack to Damgård).

Finally, we showed that automated proof tools can handle small examples in our model, based on the the secure message-transmission abstraction [7, 6].

1.2 Further Related Literature

Several researchers pursue the goal of conducting security proofs that allow the use of formal methods and tools while retaining a sound cryptographic semantics. The earliest such work is [39, 40]. They define and verify the cryptographic security of specific systems directly using a formal language, the π-calculus. The notion of security is observational equivalence. This is even stronger than reactive simulatability because the entire environment (corresponding to our users and adversary together) must not be able to distinguish the implementation and the specification. However, it excludes many abstractions because a typical abstract specification is one machine and the real system is distributed, so that an adversary can already distinguish them by their structure. Correspondingly, the concrete specifications used were not abstract; they essentially comprise the actual protocols including all cryptographic details. There was no tool support yet, as even the specifications involve ad-hoc notations, e.g., for generating random primes. A very similar motivation to our paper was given in [43]. There however, cryptographic systems are restricted to the usual equational specifications following the Dolev-Yao model [25] and the semantics is not probabilistic. Hence the abstraction from cryptography is no more faithful than in other papers on formal methods in security. Moreover, only passive adversaries are considered and only one class of users, called environment. The author actually remarks that the model of what the adversary learns from the environment is not yet general, and that general theorems for the abstraction from probabilism would be useful. Our model solves these problems. In [3, 2, 38] it is shown that a slight variation of the standard Dolev-Yao model is cryptographically faithful specifically for symmetric encryption, but only under passive attacks. Hence these papers do not contain any reactive model and no general notions of security.

Simulatability was first sketched for secure multi-party function evaluation, i.e., for the computation of one output tuple from one tuple of secret inputs from each participant in [62] and defined (with different degrees of generality and rigorosity) in [29, 14, 46, 21]. Among these, [14] and an unpublished longer version of [46] contain the earliest detailed execution models for cryptographic systems that we are aware of. Both are synchronous models. Problems such as the separation of users and adversaries, or defining runtime restrictions in the face of continuous external inputs, do not occur in this case. A composition theorem for non-reactive

simulatability was proven in [21]. The idea of simulatability was subsequently also used for specific reactive problems, e.g., [26, 16, 23], without a detailed or general definition. In a similar way it was used for the construction of generic solutions for large classes of reactive problems [28, 27, 32] (usually yielding inefficient solutions and assuming that all parties take part in all subprotocols). A reactive simulatability definition was first proposed (after some earlier sketches, in particular in [28, 54, 21]) in [32]. It is synchronous, covers a restricted class of protocols (straightline programs with restricted operators, in view of the constructive result of this paper), and simulatability is defined for the information-theoretic case only, where it can be done with a quantification over input sequences instead of active honest users.

We first presented a synchronous version of a general reactive model and reactive simulatability in [55]. After that, and later but independently to the conference version [56] of the current paper, an asynchronous version of a general reactive model and reactive simulatability was also given in [22]. The model parts that are relatively well-defined seem to us a strict subset of our model: It defines cryptographic systems with adaptive adversaries as discussed in Section 7.3, always with polynomial-time users and adversaries. It contains Turing machines only, i.e., there is no explicit abstraction layer. The simulatability definition corresponds to the universal case of ours. (Besides the model, the paper contains a composition theorem which was more general than ours at that time, while we had a property preservation theorem. It also contains some sketches, while we had decided to only publish parts that we had actually defined and proved.)

1.3 Overview of this Paper

In Section 2 we give an informal overview of our model and of reactive simulatability. Section 3 introduces notation. Section 4 defines the general system model, i.e., machines, both their abstract version and their computational realization, and executions of collections of machines. Section 5 defines the security-specific system model, i.e., systems with users and adversaries. Section 6 defines reactive simulatability, i.e., our notion of secure refinement. Section 7 shows how to represent typical trust models, i.e., assumptions about the adversary, such as static threshold models and adaptive adversaries, with secure, authenticated and insecure channels. Section 8 concludes the paper.

2 Informal Overview

The most basic and most general part of our model are the definitions of asynchronous reactive systems with distributed scheduling.

We consider sets of asynchronously communicating probabilistic state machines; we call such sets *collections* of machines. The left-hand side of Figure 1 sketches a collection of three machines connected via channels represented by solid arrows. To model asynchronous timing, messages sent between the machines stay on their respective channel until they are scheduled. Technically, each channel contains an additional machine called a buffer, which stores messages in transit. This is shown on the right-hand side of Figure 1. When M_2 sends a message to M_3, this message is stored in the buffer. An incoming message at a clock channel for the buffer, represented by the dashed arrow, is interpreted as a number i, and the i-th message in the buffer is removed and output to M_3. Buffers need not be specified explicitly; a completion operator adds all necessary buffers to a collection of normal machines.

As motivated in Section 1, we allow a machine that has been scheduled to schedule certain other machines itself. This is done by giving the machine the control over the clock channels

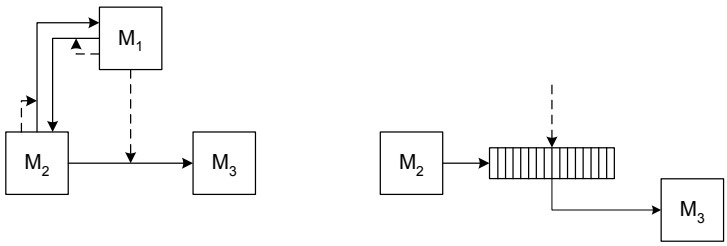

Figure 1: A collection of three machines is shown on the left. Solid arrows represent channels. The dashed arrow depicts that M_1 schedules messages on the channel from M_2 to M_3. Each channel implicitly contains a buffer for storing messages in transit, shown on the right.

of certain buffers. In Figure 1, the machine M_1 can schedule messages sent from M_2 to M_3, while the channels between M_1 and M_2 show procedure-call-style local interaction. Where one wants to express that an adversary schedules everything, one simply gives the adversary all the scheduling rights. Problems with purely adversarial scheduling were already noted in [40]; hence they schedule secure channels with uniform probability before adversary-chosen events. However, that introduces a certain amount of global synchrony. Furthermore, we do not require local scheduling for all secure channels; they may be blindly scheduled by the adversary (i.e., without even seeing if there are messages on the channel). For instance, this models cases where the adversary has a global influence on relative network speed.

Probability spaces for runs are defined in detail for such collections of machines, as well as the view of a subset of the machines. These definitions are useful beyond the more security-specific system classes considered later. Further, the Turing-machine realization and runtime considerations are defined in this generality.

Security-specific *structures* are defined as collections of machines with distinguished service ports for the honest users, as explained in the introduction. Such structures are augmented by arbitrary machines H and A representing the honest users and the adversary, who can interact. We then speak of a *configuration*. For configurations, we also introduce specific families of executions corresponding to different security parameters for cryptographic aspects.

In the presence of adversaries, the structure of correct machines running may not be the *intended structure* that the designer originally planned. For instance, some machines might have been corrupted; hence they are missing from the actual structure and the adversary took over their connections. We model this by defining a *system* as a set of possible actual structures. A system is typically derived automatically from an intended structure and a *trust model*. We define this for static and adaptive adversaries, arbitrary access structures limiting the corruption capabilities of an adversary, and different channel types. While one typically considers all basic channels insecure in a security protocol, secure or authentic channels are useful to model initialization phases, e.g., the assumption that a public-key infrastructure exists. In contrast to some formal methods which immediately abstract from cryptography, we cannot represent this by a fixed initial key set because we need probabilities over the key generation for security; further, it would not allow a polynomial number of keys to be chosen reactively.

Reactive simulatability, our notion of secure refinement, is defined for individual structures and lifted entire systems. Two structures $struc_1$ and $struc_2$ can be compared if they have the same service ports, so that the same honest users can connect to them. In other words, they offer the same interface for the design of a larger system, so that either $struc_1$ or $struc_2$ can be plugged into that system. Now $struc_1$ is considered *at least as secure as* $struc_2$, written $struc_1 \geq struc_2$, if whatever any adversary A_1 can do to any honest user H in $struc_1$, some adversary A_2 can do

to the same H in $struc_2$ essentially with the same probability. More precisely, the families of views of H in these two configurations are indistinguishable. We present several flavors of this definition. Structure $struc_1$ typically represents a real distributed cryptographic primitive or protocol, while $struc_2$ is a specification in terms of one joint deterministic machine. This gives us the desired rigorous link between real cryptographic systems and abstractions suitable for automated theorem provers and model checkers.

3 Notation

Let $Bool := \{\mathsf{true}, \mathsf{false}\}$. For an arbitrary set A, let $\mathcal{P}(A)$ denote its powerset. Further, let A^n denote the set of sequences over A with index set $\mathcal{I} = \{1, \ldots, n\}$ for $n \in \mathbb{N}_0$, $A^* := \bigcup_{n \in \mathbb{N}_0} A^n$, and A^∞ the set of sequences over A with index set $\mathcal{I} = \mathbb{N}$. We write a sequence over A with index set $\mathcal{I}$ as $S = (S_i)_{i \in \mathcal{I}}$, where $\forall i \in \mathcal{I}\colon S_i \in A$. Let $\circ$ denote sequence concatenation, and $()$ the empty sequence. For a sequence $S \in A^* \cup A^\infty$ we define the following notation:

- For a function $f\colon A \to A'$, let $f(S)$ apply f to each element of S, retaining the order.
- For a predicate $pred\colon A \to Bool$, let $(S_i \in S \mid pred(S_i))$ denote the subsequence of S containing those elements S_i of S with $pred(S_i) = \mathsf{true}$, retaining the order.
- Let $\mathsf{size}(S)$ denote the length of S, i.e., $\mathsf{size}(S) := |\mathcal{I}|$.
- For $l \in \mathbb{N}_0$, let $S\lceil_l$ denote the l-element prefix of S, and $S[l]$ the l-th element of S with the convention that $S[l] = \epsilon$ if $\mathsf{size}(S) < l$.

In the following, we assume that a finite alphabet $\Sigma \supseteq \{0, 1\}$ is given, where $\sim, !, ?, \hookrightarrow, \triangleleft \notin \Sigma$. Then Σ^* denotes the strings over Σ. Let ϵ be the empty string and $\Sigma^+ := \Sigma^* \setminus \{\epsilon\}$. All notation for sequences can be used for strings, such as $\circ$ for string concatenation, but $\circ$ is often omitted.

For representing natural numbers and sequences of strings as strings, we assume bijective functions $\mathsf{nat}\colon \Sigma^* \to \mathbb{N}$ with the convention $\mathsf{nat}(1) = 1$ and $\iota\colon (\Sigma^*)^* \to \Sigma^*$. We assume that standard operations are efficiently (polynomial-time) computable in these encodings; concretely we need this for inverting the function ι, for appending an element to a sequence of strings, and for retrieving and removing the $\mathsf{nat}(u)$-th element from a sequence of strings.

For an arbitrary set A let $\mathsf{Prob}(A)$ denote the set of all finite probability distributions over A. For a probability distribution D over A, the probability of a predicate $pred\colon A \to Bool$ is written $\mathsf{Pr}_D(pred)$. If x is a random variable over A with distribution D, we also write $\mathsf{Pr}_D(pred(x))$. In both cases we omit D if it is clear from the context.

We write $:=$ for deterministic and $\leftarrow$ for probabilistic assignment. The latter means that for a function $f\colon X \to \mathsf{Prob}(Y)$, we write $y \leftarrow f(x)$ to denote that y is chosen according to the distribution $f(x)$. For such a function f we write $y := f(x)$ if there exists $y' \in Y$ with $\mathsf{Pr}_{f(x)}(y') = 1$. If the function f is clear from the context, we also write $x \to_p y$ for $\mathsf{Pr}_{f(x)}(y) = p$, and $\to$ for $\to_1$. Further, we sometimes treat $f(x)$ as a random variable instead of a distribution, e.g., by writing $\mathsf{Pr}(f(x) = y)$ for $\mathsf{Pr}_{f(x)}(y)$.

4 Asynchronous Reactive Systems

In this section, we define our model of interacting probabilistic machines with distributed scheduling and with computational realizations.

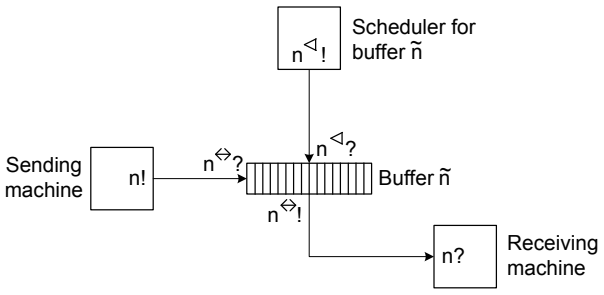

Figure 2: Ports and buffers.

4.1 Ports

Machines can exchange messages with each other via *ports*. Intuitively, a port is a possible attachment point for a channel when a machine of Figure 1 is considered in isolation. As in many other models, channels in collections of machines are specified implicitly by naming conventions on the ports; hence we define port names carefully. Figure 2 gives an overview of the naming scheme; it can be seen as a yet more detailed view of the right-hand side of Figure 1.

Definition 4.1 *(Ports) Let* $\mathcal{P} := \Sigma^+ \times \{\epsilon, \leftrightarrow, \triangleleft\} \times \{!, ?\}$. *Then* $p \in \mathcal{P}$ *is called a* port. *For* $p = (\mathsf{n}, l, d) \in \mathcal{P}$, *we call* $\mathsf{name}(p) := \mathsf{n}$ *its* name, $\mathsf{label}(p) := l$ *its* label, *and* $\mathsf{dir}(p) := d$ *its* direction. ◇

In the following we usually write (n, l, d) as $\mathsf{n}ld$, i.e., as string concatenation. This is possible without ambiguity, since the mapping $\varphi\colon \mathcal{P} \to \Sigma^+ \circ \{\epsilon, \leftrightarrow, \triangleleft\} \circ \{!, ?\}$ with $\varphi((\mathsf{n}, l, d)) := \mathsf{n}ld$ is bijective because of the precondition $!, ?, \leftrightarrow, \triangleleft \notin \Sigma$.

The name of a port serves as an identifier and will later be used to define which ports are connected to each other. The direction of a port determines whether it is a port where inputs occur or where outputs are made. Inspired by the CSP [33] notation, this is represented by the symbols ? and !, respectively. The label becomes clear in Definition 4.3.

Definition 4.2 *(In-Ports, Out-Ports) A port* (n, l, d) *is called an* in-port *or* out-port *iff* $d = ?$ *or* $d = !$, *respectively. For a set* P *of ports let* $\mathsf{out}(P) := \{p \in P \mid \mathsf{dir}(p) = !\}$ *and* $\mathsf{in}(P) := \{p \in P \mid \mathsf{dir}(p) = ?\}$. *For a sequence* P *of ports let* $\mathsf{out}(P) := (p \in P \mid \mathsf{dir}(p) = !)$ *and* $\mathsf{in}(P) := (p \in P \mid \mathsf{dir}(p) = ?)$. ◇

The label of a port determines the port's role in the upcoming scheduling model. Roughly, ports p with $\mathsf{label}(p) \in \{\leftrightarrow, \triangleleft\}$ are used for scheduling whereas ports p with $\mathsf{label}(p) = \epsilon$ are used for "usual" message transmission.

Definition 4.3 *A port* $p = (\mathsf{n}, l, d)$ *is called a* simple port, buffer port *or* clock port *iff* $l = \epsilon$, $\leftrightarrow$, *or* $\triangleleft$, *respectively.* ◇

After introducing ports on their own, we now define the *low-level complement* of a port. This will later be the port which it connects to. Two connected ports have identical names and different directions. The relationship of their labels l and l' is visible in Figure 2, i.e., $l = l' = \triangleleft$ or $\{l, l'\} = \{\epsilon, \leftrightarrow\}$. The remaining notation of Figure 2 is explained below. In particular, "Buffer $\widetilde{\mathsf{n}}$" represents the network between the two simple ports $\mathsf{n}!$ and $\mathsf{n}?$. If we are not interested in the network details then we regard the ports $\mathsf{n}!$ and $\mathsf{n}?$ as connected; thus we call them *high-level complements* of each other.

Definition 4.4 *(Complement Operators) Let $p = (\mathsf{n}, l, d)$ be a port.*

a) The low-level complement *p^c of p is defined as $p^c := (\mathsf{n}, l', d')$ such that $\{d, d'\} = \{!, ?\}$, and $l = l' = {}^{\triangleleft}$ or $\{l, l'\} = \{\epsilon, {}^{\leftrightarrow}\}$.*

b) If p is simple, the high-level complement *p^C of p is defined as $p^C := (\mathsf{n}, l, d')$ with $\{d, d'\} = \{!, ?\}$.*

$\diamond$

4.2 Machines

After introducing ports, we now define *machines*. Our primary machine model is probabilistic state-transition machines, similar to probabilistic I/O automata as in [42, 57]. A machine has a *sequence of ports*, containing both in-ports and out-ports, and a set of *states*, comprising sets of *initial* and *final states*. If a machine is switched, it receives an input tuple at its input ports and performs its *transition function* yielding a new state and an output tuple in the deterministic case, or a finite distribution over the set of states and possible outputs in the probabilistic case. Furthermore, each machine has state-dependent bounds on the length of the inputs accepted at each port to enable flexible enforcement of runtime bounds, as motivated in Section 1. The parts of an input that are beyond the length bound are ignored. The value ∞ denotes that arbitrarily long inputs are accepted.

Definition 4.5 *(Machines) A* machine *is a tuple*

$$\mathsf{M} = (name, Ports, States, \delta, l, Ini, Fin)$$

where

- *$name \in \Sigma^+ \circ \{\sim, \epsilon\}$ is called the* name *of M,*
- *$Ports$ is a* finite sequence of ports *with pairwise distinct elements,*
- *$States \subseteq \Sigma^*$ is called a set of* states*,*
- *δ is called a* probabilistic state-transition function *and defined as follows:*
  Let $\mathcal{I} := (\Sigma^*)^{|\mathsf{in}(Ports)|}$ and $\mathcal{O} := (\Sigma^*)^{|\mathsf{out}(Ports)|}$ denote the input set *and* output set *of M, respectively. Then $\delta\colon States \times \mathcal{I} \to \mathsf{Prob}(States \times \mathcal{O})$ with the following restrictions:*
  - *If $I = (\epsilon, \ldots, \epsilon)$, then $\delta(s, I) := (s, (\epsilon, \ldots, \epsilon))$ deterministically.*
  - *$\delta(s, I) = \delta(s, I\lceil_{l(s)})$ for all $I \in \mathcal{I}$, where $(I\lceil_{l(s)})_i := I_i\lceil_{l(s)_i}$ for all $i \in \{1, \ldots, |\mathsf{in}(Ports)|\}$. (The parts of an input beyond the length bounds are ignored.)*
- *$l\colon States \to (\mathbb{N}_0 \cup \{\infty\})^{|\mathsf{in}(Ports)|}$ is called a* length function*; we require $l(s) = (0, \ldots, 0)$ for all $s \in Fin$,*
- *$Ini, Fin \subseteq States$ are called the sets of* initial *and* final states*.*

$\diamond$

In the following, we write $name_{\mathsf{M}}$ for the name of machine M, $Ports_{\mathsf{M}}$ for its sequence of ports, $States_{\mathsf{M}}$, Ini_{M}, Fin_{M} for its respective sets of states, $\mathcal{I}_{\mathsf{M}}$, $\mathcal{O}_{\mathsf{M}}$ for its input and output set, l_{M} for its length function and δ_{M} for its transition function.

The chosen representation makes the transition function δ independent of the port names; this enables port renaming in our proofs. The requirement for ϵ-inputs, i.e., the first restriction on δ, means that it does not matter if we switch a machine without inputs or not, i.e., there are no spontaneous transitions. The second restriction means that the part of each input beyond the current length bound for its port is ignored. In particular one can mask an input by a length bound 0 for a port. The restriction on l means that a machine ignores all inputs if it is in a final state, and hence it no longer switches.

We will often need the *port set* of a machine instead of its port sequence. Port sets are easily extendible to a set of machines.

Definition 4.6 *(Port Set) The* port set $\mathsf{ports}(\mathsf{M})$ *of a machine* M *is the set of ports in the sequence* $Ports_{\mathsf{M}}$*. For a set* $\hat{M}$ *of machines, let* $\mathsf{ports}(\hat{M}) := \bigcup_{\mathsf{M}\in\hat{M}} \mathsf{ports}(\mathsf{M})$. ◇

In the following, we define three disjoint sets of machines. Membership in each set can be determined by the name and the ports of a machine. These sets do not serve as a partition of the set of all machines, but only machines contained in these sets matter in the following. This will become clear when we introduce how several machines interact and schedule each other.

Simple machines only have simple ports and clock out-ports, and their names are contained in Σ^+. We do not make any restrictions on their internal behavior.

Definition 4.7 *(Simple Machines) A machine* M *is* simple *iff* $name_{\mathsf{M}} \in \Sigma^+$ *and for all* $p = (\mathsf{n}, l, d) \in \mathsf{ports}(\mathsf{M})$ *we have* $l = \epsilon$ *or* $(l, d) = (^{\triangleleft}, !)$. ◇

Similar to simple machines, *master schedulers* only have simple ports and clock out-ports, except that they have one special clock in-port $\mathsf{clk}^{\triangleleft}?$, called the *master-clock* in-port. When we define the interaction of several machines, this port will be used to resolve situations where the interaction cannot proceed otherwise. A master scheduler must not make non-empty outputs in a transition that enters a final state. This will simplify the later definition that the entire interaction between machines stops if a master scheduler enters a final state.

Definition 4.8 *(Master Schedulers) A machine* M *is a* master scheduler *iff*

- $name_{\mathsf{M}} \in \Sigma^+$,
- $\mathsf{clk}^{\triangleleft}? \in \mathsf{ports}(\mathsf{M})$,
- *for all* $p = (\mathsf{n}, l, d) \in \mathsf{ports}(\mathsf{M}) \setminus \{\mathsf{clk}^{\triangleleft}?\}$*, we have* $l = \epsilon$ *or* $(l, d) = (^{\triangleleft}, !)$*, and*
- *if* $\mathsf{Pr}(\delta_{\mathsf{M}}(s, I) = (s', O)) > 0$ *with* $s' \in Fin_{\mathsf{M}}$ *and arbitrary* $s \in States_{\mathsf{M}}$ *and* $I \in \mathcal{I}_{\mathsf{M}}$*, then* $O = (\epsilon, \ldots, \epsilon)$.

◇

If a simple machine or a master scheduler M has an out-port $\mathsf{n}!$ or an in-port $\mathsf{n}?$ we say that M is the *sending machine* or *receiving machine* for n, as shown in Figure 2.

As the third machine set, we define *buffers*. All buffers have the same predefined transition function. They model the asynchronous channels between other machines, and will later be inserted between two ports $\mathsf{n}!$ and $\mathsf{n}?$ as shown in Figure 2. More precisely, for each port name

n, we define a buffer denoted as $\widetilde{\mathsf{n}}$ with three ports $\mathsf{n}^{\triangleleft}?, \mathsf{n}^{\leftrightarrow}?$, and $\mathsf{n}^{\leftrightarrow}!$. When a value is input at $\mathsf{n}^{\leftrightarrow}?$, the transition function of the buffer appends this value to an internal queue over Σ^*. An input $u \neq \epsilon$ at $\mathsf{n}^{\triangleleft}?$ is interpreted as a natural number, captured by the function nat, and the $\mathsf{nat}(u)$-th element of the internal queue is removed and output at $\mathsf{n}^{\leftrightarrow}!$. If there are less than $\mathsf{nat}(u)$ elements, the output is ϵ. As the two inputs never occur together in the upcoming run algorithm, we define that the buffer only evaluates its first non-empty input. Since the states have to be a subset of Σ^* by Definition 4.5, we embed the queue into Σ^* using the embedding function ι for sequences (see Section 3).

Definition 4.9 *(Buffers) For every $\mathsf{n} \in \Sigma^+$ we define a machine $\widetilde{\mathsf{n}}$ called a* buffer*:*

$$\widetilde{\mathsf{n}} := (\mathsf{n}\sim, (\mathsf{n}^{\triangleleft}?, \mathsf{n}^{\leftrightarrow}?, \mathsf{n}^{\leftrightarrow}!), States_{\widetilde{\mathsf{n}}}, \delta_{\widetilde{\mathsf{n}}}, l_{\widetilde{\mathsf{n}}}, Ini_{\widetilde{\mathsf{n}}}, \emptyset)$$

with

- $States_{\widetilde{\mathsf{n}}} := \{\iota(P) \mid P \in (\Sigma^*)^*\}$,
- $Ini_{\widetilde{\mathsf{n}}} := \{\iota(\epsilon)\}$,
- $l_{\widetilde{\mathsf{n}}}(\iota(P)) := (\infty, \infty)$ *for all* $\iota(P) \in States_{\widetilde{\mathsf{n}}}$*, and*
- $\delta_{\widetilde{\mathsf{n}}}(\iota(P), (u, v)) := (\iota(P'), (o))$ *deterministically as follows for* $(P_1, \ldots, P_{\mathsf{size}(P)}) := P$*:*

  if $u \neq \epsilon$ ***then***
    if $\mathsf{nat}(u) \leq \mathsf{size}(P)$ ***then***
      $P' := (P_i \in P \mid i \neq \mathsf{nat}(u))$
      $o := P_{\mathsf{nat}(u)}$
    else
      $P' := P$ *and* $o = \epsilon$
    end if
  else if $v \neq \epsilon$ ***then***
    $P' := P \circ (v)$ *and* $o := \epsilon$
  else
    $P' := P$ *and* $o := \epsilon$
  end if

$\diamond$

In the following, a machine with a tilde such as $\widetilde{\mathsf{n}}$ always means the unique buffer for $\mathsf{n} \in \Sigma^+$ according to Definition 4.9.

4.3 Computational Realization

For computational aspects, a machine M is regarded as implemented by a probabilistic interactive Turing machine as introduced in [31]. We need some refinements of this model. The main feature of interactive Turing machines is that they have communication tapes where one machine can write and one other machine can read. Thus we will use one communication tape to model each low-level connection. Probabilism is modeled by giving each Turing machine a read-only random tape containing an infinite sequence of independent, uniformly random bits. To make each Turing configuration finite, we can instead newly choose such a bit whenever a

cell of the random tape is first accessed. Each Turing machine has one distinguished work tape; it may or may not have further local tapes, which are initially empty.

Our first refinement concerns how the heads move on communication tapes; our choice guarantees that a machine can ignore the ends of long messages as defined by the length functions in our I/O machine model, and nevertheless read the following message. This helps machines to guarantee certain runtimes without becoming vulnerable to denial-of-service attacks by an adversary sending a message longer than this runtime. This enables liveness properties, although those are not considered in this paper. The second refinement concerns restarts of machines in a multi-machine scenario. We guarantee that a switching step with only empty inputs is equivalent to no step at all, as in the I/O machine model.

By a Turing machine whose heads recognize partner heads we mean a Turing machine whose transition function is based on its finite state and, for each of its heads, on the content of the cell under this head and a bit denoting whether another head is on the same cell.

Definition 4.10 *(Computational Realization of Machines) A* probabilistic interactive Turing machine T *is a probabilistic multi-tape Turing machine whose heads recognize partner heads. Tapes have a left boundary, and heads start on the left-most cell.* T implements *a machine* M *as defined in Definition 4.5 if the following holds. Let* $i_{\mathsf{M}} := |\mathsf{in}(Ports_{\mathsf{M}})|$. *We write "finite state" for a state of the finite control of* T *and "*M*-state" for an element of* $States_{\mathsf{M}}$.

a) T *has a read-only tape for each in-port of* M. *Here the head never moves left, nor to the right of the other head on that tape. For each out-port of* M, T *has a write-only tape where the head never moves left of the other head on that tape.*

b) T *has special finite states* $\mathsf{restart}_{int}$ *(where "int" is similar to an interrupt vector) with* $int \in \mathcal{P}(\{1, \ldots, i_{\mathsf{M}}\})$ *for waking up asynchronously with inputs at a certain set of ports,* sleep *denoting the end of an* M*-transition, and* end *for termination. Here* $\mathsf{restart}_{\emptyset} = \mathsf{sleep}$, *i.e.,* T *needs no time for "empty" transitions.*

c) T *realizes* $\delta_{\mathsf{M}}(s, I)$ *as follows for all* $s \in States_{\mathsf{M}}$ *and* $I \in \mathcal{I}_{\mathsf{M}}$*: Let* T *start in finite state* $\mathsf{restart}_{int}$ *where* $int := \{i \mid I_i\lceil_{l_{\mathsf{M}}(s)_i} \neq \epsilon\} \neq \emptyset$, *with worktape content* s, *and with* I_i *on the* i*-th input tape from (including)* T*'s head to (excluding) the other head on this tape for all* i. *Let* s' *be the worktape content in the next finite state* sleep *or* end, *and* O_i *the content of the* i*-th output tape from (including) the other head to (excluding)* T*'s head in that state. Then the pairs* (s', O) *are distributed according to* $\delta_{\mathsf{M}}(s, I)$, *and the finite state is* end *iff* $s' \in Fin_{\mathsf{M}}$.

$\diamond$

The main reason to introduce a Turing-machine realization of the machine model is to define complexity notions. The interesting question is how we handle the inputs on communication tapes in the complexity definition, in particular for the notion of polynomial time, which is the maximum complexity allowed to adversaries against typical cryptographic systems.

One can imagine three degrees of an interactive machine being polynomial-time. The weakest would be that each M-transition only needs time polynomial in the current inputs and the current state, i.e., the current content of the local tapes. However, such a machine might double the size of its current state in each M-transition; then it would be allowed time exponential in an initial security parameter after a linear number of M-transitions. Hence we do not use this notion.

In the medium notion, which we call *weakly polynomial-time*, the runtime of the machine is polynomial in the overall length of its inputs, including the initial worktape content. Equivalently, the runtime for each M-transition is polynomial in the overall length of the inputs received so far. This makes the machine a permissible adversary when interacting with a cryptographic system which is in itself polynomially bounded. However, several weakly polynomial-time machines together (or even one with a self-connection) can become too powerful. E.g., each new output may be twice as long as the inputs so far. Then after a linear number of M-transitions, these weakly polynomial-time machines are allowed time exponential in an initial security parameter. We nevertheless use weakly polynomial-time machines sometimes, because many functionalities are naturally weakly polynomial-time and not naturally polynomial-time in the following strong sense. However, one always has to keep in mind that this notion does not compose as we just saw.

Finally, polynomial-time machines are those who only need time polynomial in their initial worktape content, independent of all inputs on communication tapes.

A run of a probabilistic, interactive Turing machine is a valid sequence of configurations of T (defined as for other Turing machines), where the finite state end can only occur in the last configuration of the run.

Definition 4.11 *(Complexity of Machines)*

a) *A probabilistic interactive Turing machine* T *is* polynomial-time *iff there exists a polynomial P such that all possible runs of* T *are of length at most $P(k)$, where k is the length of the initial worktape content.*

b) T *is called weakly polynomial-time iff there exists a polynomial P such that for every fixed initial worktape content and fixed contents of all input tapes with overall length k', all possible runs of* T *are of length at most $P(k')$.*

c) *A machine* M *according to Definition 4.5 is (weakly) polynomial-time iff it has a realization as a (weakly) polynomial-time probabilistic interactive Turing machine.*

More generally, if we say without further qualification that T *fulfills some complexity measure, we mean that all possible runs of the machine fulfill this measure as a function of the length k of the initial worktape content.* ◇

Besides the deterministic runtime bounds that we have defined and that we will use in the following, one could define bounds for the expected runtime.

4.4 Collections of Machines

After introducing individual machines, we now focus on *collections* of finitely many machines, with the intuition that these machines interact. Each machine in a collection must be uniquely determined by its name, and their port sets must be pairwise disjoint so that the naming conventions for low- and high-level complements will lead to well-defined one-to-one connections.

Definition 4.12 *(Collections)*

a) *A* collection $\hat{C}$ *is a finite set of machines with pairwise different machine names, pairwise disjoint port sets, and where each machine is a simple machine, a master scheduler, or a buffer.*

b) A collection is called (weakly) polynomial-time *iff all its non-buffer machines are (weakly) polynomial-time.*

c) If $\tilde{\mathsf{n}}, \mathsf{M} \in \hat{C}$ *and* $\mathsf{n}^{\triangleleft}! \in \mathsf{ports}(\mathsf{M})$ *then we call* M *the* scheduler for *buffer* $\tilde{\mathsf{n}}$ *in* $\hat{C}$*, and we omit "in* $\hat{C}$*" if it is clear from the context.*

◇

If a port and its low-level complement are both contained in the port set of the collection, they form a *low-level connection*; recall Definition 4.4. *High-level connections* for simple ports are defined similarly. If a port p is contained in the port set of the collection but its low-level complement is not, p is called *free*. Free ports will later be used to connect external, machines to the collection. For instance, a collection may consist of machines that execute a cryptographic protocol, and their free ports can be connected to users and an adversary.

Definition 4.13 *(Connections) Let* $\hat{C}$ *be a collection.*

a) If $p, p^c \in \mathsf{ports}(\hat{C})$ *then* $\{p, p^c\}$ *is called a* low-level connection*. The set* $\mathsf{gr}(\hat{C}) := \{\{p, p^c\} \mid p, p^c \in \mathsf{ports}(\hat{C})\}$ *is called the* low-level connection graph *of* $\hat{C}$*.*

b) By $\mathsf{free}(\hat{C}) := \mathsf{ports}(\hat{C}) \setminus \mathsf{ports}(\hat{C})^c$ *we denote the* free *ports in* $\hat{C}$*.*

c) If $p, p^C \in \mathsf{ports}(\hat{C})$ *then* $\{p, p^C\}$ *is called a* high-level connection*. The set* $\mathsf{Gr}(\hat{C}) := \{\{p, p^C\} \mid p, p^C \in \mathsf{ports}(\hat{C})\}$ *is called the* high-level connection graph *of* $\hat{C}$*.*

◇

Given a collection of (usually simple) machines, we want to add buffers for all high-level connections to model asynchronous timing. This is modeled by the *completion* of a collection $\hat{C}$. The completion is the union of $\hat{C}$ and buffers for all existing ports except the master-clock in-port. Note that completion leaves already existing buffers in $\hat{C}$ unchanged. A collection is called *closed* if the only free port of its completion is the master-clock in-port $\mathsf{clk}^{\triangleleft}?$. This implies that a closed collection has precisely one master scheduler, identified by having the unique master-clock in-port.

Definition 4.14 *(Closed Collection, Completion) Let* $\hat{C}$ *be a collection.*

a) The completion $[\hat{C}]$ *of* $\hat{C}$ *is defined as*

$$[\hat{C}] := \hat{C} \cup \{\tilde{\mathsf{n}} \mid \exists l, d\colon (\mathsf{n}, l, d) \in \mathsf{ports}(\hat{C}) \setminus \{\mathsf{clk}^{\triangleleft}?\}\}.$$

b) $\hat{C}$ *is* closed *iff* $\mathsf{free}([\hat{C}]) = \{\mathsf{clk}^{\triangleleft}?\}$*, and* $\hat{C}$ *is* complete *iff* $[\hat{C}] = \hat{C}$*.*

◇

4.5 Runs and their Probability Spaces

For a closed collection, we now define *runs* (in other terminologies executions or traces).

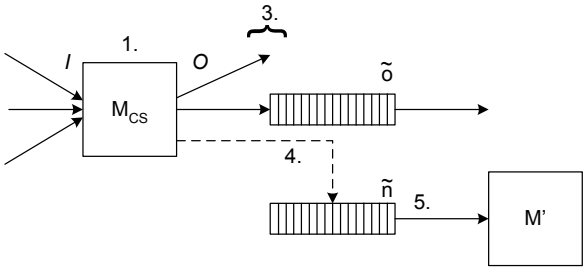

Figure 3: Phases of the run algorithm.

Informal Description

We start with an informal description. Since the collection is closed, it contains a unique master scheduler X. Machines switch sequentially, i.e., we have exactly one active machine M at any time. If this machine has clock out-ports, it can select the next message to be delivered by scheduling a buffer via one of these clock out-ports. If a message exists at the respective position of the buffer's internal queue, 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 master scheduler X becomes active.

Next we give a more precise, but still only semi-formal definition of runs. Runs and their probability spaces are defined inductively by the following algorithm for each tuple $ini \in \times_{\mathsf{M}\in\hat{C}} Ini_{\mathsf{M}}$ of initial states of the machines of a collection $\hat{C}$. The algorithm maintains variables for the states of all machines of the collection and treats each port as a variable over Σ^*, initialized with ϵ except for $\mathsf{clk}^{\triangleleft}? := 1$. The algorithm further maintains a variable $\mathsf{M_{CS}}$ ("current scheduler") over machine names, initialized with $\mathsf{M_{CS}} := \mathsf{X}$, for the name of the currently active simple machine or master scheduler, and a variable r for the resulting run, an initially empty list. The algorithm operates in five phases, which are illustrated in Figure 3. Probabilistic choices only occur in Phase 1.

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

2. *Termination:* If X is in a final state, the run stops. (As X made no outputs in this case, this only prevents repeated master clock inputs.)

3. *Store outputs:* For each simple out-port $\mathsf{o}!$ of $\mathsf{M_{CS}}$ with $\mathsf{o}! \neq \epsilon$, in their given order, switch buffer $\tilde{\mathsf{o}}$ with input $\mathsf{o}^{\leftrightarrow}? := \mathsf{o}!$. Then assign ϵ to all these ports $\mathsf{o}!$ and $\mathsf{o}^{\leftrightarrow}?$.

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

5. *Deliver scheduled message:* Switch $\tilde{\mathsf{n}}$ with input $\mathsf{n}^{\triangleleft}? := \mathsf{n}^{\triangleleft}!$, set $\mathsf{n}? := \mathsf{n}^{\leftrightarrow}!$ and then assign ϵ to all ports of $\tilde{\mathsf{n}}$ and to $\mathsf{n}^{\triangleleft}!$. If $\mathsf{n}? = \epsilon$ let $\mathsf{clk}^{\triangleleft}? := 1$ and $\mathsf{M_{CS}} := \mathsf{X}$. Else let $\mathsf{M_{CS}} := \mathsf{M}'$ for the unique machine M' with $\mathsf{n}? \in \mathsf{ports}(\mathsf{M}')$. Go to Phase 1.

Whenever a machine (this may be a buffer) with name $name_{\mathsf{M}}$ is switched from (s, I) to (s', O), we add a *step* $(name_{\mathsf{M}}, s, I, s', O)$ to the run r with the following two restrictions. First, we cut each input according to the respective length function, i.e., we replace I by $I' := I\lceil_{l_{\mathsf{M}}(s)}$.

Secondly, we do not add the step to the run if $I' = (\epsilon, \ldots, \epsilon)$, i.e., if nothing happens in reality. This gives a family of probability distributions $(run_{\hat{C},ini})$, one for each tuple *ini* of initial states of the machines of the collection. Moreover, for a set $\hat{M}$ of machines, we define the restriction of runs to those steps where a machine of $\hat{M}$ switches. This is called the *view* of $\hat{M}$. Similar to runs, this gives a family of probability distributions $(view_{\hat{C},ini}(\hat{M}))$, one for each tuple *ini* of initial states.

Rigorous Definitions

We now define the probability space of runs rigorously. Since the semi-formal description is sufficient to understand our subsequent results and the rigorous definitions are quite technical, this subsection can be skipped at first reading.

We first define a *global state space* and a *global transition function* on these states. The global state space has five parts: the states of all machines of the collection, the current scheduler (currently active machine), a function assigning strings, the current values, to the ports of the collection, the current phase, and a subset of the out-port sequence of one machine for modeling the "for"-loop in the third phase of the informal algorithm. Additionally, the global state space contains a distinguished global final state s_{fin}.

Definition 4.15 *(Global States of a Collection) Let $\hat{C}$ be a complete, closed collection with master scheduler X. Let $\mathcal{P}_{\hat{C}} := \{P \mid \exists \mathsf{M} \in \hat{C} \colon P \subseteq \mathsf{out}(Ports_{\mathsf{M}})\}$ where $\subseteq$ denotes the subsequence relation.*

- *The* set of global states *of $\hat{C}$ is defined as*

$$States_{\hat{C}} := \times_{\mathsf{M} \in \hat{C}} States_{\mathsf{M}} \times \hat{C} \times (\Sigma^*)^{\mathsf{ports}(\hat{C})} \times \{1, \ldots, 5\} \times \mathcal{P}_{\hat{C}} \cup \{s_{fin}\}.$$

- *The* set of initial global states *of $\hat{C}$ is defined as*

$$Ini_{\hat{C}} := \times_{\mathsf{M} \in \hat{C}} Ini_{\mathsf{M}} \times \{\mathsf{X}\} \times \{f\} \times \{1\} \times \{()\}$$

  with $f(\mathsf{clk}^{\triangleleft}?) := 1$ and $f(p) := \epsilon$ for $p \in \mathsf{ports}(\hat{C}) \setminus \{\mathsf{clk}^{\triangleleft}?\}$.

$\diamond$

On these global states, we define a global transition function. It reflects the informal run algorithm.

Definition 4.16 *(Global Transition Function) Let $\hat{C}$ be a complete, closed collection with master scheduler X. We define the* global transition function

$$\delta_{\hat{C}} \colon States_{\hat{C}} \to \mathsf{Prob}(States_{\hat{C}})$$

by $\delta_{\hat{C}}(s_{fin}) := s_{fin}$ and otherwise by the following rules:

Phase 1: Switch current scheduler.

$$((s_{\mathsf{M}})_{\mathsf{M} \in \hat{C}}, \mathsf{M}_{\mathsf{CS}}, f, 1, P) \to_p ((s'_{\mathsf{M}})_{\mathsf{M} \in \hat{C}}, \mathsf{M}_{\mathsf{CS}}, f', 2, ()) \tag{1}$$

where, with $I := f(\mathsf{in}(Ports_{\mathsf{M}_{\mathsf{CS}}}))$ and $O := f'(\mathsf{out}(Ports_{\mathsf{M}_{\mathsf{CS}}}))$,

- $p = \mathsf{Pr}(\delta_{\mathsf{M_{CS}}}(s_{\mathsf{M_{CS}}}, I) = (s'_{\mathsf{M_{CS}}}, O))$,
- $s_{\mathsf{M}} = s'_{\mathsf{M}}$ *for all* $\mathsf{M} \in \hat{C} \setminus \{\mathsf{M_{CS}}\}$, *and*
- $f'(\mathsf{in}(Ports_{\mathsf{M_{CS}}})) = (\epsilon)^{|\mathsf{in}(Ports_{\mathsf{M_{CS}}})|}$ *and* $f \equiv f'$ *on* $\mathsf{ports}(\hat{C}) \setminus \mathsf{ports}(\mathsf{M_{CS}})$.

Phase 2: Termination.

$$((s_{\mathsf{M}})_{\mathsf{M}\in\hat{C}}, \mathsf{M_{CS}}, f, 2, P) \to s_{fin} \qquad \text{if } s_{\mathsf{X}} \in Fin_{\mathsf{X}}; \tag{2}$$
$$((s_{\mathsf{M}})_{\mathsf{M}\in\hat{C}}, \mathsf{M_{CS}}, f, 2, P) \to ((s_{\mathsf{M}})_{\mathsf{M}\in\hat{C}}, \mathsf{M_{CS}}, f, 3, P') \qquad \text{if } s_{\mathsf{X}} \notin Fin_{\mathsf{X}} \tag{3}$$

where $P' = (\mathsf{o}! \in Ports_{\mathsf{M_{CS}}} \mid \mathsf{o} \in \Sigma^+ \wedge f(\mathsf{o}!) \neq \epsilon)$.

Phase 3: Store outputs.

$$((s_{\mathsf{M}})_{\mathsf{M}\in\hat{C}}, \mathsf{M_{CS}}, f, 3, ()) \to ((s_{\mathsf{M}})_{\mathsf{M}\in\hat{C}}, \mathsf{M_{CS}}, f, 4, ()); \tag{4}$$
$$((s_{\mathsf{M}})_{\mathsf{M}\in\hat{C}}, \mathsf{M_{CS}}, f, 3, P) \to ((s'_{\mathsf{M}})_{\mathsf{M}\in\hat{C}}, \mathsf{M_{CS}}, f', 3, P') \qquad \text{if } P \neq () \tag{5}$$

where there exists $\mathsf{n} \in \Sigma^+$ *with*

- $P = (\mathsf{n}!) \circ P'$,
- $(s'_{\widetilde{\mathsf{n}}}, (\epsilon)) = \delta_{\widetilde{\mathsf{n}}}(s_{\widetilde{\mathsf{n}}}, (\epsilon, f(\mathsf{n}!)))$,
- $s_{\mathsf{M}} = s'_{\mathsf{M}}$ *for all* $\mathsf{M} \in \hat{C} \setminus \{\widetilde{\mathsf{n}}\}$, *and*
- $f'(\mathsf{n}!) = \epsilon$ *and* $f \equiv f'$ *on* $\mathsf{ports}(\hat{C}) \setminus \{\mathsf{n}!\}$.

Phase 4: Clean up scheduling. *Let* $Clks := (\mathsf{n}^{\triangleleft}! \in Ports_{\mathsf{M_{CS}}} \mid f(\mathsf{n}^{\triangleleft}!) \neq \epsilon)$. *Then*

$$((s_{\mathsf{M}})_{\mathsf{M}\in\hat{C}}, \mathsf{M_{CS}}, f, 4, P) \to ((s_{\mathsf{M}})_{\mathsf{M}\in\hat{C}}, \mathsf{X}, f', 1, ()) \qquad \text{if } Clks = () \tag{6}$$

where $f'(p) = \epsilon$ *for all* $p \in \mathsf{ports}(\hat{C}) \setminus \{\mathsf{clk}^{\triangleleft}?\}$ *and* $f'(\mathsf{clk}^{\triangleleft}?) = 1$, *and*

$$((s_{\mathsf{M}})_{\mathsf{M}\in\hat{C}}, \mathsf{M_{CS}}, f, 4, P) \to ((s_{\mathsf{M}})_{\mathsf{M}\in\hat{C}}, \mathsf{M_{CS}}, f', 5, P') \qquad \text{if } Clks \neq () \tag{7}$$

where

- $P' = (Clks[1])$, *and*
- $f'(Clks[1]^c) = f(Clks[1])$ *and* $f'(p) = \epsilon$ *for all* $p \in \mathsf{ports}(\hat{C}) \setminus \{Clks[1]^c\}$.

Phase 5: Deliver scheduled message.

$$((s_{\mathsf{M}})_{\mathsf{M}\in\hat{C}}, \mathsf{M_{CS}}, f, 5, P) \to s_{fin} \qquad \text{if } \nexists \mathsf{n} \in \Sigma^+ \colon P = (\mathsf{n}^{\triangleleft}!); \tag{8}$$
$$((s_{\mathsf{M}})_{\mathsf{M}\in\hat{C}}, \mathsf{M_{CS}}, f, 5, (\mathsf{n}^{\triangleleft}!)) \to ((s'_{\mathsf{M}})_{\mathsf{M}\in\hat{C}}, \mathsf{M'_{CS}}, f', 1, ()) \tag{9}$$

where, with $u := f(\mathsf{n}^{\triangleleft}?)$, *there exists* $o \in \Sigma^+$ *such that*

- $s_{\mathsf{M}} = s'_{\mathsf{M}}$ *for all* $\mathsf{M} \in \hat{C} \setminus \{\widetilde{\mathsf{n}}\}$,
- $(s'_{\widetilde{\mathsf{n}}}, (o)) = \delta_{\widetilde{\mathsf{n}}}(s_{\widetilde{\mathsf{n}}}, (u, \epsilon))$,

and

- *either* $o = \epsilon$ *and* $\mathsf{M'_{CS}} = \mathsf{X}$ *and* $f'(\mathsf{clk}^{\triangleleft}?) = 1$ *and* $f'(p) = \epsilon$ *for all* $p \in \mathsf{ports}(\hat{C}) \setminus \{\mathsf{clk}^{\triangleleft}?\}$

- *or* $o \neq \epsilon$ *and* $\mathsf{n}? \in Ports_{\mathsf{M}'_{\mathsf{CS}}}$ *and* $f'(\mathsf{n}?) = o$ *and* $f'(p) = \epsilon$ *for all* $p \in \mathsf{ports}(\hat{C}) \setminus \{\mathsf{n}?\}$.

$\diamond$

Rule (8) has only been included to define the function δ on the entire state space $States_{\hat{C}}$ as claimed at the beginning of the definition. It will not matter in the execution since the previous state has to be in Phase 4, and this ensures that P contains exactly one clock port. Similarly, the other rules make no assumptions about reachable states. Further, $\delta(s)$ is indeed an element of $\mathsf{Prob}(States_{\hat{C}})$ for every $s \in States_{\hat{C}}$: It is deterministic for all states s except those treated in Rule (1). For those, the claim follows immediately from the fact that $\delta_{\mathsf{M}_{\mathsf{CS}}}(s_{\mathsf{M}_{\mathsf{CS}}}, I)$ is a finite distribution.

Given the global probabilistic transition function δ, we now obtain probability distributions on sequences of global states by canonical constructions as for Markov chains. This even holds for infinite sequences by the theorem of Ionescu-Tulcea; see, e.g., Section V.1 of [50]. More precisely, we obtain one such probability distribution for every global initial state. Applying the theorem of Ionescu-Tulcea to our situation yields the following lemma.

Lemma 4.1 (Probabilities of State Sequences) *Let $\hat{C}$ be a complete, closed collection, and let an initial global state $ini \in Ini_{\hat{C}}$ be given. For each set of fixed-length sequences $States^i_{\hat{C}}$ with $i \in \mathbb{N}$, we can define a finite probability distribution $PStates_{\hat{C},ini,i}$ by*

$$\mathsf{Pr}(S) = \prod_{j=2}^{i} \mathsf{Pr}(\delta_{\hat{C}}(S_{j-1}) = S_j)$$

for every sequence $S = (S_1, \ldots, S_i)$ over $States_{\hat{C}}$ with $S_1 = ini$, and $\mathsf{Pr}(S) = 0$ otherwise.

Further, for every sequence $S \in States^i_{\hat{C}}$, let $\mathsf{Rect}(S)$ denote the corresponding rectangle *of infinite sequences with this prefix, i.e., $\mathsf{Rect}(S) := \{S' \in States^\infty_{\hat{C}} \mid S' \lceil_i = S\}$. Then there exists a unique probability distribution $PStates_{\hat{C},ini,\infty}$ over $States^\infty_{\hat{C}}$ whose value for every rectangle $R := \mathsf{Rect}(S)$ with $S \in States^i_{\hat{C}}$ equals $\mathsf{Pr}(S)$, or more precisely,*

$$\mathsf{Pr}_{PStates_{\hat{C},ini,\infty}}(R) := \mathsf{Pr}_{PStates_{\hat{C},ini,i}}(S).$$

We usually omit the indices "∞" and "i" of these distributions; this cannot lead to confusion.
□

Varying *ini* gives a family of probability distributions over $States^\infty_{\hat{C}}$; we write it

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

So far we have defined probabilities for sequences of entire global states. Each step in the runs introduced semi-formally above intuitively corresponds to the difference between two successive states: Moreover, only the switching of machines is considered in a run, i.e., the intermediate phases for termination checks and cleaning up scheduling are omitted, as well as the switching of machines with empty input (after application of the length function) because nothing happens then. We first define the set of possible steps, i.e., five-tuples containing the name of the currently switched machine M, its old state, its input tuple, its new state, and its output tuple. Further, we define an encoding of steps as strings for the sole purpose of defining overall lengths of potential runs.

Definition 4.17 *(Steps) The set of* steps *is defined as*

$$\mathit{Steps} := (\Sigma^+ \circ \{\sim, \epsilon\}) \times \Sigma^* \times (\Sigma^*)^* \times \Sigma^* \times (\Sigma^*)^*.$$

Let $\iota_{\mathit{Steps}} \colon \mathit{Steps} \to \Sigma^+$ *denote a bijective function such that standard operations on Steps are efficiently computable in this encoding. For all* $r = (r_i)_{i \in \mathcal{I}} \in \mathit{Steps}^*$ *let* $\mathsf{len}(r) := \sum_{i \in \mathcal{I}} \mathsf{size}(\iota_{\mathit{Steps}}(r))$*, and for* $r \in \mathit{Steps}^\infty$ *let* $\mathsf{len}(r) := \infty$*.* ◇

Now we define a mapping that extracts a run from a step sequence. Our definition first extracts a sequence of one potential step from each pair of a global state and the next one. This first part maps intermediate phases to steps with empty input tuples, so that it is then sufficient to restrict the obtained sequence to the elements with a non-empty input tuple for getting rid of both the intermediate phases and of actual switching with empty inputs. When extracting a potential step, we mainly distinguish whether the current scheduler switches or a buffer; such a buffer is always indicated by the first element in the port sequence of outputs still to be handled, the fifth element of the global state.

Definition 4.18 *(Run Extraction) Let* $\hat{C}$ *be a complete, closed collection. Then the* run extraction *of* $\hat{C}$ *is the function*

$$\mathit{run}_{\hat{C}} \colon \mathit{States}^\infty_{\hat{C}} \to \mathit{Steps}^* \cup \mathit{Steps}^\infty$$

defined as follows. (It is independent of $\hat{C}$ *except for its domain.) Let* $S = (S_i)_{i \in \mathbb{N}} \in \mathit{States}^\infty_{\hat{C}}$ *with* $S_i = s_{\mathit{fin}}$ *or* $S_i = ((s^i_{\mathsf{M}})_{\mathsf{M} \in \hat{C}}, \mathsf{M}^i_{\mathsf{CS}}, f^i, j^i, P^i)$ *for all* $i \in \mathbb{N}$*. If* $i^* := \min\{i \in \mathbb{N} \mid s_i = s_{\mathit{fin}}\}$ *exists, let* $\mathcal{I} := \{1, \ldots, i^* - 2\}$*, otherwise* $\mathcal{I} := \mathbb{N}$*. We define a sequence* $\mathit{pot\_step}(S) = ((\mathsf{n}_i, s_i, I_i, s'_i, O_i))_{i \in \mathcal{I}}$ *as follows. (Thus we already omit the last termination check.)*

- *If* $P^i = ()$ *then, with* $\mathsf{M} := \mathsf{M}^i_{\mathsf{CS}}$*,*
  - $\mathsf{n}_i := \mathit{name}_{\mathsf{M}}$*,*
  - $s_i := s^i_{\mathsf{M}}$ *and* $s'_i := s^{i+1}_{\mathsf{M}}$*, and*
  - *if* $j^i = 1$ *then* $I_i := f^i(\mathsf{in}(\mathit{Ports}_{\mathsf{M}}))\lceil_{l_{\mathsf{M}}(s_i)}$ *and* $O_i := f^{i+1}(\mathsf{out}(\mathit{Ports}_{\mathsf{M}}))$*, else* $I_i := (\epsilon)$ *and* $O_i := (\epsilon)$*.*
- *If* $P^i \neq ()$*, let* $p := P^i[1]$ *and* $p \in \mathsf{ports}(\widetilde{\mathsf{n}})$*. Then*
  - $\mathsf{n}_i := \mathsf{n}{\sim}$*.*
  - $s_i := s^i_{\widetilde{\mathsf{n}}}$ *and* $s'_i := s^{i+1}_{\widetilde{\mathsf{n}}}$*,*
  - $I_i := f^i((\mathsf{n}^{\triangleleft}?, \mathsf{n}^{\leftrightarrow}?))$ *and* $O_i := f^{i+1}((\mathsf{n}^{\leftrightarrow}!))$*.*

Then $\mathit{run}_{\hat{C}}(S) := ((\mathsf{n}_i, s_i, I_i, s'_i, O_i) \in \mathit{pot\_step}(S) \mid I_i \neq (\epsilon, \ldots, \epsilon))$*. For every number* $l \in \mathbb{N}$*, let* $\mathit{run}_{\hat{C},l}$ *denote the extraction of* l*-step prefixes of runs,*

$$\mathit{run}_{\hat{C},l} \colon \mathit{States}^\infty_{\hat{C}} \to \bigcup_{i \leq l} \mathit{Steps}^i$$

with $\mathit{run}_{\hat{C},l}(S) := \mathit{run}_{\hat{C}}(S)\lceil_l$*.* ◇

The run extraction is a random variable on every probability space over $States^{\infty}_{\hat{C}}$. For our particular probability space, this induces a family $run_{\hat{C}} = (run_{\hat{C},ini})_{ini \in Ini_{\hat{C}}}$ of probability distributions over $Steps^* \cup Steps^{\infty}$ via

$$\mathsf{Pr}_{run_{\hat{C},ini}}(r) := \mathsf{Pr}_{PStates_{\hat{C},ini}}(run^{-1}_{\hat{C}}(r))$$

for all $r \in Steps^* \cup Steps^{\infty}$, where $run^{-1}_{\hat{C}}(r)$ is the set of pre-images of r. For a function $l\colon Ini_{\hat{C}} \to \mathbb{N}$, this similarly gives a family of probability distributions

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

each over $\bigcup_{i \leq l(ini)} Steps^i$.

We finally introduce the *view* of a set $\hat{M}$ of machines in a collection. It is the restriction of a run (a sequence of steps) to those steps whose machine name belongs to $\hat{M}$. The extraction function is independent of the collection, but we index it with $\hat{C}$ anyway for similarity with the run notation.

Definition 4.19 *(Views) Let $\hat{C}$ be a complete, closed collection and $\hat{M} \subseteq \hat{C}$. The* view *of $\hat{M}$ in $\hat{C}$ is the function $view_{\hat{C}}(\hat{M})\colon Steps^* \cup Steps^{\infty} \to Steps^* \cup Steps^{\infty}$ with*

$$view_{\hat{C}}(\hat{M})(r) := (s_i \in r \mid \exists \mathsf{M} \in \hat{M} : s_i[1] = name_{\mathsf{M}}).$$

For every number $l \in \mathbb{N}$ let $view_{\hat{C},l}(\hat{M})$ denote the extraction of l-step prefixes of a view, i.e., $view_{\hat{C},l}(\hat{M})\colon Steps^* \cup Steps^{\infty} \to \bigcup_{i \leq l} Steps^i$ with $view_{\hat{C},l}(\hat{M})(r) := view_{\hat{C}}(\hat{M})(r)\lceil_l$. ◇

For a singleton $\hat{M} = \{\mathsf{M}\}$, we write $view_{\hat{C}}(\mathsf{M})$ instead of $view_{\hat{C}}(\{\hat{M}\})$, and similar for l-step prefixes. Based on the family $run_{\hat{C}}$ of probability distributions of runs, this induces a family of probability distributions

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

over $Steps^* \cup Steps^{\infty}$. For a function $l\colon Ini_{\hat{C}} \to \mathbb{N}$, this similarly gives a family $view_{\hat{C},l} = (view_{\hat{C},ini,l(ini)})_{ini \in Ini_{\hat{C}}}$ over $\bigcup_{i \leq l(ini)} Steps^i$.

Finally, we define sets of *state-traces* and of *traces* (or step-traces). Intuitively, they are the possible sequences of global states and of steps, respectively. More precisely, each finite prefix of such a sequence happens with positive probability.

Definition 4.20 *(State-Trace, Trace) Let $\hat{C}$ be a complete, closed collection and $ini \in Ini_{\hat{C}}$ an initial global state.*

- *The set of* state-traces *for ini is*

$$StateTrace_{\hat{C},ini} := \{S \in States^{\infty}_{\hat{C}} \mid \forall l \in \mathbb{N}\colon \mathsf{Pr}_{PStates_{\hat{C},ini,l}}(S\lceil_l) > 0\}.$$

- *The set of* traces *for ini is*

$$\begin{aligned} StepTrace_{\hat{C},ini} \quad &:= \quad \{tr \in Steps^* \mid \mathsf{Pr}_{run_{\hat{C},ini}}(tr) > 0\} \\ &\cup \quad \{tr \in Steps^{\infty} \mid \forall l \in \mathbb{N}\colon \mathsf{Pr}_{run_{\hat{C},ini,l}}(tr\lceil_l) > 0\}. \end{aligned}$$

- *The set of* possible views *of a machine set* $\hat{M} \subseteq \hat{C}$ *for ini is*

$$\begin{aligned} \mathit{ViewTrace}_{\hat{C},ini}(\hat{M}) &:= \{v \in \mathit{Steps}^* \mid \mathsf{Pr}_{\mathit{view}_{\hat{C},ini}}(\hat{M})(v) > 0\} \\ &\cup \{v \in \mathit{Steps}^\infty \mid \forall l \in \mathbb{N}\colon \mathsf{Pr}_{\mathit{view}_{\hat{C},ini,l}}(\hat{M})(v\lceil_l) > 0\}. \end{aligned}$$

Set $\mathit{StateTrace}_{\hat{C}} := \bigcup_{ini \in \mathit{Ini}_{\hat{C}}} \mathit{StateTrace}_{\hat{C},ini}$ *and* $\mathit{Trace}_{\hat{C}} := \bigcup_{ini \in \mathit{Ini}_{\hat{C}}} \mathit{Trace}_{\hat{C},ini}$ *and* $\mathit{ViewTrace}_{\hat{C}}(\hat{M}) := \bigcup_{ini \in \mathit{Ini}_{\hat{C}}} \mathit{ViewTrace}_{\hat{C},ini}(\hat{M})$.

◇

We conclude this section with two properties of state-traces and views. The first, technical one states that whenever a non-buffer machine M is switched in a state-trace of $\hat{C}$, which only happens in Phase 1, there is at most one port $p \in \mathsf{ports}(\hat{C})$ with a non-empty value, and it must fulfill $p \in \mathsf{ports}(\mathsf{M})$.

Lemma 4.2 (Unique Inputs in Traces) *Let* $\hat{C}$ *be a complete, closed collection. Let* $S := (S_i)_{i \in \mathbb{N}} \in \mathit{StateTrace}_{\hat{C}}$ *with* $S_i = s_{\mathit{fin}}$ *or* $S_i = ((s^i_{\mathsf{M}})_{\mathsf{M} \in \hat{C}}, \mathsf{M}^i, f^i, j^i, P^i)$ *for all* i. *For all* $i \in \mathbb{N}$ *with* $S_i \neq s_{\mathit{fin}}$*:*

$$(j^i = 1 \wedge \exists p \in \mathsf{ports}(\hat{C})\colon (f^i(p) \neq \epsilon)) \Rightarrow (p \in \mathsf{ports}(\mathsf{M}^i) \wedge \forall p' \in \mathsf{ports}(\hat{C}) \setminus \{p\}\colon (f^i(p') = \epsilon)).$$

□

Proof. For $i = 1$ this holds since $S \in \mathit{StateTrace}_{\hat{C}}$ implies $S_1 \in \mathit{Ini}_{\hat{C}}$ and each element of $\mathit{Ini}_{\hat{C}}$ fulfills the claim with $p = \mathsf{clk}^{\triangleleft}!$. Let now $i > 1$ and S_i with $j^i = 1$. Only two cases are possible for the previous state S_{i-1}. The case $j^{i-1} = 4$ and $\mathsf{M}^i = \mathsf{X}$ fulfills the claim for $p = \mathsf{clk}^{\triangleleft}? \in \mathsf{ports}(\mathsf{X})$, and the case $j^{i-1} = 5$ for $p = \mathsf{n}?$. ■

The second property shows that the views of polynomial-time machines are polynomially bounded, and so are runs of polynomial-time collections.

Lemma 4.3 (Polynomial Views and Traces) *Let* $\hat{M} \subseteq \hat{C}$ *be polynomial-time. For all* $ini = ((ini_{\mathsf{M}})_{\mathsf{M} \in \hat{C}}, \mathsf{X}, f, 1, ()) \in \mathit{Ini}_{\hat{C}}$ *let* $\mathsf{size}_{\hat{M}}(ini) := \sum_{\mathsf{M} \in \hat{M}} \mathsf{size}(ini_{\mathsf{M}})$. *Then there exists a polynomial* P *such that for all* $ini \in \mathit{Ini}_{\hat{C}}$ *and all* $v \in \mathit{ViewTrace}_{\hat{C},ini}(\hat{M})$ *we have* $\mathsf{len}(v) \leq P(\mathsf{size}_{\hat{M}}(ini))$.

If the entire collection $\hat{C}$ *is polynomial-time, then there exists a polynomial* P *such that for all* $ini \in \mathit{Ini}_{\hat{C}}$ *and all* $r \in \mathit{Trace}_{\hat{C},ini}$ *we have* $\mathsf{len}(r) \leq P(\mathsf{size}_{\hat{C}}(ini))$. □

Proof. By Definition 4.11, a polynomial-time machine M only makes a polynomial number of Turing steps, relative to the length $\mathsf{size}(ini_{\mathsf{M}})$ of its own input, which is smaller than the overall input size $\mathsf{size}_{\hat{M}}(ini)$ of the considered machines. Consequently, M can only build up a polynomial-size state and outputs, and read a polynomial-size part of its inputs. Each step in the view requires at least one Turing step; hence there is also only a polynomial number of these steps. Further, only states, outputs, and read parts of the inputs are part of the steps (see Definitions 4.16 and 4.18). This proves the first statement.

A run of a polynomial-time collection $\hat{C}$ consists of the steps of its polynomial-time machines and its buffers. Buffers are not polynomial-time, but weakly polynomial-time; this follows immediately from the assumption we made about the encoding ι of the internal queue. Each buffer obtains all its inputs from polynomial-time machines; hence its overall input is of polynomial length. Thus each buffer only makes a polynomial number of Turing steps. This yields an overall polynomial size of its steps as above. ■

5 Security-specific System Model

We now define specific collections for security purposes. We start with the definition of *structures*. Intuitively, these are the machines that execute a security protocol. Structures additionally have so-called *service ports*, which are a subset of the free ports of the completion of the collection. Roughly, service ports are those where a certain service of the structure is guaranteed. Typical examples of inputs at service ports are "send message m to participant id" for a message transmission system or "pay amount x to participant id" for a payment system. Concretely, service ports will later be used to connect a user machine to a structure. For cryptographic purposes, the initial state of all machines is typically a security parameter k in unary representation. Hence we require that such strings of 1s are valid initial states of all machines.

Definition 5.1 *(Structures and Service Ports) A* structure *is a pair* $struc = (\hat{M}, S)$ *where* $\hat{M}$ *is a collection of simple machines with* $\{1\}^* \subseteq Ini_{\mathsf{M}}$ *for all* $\mathsf{M} \in \hat{M}$*, and* $S \subseteq \mathsf{free}([\hat{M}])$*. The set* S *is called* service ports. ◇

Forbidden ports for users of a structure are those that clash with port names of given machines and those that would link the user to a non-service port.

Definition 5.2 *(Forbidden Ports) For a structure* $(\hat{M}, S)$ *let* $\bar{S}_{\hat{M}} := \mathsf{free}([\hat{M}]) \setminus S$*. We call* $\mathsf{forb}(\hat{M}, S) := \mathsf{ports}(\hat{M}) \cup \bar{S}^c_{\hat{M}}$ *the* forbidden ports. ◇

A *system* is a set of structures. The idea behind systems, as motivated in Section 2, is that there may be different actual structures depending on the set of actually malicious participants. Typical derivations of systems from one explicitly defined intended structure and a trust model will be discussed in Section 7.

Definition 5.3 *(Systems) A* system *Sys is a set of structures. It is (weakly) polynomial-time iff the machine collections* $\hat{M}$ *of all its structures are (weakly) polynomial-time.* ◇

A structure can be complemented to a *configuration* by adding a *user machine* and an *adversary machine* (or *user* and *adversary* for short). The user is restricted to connecting to the service ports. The adversary closes the collection, i.e., it connects to the remaining service ports, to the other free ports $\bar{S}_{\hat{M}}$ of the collection, and to the free ports of the user. Thus, user and adversary can interact, e.g., for modeling active attacks.

Definition 5.4 *(Configurations)*

a) A configuration *of a structure* $(\hat{M}, S)$ *is a tuple* $conf = (\hat{M}, S, \mathsf{H}, \mathsf{A})$ *where*

- H *is a machine called* user *without forbidden ports, i.e.,* $\mathsf{ports}(\mathsf{H}) \cap \mathsf{forb}(\hat{M}, S) = \emptyset$*, and with* $\{1\}^* \subseteq Ini_{\mathsf{H}}$*,*
- A *is machine called* adversary *with* $\{1\}^* \subseteq Ini_{\mathsf{A}}$*,*
- *and the completion* $\hat{C} := [\hat{M} \cup \{\mathsf{H}, \mathsf{A}\}]$ *is a closed collection.*

b) The set of configurations of $(\hat{M}, S)$ *is written* $\mathsf{Conf}(\hat{M}, S)$*. The set of configurations of* $(\hat{M}, S)$ *with polynomial-time user* H *and adversary* A *is written* $\mathsf{Conf}_{\mathsf{poly}}(\hat{M}, S)$*.*

c) The set of configurations of a system Sys is defined as $\mathsf{Conf}(Sys) := \bigcup_{(\hat{M},S)\in Sys} \mathsf{Conf}(\hat{M}, S)$*, and similarly* $\mathsf{Conf}_{\mathsf{poly}}(Sys) := \bigcup_{(\hat{M},S)\in Sys} \mathsf{Conf}_{\mathsf{poly}}(\hat{M}, S)$*.*

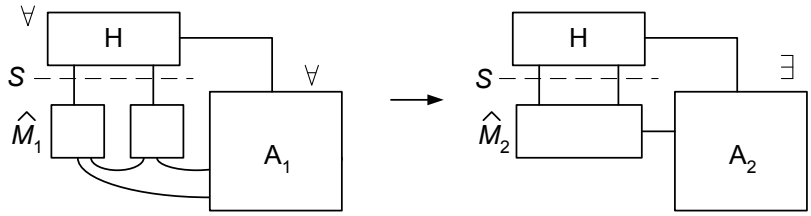

Figure 4: Example of reactive simulatability. The view of H is compared. Whatever an adversary A_1 can achieve in the configuration on the left-hand side, another A_2 can achieve the same in the configuration on the right-hand side.

We omit the index "poly*" from* $\mathsf{Conf}_{\mathsf{poly}}(Sys)$ *if it is clear from the context.* ◇

In cryptographic applications, all machines typically start with the same security parameter. Hence we make a global constraint on the valid global initial states in the family of runs of a configuration that the security parameters are equal.

Definition 5.5 *(Runs and Views of Configurations) Let* $conf = (\hat{M}, S, \mathsf{H}, \mathsf{A})$ *be a configuration and* $\hat{C} := [\hat{M} \cup \{\mathsf{H}, \mathsf{A}\}]$*. We define* $Ini_{conf} := \{((1^k)_{\mathsf{M} \in \hat{M} \cup \{\mathsf{H},\mathsf{A}\}} \circ (\iota(\epsilon))_{\tilde{\mathsf{n}} \in \hat{C}}, (\mathsf{X}, f, 1, ())) \mid k \in \mathbb{N}\} \subseteq Ini_{\hat{C}}$ *with* X *and* f *as in Definition 4.15. Then we define the* family of probability distributions of runs of the configuration *as*

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

and for all sets $\hat{M}' \subseteq \hat{C}$ *the* family of probability distributions of views *similarly*

$$view_{\hat{C}}(\hat{M}') := (view_{\hat{C},ini}(\hat{M}'))_{ini \in Ini_{conf}},$$

and analogously for l*-step prefixes. Furthermore, we identify* Ini_{conf} *with* $\mathbb{N}$ *and thus write* $run_{conf,k}$ *etc. for the individual probability distributions in the families.* ◇

6 Reactive Simulatability

The definition of one structure securely implementing another one is based on the concept of simulatability. Our definition is the first such definition for an asynchronous reactive setting and is hence called *reactive simulatability*. As explained in Section 2, reactive simulatability essentially means that whatever might happen to an honest user in a real structure $(\hat{M}_1, S)$ can also happen in an ideal structure $(\hat{M}_2, S)$. More precisely, for every configuration $conf_1 \in \mathsf{Conf}(\hat{M}_1, S)$, there exists a configuration $conf_2 \in \mathsf{Conf}(\hat{M}_2, S)$ with the same user yielding *indistinguishable views* for this user in both configurations. A typical situation is illustrated in Figure 4.

We need notions that functions are small; hence we define corresponding function classes.

Definition 6.1 *(Small functions)*

a) The class *NEGL* of *negligible functions* contains all functions $s\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ that decrease faster than the inverse of every polynomial, i.e., for all positive polynomials Q $\exists k_0\ \forall k > k_0 : s(k) < \frac{1}{Q(k)}$.

b) A set *SMALL* of functions $\mathbb{N} \to \mathbb{R}_{\geq 0}$ is a *class of small functions* if it is closed under addition, and with a function g also contains every function $g' \colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ with $g' \leq g$.

◇

Typical classes of small functions are *EXPSMALL*, which contains all functions bounded by $Q(k) \cdot 2^{-k}$ for a polynomial Q, and the larger class *NEGL*.

Simulatability is based on indistinguishability of views; hence we repeat the definition of indistinguishability, essentially from [63].

Definition 6.2 *(Indistinguishability) Two families* $(\mathsf{var}_k)_{k\in\mathbb{N}}$ *and* $(\mathsf{var}'_k)_{k\in\mathbb{N}}$ *of probability distributions (or random variables) on common domains* $(D_k)_{k\in\mathbb{N}}$ *are*

a) perfectly indistinguishable *("=") iff* $\forall k \in \mathbb{N} \colon \mathsf{var}_k = \mathsf{var}'_k$.

b) statistically indistinguishable *("$\approx_{SMALL}$") for a class SMALL of small functions iff the distributions are discrete and their statistical distances, as a function of k, are small, i.e.,*

$$(\Delta_{\mathsf{stat}}(\mathsf{var}_k, \mathsf{var}'_k))_{k\in\mathbb{N}} := (\frac{1}{2} \sum_{d \in D_k} |\mathsf{Pr}(\mathsf{var}_k = d) - \mathsf{Pr}(\mathsf{var}'_k = d)|)_{k\in\mathbb{N}} \in SMALL.$$

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

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

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

We write $\approx$ *if we want to treat all cases together.* ◇

We now present the reactive simulatability definition. One technical problem is that a user might legitimately connect to the service ports in a configuration of $(\hat{M}_1, S)$, but in a configuration of $(\hat{M}_2, S)$ the same user might have forbidden ports. This is excluded by considering *suitable configurations* only.

Definition 6.3 *(Suitable Configurations for Structures) Let* $(\hat{M}_1, S)$ *and* $(\hat{M}_2, S)$ *be structures with the same set of service ports. The set of* suitable configurations $\mathsf{Conf}^{\hat{M}_2}(\hat{M}_1, S) \subseteq \mathsf{Conf}(\hat{M}_1, S)$ *is defined by* $(\hat{M}_1, S, \mathsf{H}, \mathsf{A}) \in \mathsf{Conf}^{\hat{M}_2}(\hat{M}_1, S)$ *iff* $\mathsf{ports}(\mathsf{H}) \cap \mathsf{forb}(\hat{M}_2, S) = \emptyset$. *The set of* polynomial-time suitable configurations *is* $\mathsf{Conf}^{\hat{M}_2}_{\mathsf{poly}}(\hat{M}_1, S) := \mathsf{Conf}^{\hat{M}_2}(\hat{M}_1, S) \cap \mathsf{Conf}_{\mathsf{poly}}(\hat{M}_1, S)$. ◇

As we have three different notions of indistinguishability, our reactive simulatability definition also comes in three flavors. Further, we distinguish the general simulatability as sketched so far and a stronger *universal* version where one adversary A_2 must be able to work for all users.

Definition 6.4 *(Reactive Simulatability for Structures) Let structures* $(\hat{M}_1, S)$ *and* $(\hat{M}_2, S)$ *with identical sets of service ports be given.*

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

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

b) $(\hat{M}_1, S) \geq_{\mathsf{sec}}^{SMALL} (\hat{M}_2, S)$, *spoken* statistically at least as secure as*, for a class SMALL of small functions iff for every configuration* $conf_1 = (\hat{M}_1, S, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}^{\hat{M}_2}(\hat{M}_1, S)$, *there exists a configuration* $conf_2 = (\hat{M}_2, S, \mathsf{H}, \mathsf{A}_2) \in \mathsf{Conf}(\hat{M}_2, S)$ *(with the same* H*) such that*

$$view_{conf_1,l}(\mathsf{H}) \approx_{SMALL} view_{conf_2,l}(\mathsf{H})$$

for all polynomials l*, i.e., if all families of* l*-step prefixes of the views are statistically indistinguishability.*

c) $(\hat{M}_1, S) \geq_{\mathsf{sec}}^{\mathsf{poly}} (\hat{M}_2, S)$, *spoken* computationally at least as secure as*, iff for every configuration* $conf_1 = (\hat{M}_1, S, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}_{\mathsf{poly}}^{\hat{M}_2}(\hat{M}_1, S)$, *there exists a configuration* $conf_2 = (\hat{M}_2, S, \mathsf{H}, \mathsf{A}_2) \in \mathsf{Conf}_{\mathsf{poly}}(\hat{M}_2, S)$ *(with the same* H*) such that*

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

In all three cases, we speak of universal simulatability *if* A_2 *in* $conf_2$ *does not depend on* H *(only on* $\hat{M}_1$*,* S*, and* A_1*), and we use the notation* $\geq_{\mathsf{sec}}^{\mathsf{uni,perf}}$ *etc. for this. In all cases, we call* $conf_2$ *an* indistinguishable configuration *for* $conf_1$*.*

◇

There is also a notion of blackbox simulatability, where the adversary A_2 consists of a fixed part, called *simulator*, using A_1 as a blackbox submachine. However, its rigorous definition needs the notion of machine combination, which we postpone to the successor paper dealing with composition. If one can simply set $\mathsf{A}_2 := \mathsf{A}_1$, we also say that the structures are *indistinguishable*; this corresponds to the definition in [39].

Where the difference between the types of security is irrelevant, we simply write $\geq_{\mathsf{sec}}$, and we omit the index sec if it is clear from the context.

Remark 6.1. Adding a free adversary out-port in the comparison (like the guessing-outputs used to define semantic security of encryption systems in [30]) does not make the definition stricter: Any such out-port can be connected to an in-port added to the honest user with sufficiently large length bounds. H does not react on inputs at this new in-port, but nevertheless it is included in the view of H, i.e., in the comparison. A rigorous proof can be found in [4]. ◦

The definition of reactive simulatability can be lifted from structures to systems Sys_1 and Sys_2 by comparing their respective structures. However, we do not want to compare a structure of Sys_1 with arbitrary structures of Sys_2, but only with certain "suitable" ones. What suitable means in a concrete situation can be defined by a mapping f from Sys_1 to the powerset of Sys_2. The mapping f is called *valid* if it maps structures with the same set of service ports, so that the same user can connect. For many systems there is only one possible mapping that meets this requirement, because the service ports of the structures correspond to one-to-one to different sets of non-corrupted machines. This mapping is then called *canonical*.

Definition 6.5 *(Valid Mappings) Let Sys_1 and Sys_2 be two systems.*

a) A valid mapping *between Sys_1 and Sys_2 is a function $f\colon Sys_1 \to \mathcal{P}(Sys_2)\setminus\emptyset$ with $S_1 = S_2$ for all $(\hat{M}_1, S_1) \in Sys_1$ and $(\hat{M}_2, S_2) \in f((\hat{M}_1, S_1))$. Structures $(\hat{M}_2, S_2)$ with $(\hat{M}_2, S_2) \in f((\hat{M}_1, S_1))$ are called* corresponding structures *of $(\hat{M}_1, S_1)$.*

b) If there is only one valid mapping f between Sys_1 and Sys_2, it is called the canonical mapping*.*

◇

Remark 6.2. In the synchronous model in [55], we allow more general users and valid mappings. The stronger requirements here simplify the presentation and are sufficient for all cryptographic examples we considered. Its report version contains non-cryptographic examples with $S_1 \neq S_2$. ∘

An example of a system that contains different structures with the same service ports, so that a non-canonical mapping is needed, is a protocol with a semi-trusted third-party machine T which needs no user interface because its behavior is fully prescribed by the protocol, and where different properties can be achieved depending on whether T is correct or not.

Before we introduce reactive simulatability for systems, we define suitable configurations for entire systems with respect to a valid mapping.

Definition 6.6 *(Suitable Configurations for Systems) Let Sys_1, Sys_2 be systems, and f a valid mapping between Sys_1 and Sys_2. The set of* suitable configurations $\mathsf{Conf}^f(Sys_1) \subseteq \mathsf{Conf}(Sys_1)$ *is defined by $conf \in \mathsf{Conf}^f(Sys_1)$ iff $conf \in \mathsf{Conf}^{\hat{M}_2}(\hat{M}_1, S)$ for all $(\hat{M}_2, S) \in f((\hat{M}_1, S))$. The set of* polynomial-time suitable configurations *is $\mathsf{Conf}^f_{\mathsf{poly}}(Sys_1) := \mathsf{Conf}^f(Sys_1) \cap \mathsf{Conf}_{\mathsf{poly}}(Sys_1)$.* ◇

Definition 6.7 *(Reactive Simulatability for Systems) Let Sys_1 and Sys_2 be systems, and f a valid mapping between Sys_1 and Sys_2. Let $x \in \{\mathsf{perf}, \mathsf{poly}\}$ or $x = SMALL$ for a class $SMALL$ of small functions, and let $x' \in \{x, (\mathsf{uni}, x)\}$.*

Then $Sys_1 \geq^{f,x'}_{\mathsf{sec}} Sys_2$ if for every $(\hat{M}_1, S) \in Sys_1$ there exists with $(\hat{M}_2, S) \in f((\hat{M}_1, S))$ such that

$$(\hat{M}_1, S) \geq^{x'}_{\mathsf{sec}} (\hat{M}_2, S).$$

Depending on the value of x, we say that Sys_1 is perfectly / statistically / computationally at least as secure as *Sys_2. We omit the indices f, x', and sec if they are clear from the context.* ◇

We conclude this section with two technical lemmas capturing an equivalent definition of the forbidden ports of a structure and additional results on valid mappings and suitable configurations, which is useful in proofs.

Recall that Definition 5.4 excludes users that would connect to the forbidden ports of a configuration. The following lemma establishes an equivalent condition.

Lemma 6.1 (Users) *Let $(\hat{M}, S)$ be a structure. Then for all machines H, $\mathsf{ports}(\mathsf{H}) \cap \mathsf{forb}(\hat{M}, S) = \emptyset$ is equivalent to $\mathsf{ports}(\mathsf{H}) \cap \mathsf{ports}(\hat{M}) = \emptyset$ (1) and $\mathsf{ports}(\mathsf{H})^c \cap \mathsf{ports}([\hat{M}]) \subseteq S$ (2).* □

Proof. Let $\mathsf{inner}(\hat{C}) := \mathsf{ports}(\hat{C}) \setminus \mathsf{free}(\hat{C})$ for every collection $\hat{C}$. Clearly $\mathsf{inner}(\hat{C})^c = \mathsf{inner}(\hat{C})$. The condition on the left-hand side is equivalent to (1) and $\mathsf{ports}(\mathsf{H})^c \cap (\mathsf{free}([\hat{M}]) \setminus S) = \emptyset$ (3). Now (3) $\Leftrightarrow \mathsf{ports}(\mathsf{H})^c \cap \mathsf{free}([\hat{M}]) \subseteq S$. It remains to be shown that $\mathsf{ports}(\mathsf{H})^c \cap \mathsf{inner}([\hat{M}]) = \emptyset$. This is equivalent to $\mathsf{ports}(\mathsf{H}) \cap \mathsf{inner}([\hat{M}]) = \emptyset$. Now $\mathsf{ports}([\hat{M}])$ only contains additional buffer ports and clock in-ports compared with $\mathsf{ports}(\hat{M})$. Hence (1) even implies $\mathsf{ports}(\mathsf{H}) \cap \mathsf{ports}([\hat{M}]) = \emptyset$. ∎

The following lemma shows that the ports of a structure that users are intended to use, i.e., the complement of the service ports, are not at the same time forbidden. Moreover, it shows that a non-suitable configuration can be transformed into a suitable one via port renaming such that this renaming does not affect the view of the user. This means that the restriction to suitable configurations in the definition of reactive simulatability is without loss of generality.

Lemma 6.2 (Valid Mappings and Suitable Configurations) *Let $(\hat{M}_1, S)$ and $(\hat{M}_2, S)$ be structures.*

a) Then $S^c \cap \mathsf{forb}(\hat{M}_1, S) = \emptyset$.

b) For every $conf_1 = (\hat{M}_1, S, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}(\hat{M}_1, S) \setminus \mathsf{Conf}^{\hat{M}_2}(\hat{M}_1, S)$, there is a configuration $conf_{\mathsf{f},1} = (\hat{M}_1, S, \mathsf{H}_\mathsf{f}, \mathsf{A}_{\mathsf{f},1}) \in \mathsf{Conf}^{\hat{M}_2}(\hat{M}_1, S)$ such that $view_{conf_{\mathsf{f},1}}(\mathsf{H}_\mathsf{f}) = view_{conf_1}(\mathsf{H})$.

□

Proof. For Part a) recall that $\mathsf{forb}(\hat{M}, S) = \mathsf{ports}(\hat{M}_1) \cup (\mathsf{free}([\hat{M}_1]) \setminus S)^c$. The part $S^c \cap (\mathsf{free}([\hat{M}_1]) \setminus S)^c = \emptyset$ is clear, and $S^c \cap \mathsf{ports}(\hat{M}_1) = \emptyset$ follows from $S \subseteq \mathsf{free}([\hat{M}_1])$.

For Part b), we want to construct H_f by giving each port $p = \mathsf{n}ld \in \mathsf{ports}(\mathsf{H}) \cap \mathsf{forb}(\hat{M}_2, S)$ a new name. Since runs and views do not depend on port names, cf. Definition 4.5, they remain the same if we consistently rename all other ports q (at most five) with $\mathsf{name}(q) = \mathsf{n}$. The new collection is a configuration $(\hat{M}_1, S, \mathsf{H}_\mathsf{f}, \mathsf{A}_{\mathsf{f},1})$ if no renamed port belongs to $\hat{M}_1$. Assume that $q = \mathsf{n}l'd' \in \mathsf{ports}(\hat{M}_1)$ is such a renamed port, then $\widetilde{\mathsf{n}} \in [\hat{M}_1]$ and hence $p^c \in \mathsf{ports}([\hat{M}_1])$. Now Lemma 6.1 implies $p^c \in S$, hence Part a) applied to the structure $(\hat{M}_2, S)$ implies $p \notin \mathsf{forb}(\hat{M}_2, S)$, in contradiction to the original condition on p. ∎

7 Special Cases for Cryptographic Purposes

After defining structures, systems and reactive simulatability generally in the previous sections, we now define some specializations of the general model, which are of special importance for cryptographic purposes. We define *standard cryptographic systems* which are derived from an *intended structure* and a *trust model*. Intuitively, an intended structure represents a benign world, where each machine is correct. A trust model is then used to designate the potentially malicious sets of machines, i.e., machines which are considered to come under control of the adversary. Moreover, it classifies the "security" of each connection between machines of the structure, e.g., that a channel keeps messages authentic, but not secret. This is depicted in Figure 5: the left-hand side shows an intended structure consisting of three machines; the right-hand side shows a possible derivation where machine M_3 is malicious and channels have been modified.

Cryptographic systems come in two flavors, depending on how the adversary takes control over certain machines. If malicious machines are malicious from the beginning, we call the system a *static standard cryptographic system*. If the adversary may corrupt machines during

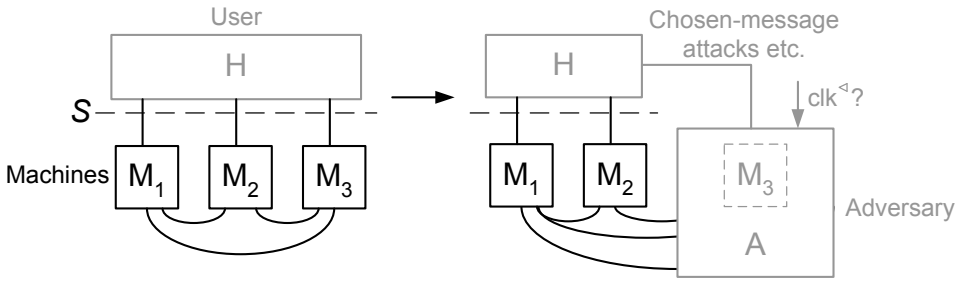

Figure 5: Derivation of one possible structure from an intended structure. The structures are the black parts.

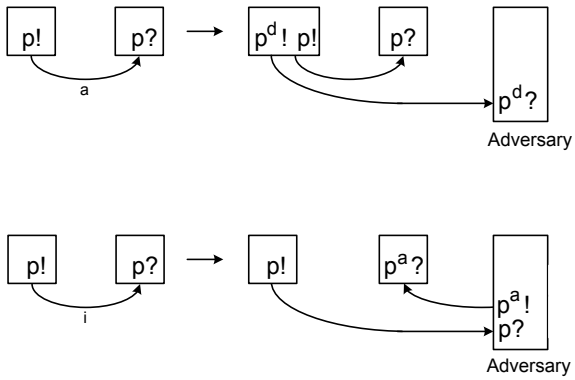

Figure 6: Channel modifications for authenticated and insecure channels.

the protocol executions, depending on the knowledge that he has already collected, we speak of an *adaptive standard cryptographic system*.

7.1 Trust Models and Intended Structures

We start with the definition of trust models for a structure. Trust models consist of two parts: an *access structure* and a *channel model*. Access structures will later be used to denote the possible sets of *correct* machines in an intended structure. Access structures have to be closed under insertion, i.e., with every such set, every larger set is contained as well.

Definition 7.1 *(Access Structure) Let A be an arbitrary set. Then $\mathcal{ACC} \subseteq \mathcal{P}(A)$ is called an* access structure *for A iff $(B \in \mathcal{ACC} \wedge C \in \mathcal{P}(A) \wedge B \subseteq C) \Rightarrow C \in \mathcal{ACC}$.* ◇

Typical examples of access structures are threshold structures $\mathcal{ACC}_{t,n} := \{\mathcal{H} \subseteq A \mid |\mathcal{H}| \geq t\}$ with $t \leq n$.

A channel model for a structure classifies each internal high-level connection as *secure* (private and authentic), *authenticated* (only authentic), or *insecure* (neither private nor authentic), represented as elements of the set $\{\mathsf{s}, \mathsf{a}, \mathsf{i}\}$. What this means will become clear in Section 7.2.

Definition 7.2 *(Channel Model) A* channel model *for a structure $(\hat{M}^*, S^*)$ is a mapping $\chi\colon \mathsf{Gr}(\hat{M}^*) \to \{\mathsf{s}, \mathsf{a}, \mathsf{i}\}$.* ◇

Definition 7.3 *(Trust Model) A* trust model *for a structure $(\hat{M}, S)$ is a pair $(\mathcal{ACC}, \chi)$ where $\mathcal{ACC}$ is an access structure for $\hat{M}$, and χ is a channel model for $(\hat{M}, S)$.* ◇

We proceed with the definition of intended structures, i.e., structures a designer of a security protocol would typically design. An intended structure is a structure that is benign in the sense that it does not offer any free simple ports for the adversary. Moreover, we demand that a machine of a structure may only schedule those connections for which it owns the corresponding input or output port, i.e., it does not intend to schedule users and adversaries. We distinguish three different kinds of intended structures, depending on how the channels between the system and the user are clocked. We call the structure *localized* if every output is clocked by the outputting machine itself. Structures of this kind are typically used as local submachines that can be clocked by the overall protocol and then immediately deliver a result. If the adversary clocks the communication between the structure and the user, we call the structure *stand-alone*. Finally, if the user and the system have to fetch the outputs of the other, we call the structure *fetching*.

Remark 7.1. We could as well distinguish channels from the user to the system and vice versa, e.g., to define that users schedule their outputs and fetch their inputs. This would give nine different combinations. Modifying the upcoming definition of intended structures in such a way is trivial. ∘

The derivation of the remaining structures based on the intended structure will rely on modifying the connections in an adequate way, e.g., by duplicating output ports that output on authentic channels so that one output port is connected as usual and the duplicated port is connected to the adversary. Hence derived structures need an extended set of possible port names. Moreover, we will need a distinguished state and distinguished ports to model adaptive corruptions of machines. Technically, we therefore parameterize intended structures by an additional alphabet $\Gamma \subsetneq \Sigma$ with $|\Gamma| = |\Sigma| - 2$ and a state s_{corr}, and we restrict port names in the intended structure to Γ^+ and possible states to $\Sigma^* \setminus \{s_{\mathsf{corr}}\}$.

Definition 7.4 *(Intended Structure) A structure $(\hat{M}^*, S^*)$ is called an* intended structure *for $s_{\mathsf{corr}} \in \Sigma^*$ and $\Gamma \subset \Sigma$ with $|\Gamma| = |\Sigma| - 2$ iff*

- *all $\mathsf{M} \in \hat{M}^*$ are simple, $\mathsf{name}(p) \in \Gamma^+$ for all $p \in \mathsf{ports}(\hat{M}^*)$, and $s_{\mathsf{corr}} \notin States_{\mathsf{M}}$ for all $\mathsf{M} \in \hat{M}^*$,*
- *for all $\mathsf{M} \in \hat{M}^*$: $(\mathsf{n}^{\triangleleft}! \in \mathsf{ports}(\mathsf{M}) \Rightarrow (\mathsf{n}? \in \mathsf{ports}(\mathsf{M})) \vee (\mathsf{n}! \in \mathsf{ports}(\mathsf{M})))$, and*
- *it has the following properties. Let $S' := \{p \in \mathsf{free}([\hat{M}^*]) \mid \mathsf{label}(p^c) = \epsilon\}$.*
  - *The structure is called* localized *iff*

    $$S^* = S' \cup \{\mathsf{n}^{\triangleleft}? \mid \mathsf{n}! \in \mathsf{free}([\hat{M}^*])^c\}.$$

    and the following condition on the port set of $\hat{M}^*$ holds:

    $\mathsf{n}? \in \mathsf{free}([\hat{M}^*])^c \Rightarrow \mathsf{n}^{\triangleleft}! \in \mathsf{ports}(\hat{M}^*)$ *# $\hat{M}^*$ schedules its outputs to the user*
  - *The structure is called* stand-alone *iff $S^* = S'$ and*

    $(\mathsf{n}, \epsilon, d) \in \mathsf{free}([\hat{M}^*])^c \Rightarrow \mathsf{n}^{\triangleleft}! \notin \mathsf{ports}(\hat{M}^*)$ *# $\hat{M}^*$ does not schedule any connection # between the user and $\hat{M}^*$*

– *The structure is called* fetching *iff*

$$S^* = S' \cup \{\mathsf{n}^{\triangleleft}? \mid \mathsf{n}? \in \mathsf{free}([\hat{M}^*])^c\}$$

and the following condition on the port set of $\hat{M}^*$ *holds:*

$$\mathsf{n}! \in \mathsf{free}([\hat{M}^*])^c \Rightarrow \mathsf{n}^{\triangleleft}! \in \mathsf{ports}(\hat{M}^*) \quad \# \ \hat{M}^* \text{ schedules the inputs from the user}$$

◇

7.2 Standard Static Cryptographic Systems

Standard static cryptographic systems are derived from an intended structure and a trust model as follows. Each system contains one structure for each element of the considered access structure, i.e., for each set $\mathcal{H}$ of potential correct machines.

The channel model is taken into account as follows. Intuitively, we change the connections such that the adversary receives messages sent on authenticated and insecure channels, and we enable him to arbitrarily modify messages sent on insecure channels. This is modeled as depicted in Figure 6 for authenticated and insecure channels; secure channels remain unchanged. Authenticated channels are modeled by additional out-ports $\mathsf{p}^{\mathsf{d}}!$ where outputs at $\mathsf{p}!$ are duplicated. The port $\mathsf{p}^{\mathsf{d}}!$ remains free; hence the adversary can connect to it. Insecure channels are modeled by renaming the input port. This breaks the existing connection and places the adversary in between.

Moreover, if $\mathsf{p}?$ belongs to a correct machine and $\mathsf{p}!$ does not, we also rename $\mathsf{p}?$ into $\mathsf{p}^{\mathsf{a}}?$ so that all inputs from the adversary have superscript a. By applying these changes to a machine M, we get a modified machine $\mathsf{M}_{\mathcal{H},\chi}$.

Formally, let $\{\mathsf{a},\mathsf{d}\} := \Sigma \setminus \Gamma$. Then we define two mappings φ^{a} and φ^{d} that assign each port $p = (\mathsf{n}, l, d)$ with $\mathsf{n} \in \Gamma^+$ ports (an, l, d) and (dn, l, d) respectively. We write p^{a} and p^{d} instead of $\varphi^{\mathsf{a}}(p)$ and $\varphi^{\mathsf{d}}(p)$.

We now define the derivation of static cryptographic systems from a given intended structure and a trust model rigorously. Similar to the definition of runs, the semi-formal description given above is sufficient to understand our subsequent results, so the following technical definition can be skipped at first reading.

Definition 7.5 *(Derivation of Standard Static Cryptographic Systems) Let* $(\hat{M}^*, S^*)$ *be an intended structure for a state* s_{corr} *and a set* Γ*, and let* $(\mathcal{ACC}, \chi)$ *be a trust model for* $(\hat{M}^*, S^*)$*. Then the corresponding* cryptographic system with static adversary

$$Sys = StanStat((\hat{M}^*, S^*), (\mathcal{ACC}, \chi))$$

is $Sys := \{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}}) \mid \mathcal{H} \in \mathcal{ACC}\}$ *where for all* $\mathcal{H} \in \mathcal{ACC}$*:*

- $S_{\mathcal{H}} := S^* \cap \mathsf{free}([\mathcal{H}])$.
- $\hat{M}_{\mathcal{H}} := \{\mathsf{M}_{\mathcal{H},\chi} | \mathsf{M} \in \mathcal{H}\}$, *where*

$$\mathsf{M}_{\mathcal{H},\chi} = (\mathsf{name}_{\mathsf{M}}, Ports_{\mathsf{M}_{\mathcal{H},\chi}}, States_{\mathsf{M}}, \delta_{\mathsf{M}_{\mathcal{H},\chi}}, l_{\mathsf{M}}, Ini_{\mathsf{M}}, Fin_{\mathsf{M}})$$

is defined as follows:

– *The sequence* $Ports_{\mathsf{M}_{\mathcal{H},\chi}}$ *is derived by the following algorithm.*

$Ports_{\mathsf{M}_{\mathcal{H},\chi}} := ()$.
for $p \in Ports_{\mathsf{M}}$ *(in the given order)* ***do***
if $c := \{p, p^C\} \in \mathsf{Gr}(\mathcal{H}) \wedge \chi(c) = \mathsf{a} \wedge \mathsf{dir}(p) = !$ ***then***
$Ports_{\mathsf{M}_{\mathcal{H},\chi}} := Ports_{\mathsf{M}_{\mathcal{H},\chi}} \circ (p, p^{\mathsf{d}})$
else if $c := \{p, p^C\} \in \mathsf{Gr}(\mathcal{H}) \wedge \chi(c) = \mathsf{i} \wedge \mathsf{dir}(p) = ?$ ***then***
$Ports_{\mathsf{M}_{\mathcal{H},\chi}} := Ports_{\mathsf{M}_{\mathcal{H},\chi}} \circ (p^{\mathsf{a}})$
else if $c := \{p, p^C\} \notin \mathsf{Gr}(\mathcal{H}) \wedge \mathsf{dir}(p) = ?$ ***then***
$Ports_{\mathsf{M}_{\mathcal{H},\chi}} := Ports_{\mathsf{M}_{\mathcal{H},\chi}} \circ (p^{\mathsf{a}})$
else
$Ports_{\mathsf{M}_{\mathcal{H},\chi}} := Ports_{\mathsf{M}_{\mathcal{H},\chi}} \circ (p)$
end if
end for

– *Let* $s, s' \in States_{\mathsf{M}}$, $I \in (\Sigma^*)^{|\mathsf{in}(\mathsf{ports}(\mathsf{M}))|}$, $O^1 \in (\Sigma^*)^{|\mathsf{out}(\mathsf{ports}(\mathsf{M}))|}$ *and* $O^2 \in (\Sigma^*)^{|\mathsf{out}(\mathsf{ports}(\mathsf{M}_{\mathcal{H},\chi}))|}$. *Then* $\mathsf{Pr}(\delta_{\mathsf{M}_{\mathcal{H},\chi}}(s, I) = (s', O^2)) := \mathsf{Pr}(\delta_{\mathsf{M}}(s, I) = (s', O^1))$ *if* O^2 *is derived from* O^1 *by the following algorithm, and zero otherwise.*[2]

$i := 1$
for $j := 1, \ldots, |\mathsf{out}(\mathsf{ports}(\mathsf{M}_{\mathcal{H},\chi}))|$ ***do***
if $(\mathsf{out}(Ports_{\mathsf{M}_{\mathcal{H},\chi}}))_j = p^{\mathsf{d}}$ *for a port* p ***then***
$O^2_j := O^2_{j-1}$
else
$O^2_j := O^1_i$
$i := i + 1$
end if
end for

◇

7.3 Standard Cryptographic Systems with Adaptive Adversaries

Standard static cryptographic systems as defined in the previous section are based on the intuition that corrupted machines are corrupted right from the start, e.g., they belong to untrusted owners. In *adaptive* (or *dynamic*) adversary models the set of corrupted machines can increase over time, e.g., because there is a "master adversary" who has to hack into machines in order to corrupt them [15, 21]. Adaptive adversary models are more powerful than static ones, i.e., there are examples of systems secure against static adversaries that are insecure against adaptive adversaries who can corrupt the same number of machines [21].

For a given intended structure and a channel model, the corresponding *cryptographic system with adaptive adversary* has only one structure. Similar to the derivation of static cryptographic systems from an intended structure, we define the derivation of a machine $\mathsf{M}_{\mathsf{corr},\chi}$ from each machine M. This derivation is used to grant the adversary the possibility to corrupt machines during the global execution. This is modeled by giving $\mathsf{M}_{\mathsf{corr},\chi}$ a *corruption port* $\mathsf{corrupt}_{\mathsf{M}_{\mathsf{corr},\chi}}?$, which is used for corruption requests, and two new ports $\mathsf{cor\_out}_{\mathsf{M}_{\mathsf{corr},\chi}}!$, $\mathsf{cor\_in}_{\mathsf{M}_{\mathsf{corr},\chi}}?$ for communication with the adversary after corruption. We assume that these ports must neither occur in an intended structure nor after port renaming as defined for the static case; this can be achieved by encoding the names of these ports into $\Sigma^+ \setminus (\{\epsilon, \mathsf{a}, \mathsf{d}\} \circ \Gamma^+)$ where $\Sigma = \Gamma \cup \{\mathsf{a}, \mathsf{d}\}$.

[2] Note that $|\mathsf{in}(\mathsf{ports}(\mathsf{M}))| = |\mathsf{in}(\mathsf{ports}(\mathsf{M}_{\mathcal{H},\chi}))|$ by definition; hence I is also a valid input tuple for the machine $\mathsf{M}_{\mathcal{H},\chi}$.

The corruption port must connect to the service ports. Upon a non-empty input at the corruption port, $\mathsf{M}_{\mathsf{corr}}$ sends its current state to the adversary via $\mathsf{cor\_out}_{\mathsf{M}_{\mathsf{corr},\chi}}!$, and from now on acts *transparently*, i.e., every input $(I_1, \ldots, I_s)$ is translated into the output $\iota(I_1, \ldots, I_s)$ at $\mathsf{cor\_out}_{\mathsf{M}_{\mathsf{corr},\chi}}!$, and every input (b) at $\mathsf{cor\_in}_{\mathsf{M}_{\mathsf{corr},\chi}}?$ is first decomposed as $(O_1, \ldots, O_t) := \iota^{-1}(b)$ and then output at the respective output ports.

Definition 7.7 *(Derivation of Standard Adaptive Cryptographic Systems) Let $(\hat{M}^*, S^*)$ be an intended structure for a state s_{corr} and a set Γ, and let χ be a channel model χ for $(\hat{M}^*, S^*)$. Then the corresponding* cryptographic system with adaptive adversary

$$Sys = StanAdap((\hat{M}^*, S^*), \chi)$$

is $Sys := \{(\hat{M}, S)\}$ where

- $S := S^* \cup \{\mathsf{corrupt}_{\mathsf{M}}{}^{\leftrightarrow}? \mid \mathsf{M} \in \hat{M}^*\}$.
- $\hat{M} := \{\mathsf{M}_{\mathsf{corr},\chi} \mid \mathsf{M} \in \hat{M}^*\}$, *where $\mathsf{M}_{\mathsf{corr},\chi}$ is derived from M with $\mathcal{H} = \hat{M}$ as follows: Let $\mathsf{M}_{\mathcal{H},\chi}$ be the machine defined in the static case (Definition 7.5). Then*

  $$\mathsf{M}_{\mathsf{corr},\chi} = (\mathsf{name}_{\mathsf{M}}, Ports_{\mathsf{M}_{\mathsf{corr},\chi}}, States_{\mathsf{M}_{\mathsf{corr},\chi}}, \delta_{\mathsf{M}_{\mathsf{corr},\chi}}, l_{\mathsf{M}_{\mathsf{corr},\chi}}, Ini_{\mathsf{M}}, Fin_{\mathsf{M}})$$

  is defined as follows:

  - $Ports_{\mathsf{M}_{\mathsf{corr},\chi}} := Ports_{\mathsf{M}_{\mathcal{H},\chi}} \circ (\mathsf{corrupt}_{\mathsf{M}_{\mathsf{corr},\chi}}?, \mathsf{cor\_out}_{\mathsf{M}_{\mathsf{corr},\chi}}!, \mathsf{cor\_in}_{\mathsf{M}_{\mathsf{corr},\chi}}?)$.
  - $States_{\mathsf{M}_{\mathsf{corr},\chi}} := States_{\mathsf{M}} \cup \{s_{\mathsf{corr}}\}$.
  - $l_{\mathsf{M}_{\mathsf{corr},\chi}}(s) := l_{\mathsf{M}}(s) \circ (1, 0)$ *for* $s \in States_{\mathsf{M}}$, *and* $l_{\mathsf{M}_{\mathsf{corr},\chi}}(s_{\mathsf{corr}}) := (\infty, \ldots, \infty) \circ (0, \infty)$.
  - *Let* $I =: I' \circ (a, b)$. *If* $(a, b) = (\epsilon, \epsilon)$ *and* $s \neq s_{\mathsf{corr}}$, *then* $\delta_{\mathsf{M}_{\mathsf{corr},\chi}}(s, I) := \delta_{\mathsf{M}_{\mathcal{H},\chi}}(s, I') \circ (\epsilon)$. *Otherwise* $\delta_{\mathsf{M}_{\mathsf{corr},\chi}}(s, I) := (s', O)$ *with* $O := O' \circ (o)$ *is defined as follows:*

    if $a \neq \epsilon$ *and* $s \neq s_{\mathsf{corr}}$ ***then***
      $s' := s_{\mathsf{corr}}$, $o := s$, *and* $O' := (\epsilon, \ldots, \epsilon)$
    else if $s = s_{\mathsf{corr}}$ ***then***
      $o := \iota(I)$
      if $b = \epsilon$ ***then***
        $O' = (\epsilon, \ldots, \epsilon)$
      else
        $(o_1, \ldots, o_t) := \iota^{-1}(b)$ *and* $t' := \mathsf{size}(\mathsf{out}(Ports_{\mathsf{M}_{\mathcal{H},\chi}}))$
        $O'_j := o_j$ *for* $j = 1, \ldots, \mathsf{min}(t, t')$ *and* $O'_j := \epsilon$ *for* $j = t + 1, \ldots, t'$
      end if
    end if

◇

Note that $\mathsf{M}_{\mathsf{corr},\chi}$ is not polynomial-time. Explicitly bounding the number of bits read in a corrupted state independent of the adversary does not capture our intuition of a machine that acts transparently for any adversary, and an explicit bound would severely limit the adversary's capabilities. The fundamental problem is that transparent machines are by definition not polynomial-time but only weakly polynomial-time.

Several extensions are possible: One may extend the corruption responses to two classes of storage, an erasable and a non-erasable one, e.g., to model the different vulnerability of

session keys and long-term keys. This means to refine the state spaces of each machine as a Cartesian product. In- and outputs would be treated like erasable storage. One can also model non-binary corruption requests, e.g., stop requests and requests to corrupt different classes of storage. To model proactive systems [51], one needs repair requests in addition to corruption requests, and appropriate repair responses, e.g., returning to an initial state with only a certain class of storage still intact.

8 Summary

We have presented a rigorous model for secure reactive systems with cryptographic parts in asynchronous networks. Common types of cryptographic systems and their trust models were expressed as special cases of this model, in particular systems with static or adaptive adversaries and with different types of underlying channels. We have defined reactive simulatability as a notion of refinement that retains not only integrity properties but also confidentiality.

As design principles based on this model, we propose to keep specifications of cryptographic systems abstract, i.e., free of all implementation details and deterministic unless the desired functionality is probabilistic by nature, e.g., a common coin-flipping protocol. This allows the cryptographically verified abstractions to be used as building blocks for systems that can be subjected to formal verification. Suitable abstractions sometimes have to explicitly include tolerable imperfections of systems, e.g., leakage of the length of encrypted messages or abilities of the adversary to disrupt protocol runs. In subsequent work, we already showed multiple desired properties of reactive simulatability, in particular composition and property-preservation theorems. We also proved several building blocks like secure message transmission and a Dolev-Yao-style cryptographic library with nested operations. Moreover, we demonstrated the applicability of formal methods over our model by small examples.

Acknowledgment

We thank *Anupam Datta, Martin Hirt, Dennis Hofheinz, Paul Karger, Ralf Küsters, John Mitchell, Jörn Müller-Quade, Andre Scedrov, Matthias Schunter, Victor Shoup, Michael Steiner, Rainer Steinwandt, Dominique Unruh* and an anonymous reviewer for interesting discussions. In particular, joint work with Matthias on examples of synchronous reactive systems also influenced this asynchronous model, and so did joint work with Michael on liveness and adaptiveness. Dennis, Jörn, Rainer, and Dominique pointed out some subtle problems. The anonymous reviewer provided enough motivation for making the model much more rigorous than it was before.

This work was partially supported by the European IST Project MAFTIA. However, it represents the view of the authors. The MAFTIA project was partially funded by the European Commission and the Swiss Federal Office for Education and Science (BBW).

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 *Lecture Notes in Computer Science*, pages 3–22. Springer, 2000.

[4] M. Backes. *Cryptographically Sound Analysis of Security Protocols*. PhD thesis, Universität des Saarlandes, 2002. `http://www.zurich.ibm.com/~mbc/papers/PhDthesis.ps.gz`.

[5] M. Backes and D. Hofheinz. How to break and repair a universally composable signature functionality. IACR Cryptology ePrint Archive 2003/240, Nov. 2003. `http://eprint.iacr.org/`.

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

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

[8] M. Backes and B. Pfitzmann. Computational probabilistic non-interference. In *Proc. 7th European Symposium on Research in Computer Security (ESORICS)*, volume 2502 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2002.

[9] M. Backes and B. Pfitzmann. A cryptographically sound security proof of the Needham-Schroeder-Lowe public-key protocol. In *Proc. 23rd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, 2003.

[10] M. Backes and B. Pfitzmann. Intransitive non-interference for cryptographic purposes. In *Proc. 24th IEEE Symposium on Security & Privacy*, pages 140–152, 2003.

[11] M. Backes, B. Pfitzmann, M. Steiner, and M. Waidner. Polynomial fairness and liveness. In *Proc. 15th IEEE Computer Security Foundations Workshop (CSFW)*, pages 160–174, 2002.

[12] M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations (extended abstract). In *Proc. 10th ACM Conference on Computer and Communications Security*, pages 220–230, 2003. Full version in IACR Cryptology ePrint Archive 2003/015, Jan. 2003, `http://eprint.iacr.org/`.

[13] M. Backes, B. Pfitzmann, and M. Waidner. A general composition theorem for secure reactive systems. In *Proc. 1st Theory of Cryptography Conference (TCC)*, 2004. To appear.

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

[15] D. Beaver and S. Haber. Cryptographic protocols provably secure against dynamic adversaries. In *Advances in Cryptology: EUROCRYPT '92*, volume 658 of *Lecture Notes in Computer Science*, pages 307–323. Springer, 1992.

[16] 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.

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

[18] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13(4):850–864, 1984.

[19] M. Burrows, M. Abadi, and R. Needham. A logic for authentication. Technical Report 39, SRC DIGITAL, 1990.

[20] 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.

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

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

[23] R. Canetti and S. Goldwasser. An efficient threshold public key cryptosystem secure against adaptive chosen ciphertext attack. In *Advances in Cryptology: EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 90–106. Springer, 1999.

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

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

[26] R. Gennaro and S. Micali. Verifiable secret sharing as secure computation. In *Advances in Cryptology: EUROCRYPT '95*, volume 921 of *Lecture Notes in Computer Science*, pages 168–182. Springer, 1995.

[27] O. Goldreich. Secure multi-party computation. Department of Computer Science and Applied Mathematics, The Weizmann Institute of Science, June 1998, revised Version 1.4 October 2002, 1998. `http://www.wisdom.weizmann.ac.il/users/oded/pp.htm`.

[28] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game – or – a completeness theorem for protocols with honest majority. In *Proc. 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 218–229, 1987.

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

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

[31] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–207, 1989.

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

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

[34] D. Hofheinz, J. Müller-Quade, and R. Steinwandt. Initiator-resilient universally composable key exchange. In *Proc. 8th European Symposium on Research in Computer Security (ESORICS)*, volume 2808 of *Lecture Notes in Computer Science*, pages 61–84. Springer, 2003.

[35] R. Impagliazzo and B. M. Kapron. Logics for reasoning about cryptographic constructions. In *Proc. 44th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 372–381, 2003.

[36] R. Kemmerer. Analyzing encryption protocols using formal verification techniques. *IEEE Journal on Selected Areas in Communications*, 7(4):448–457, 1989.

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

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

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

[40] P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. Probabilistic polynomial-time equivalence and security analysis. In *Proc. 8th Symposium on Formal Methods Europe (FME 1999)*, volume 1708 of *Lecture Notes in Computer Science*, pages 776–793. Springer, 1999.

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

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

[43] N. Lynch. I/O automaton models and proofs for shared-key communication systems. In *Proc. 12th IEEE Computer Security Foundations Workshop (CSFW)*, pages 14–29, 1999.

[44] C. Meadows. Using narrowing in the analysis of key management protocols. In *Proc. 10th IEEE Symposium on Security & Privacy*, pages 138–147, 1989.

[45] C. Meadows. Formal verification of cryptographic protocols: A survey. In *Proc. ASIACRYPT '94*, volume 917 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1994.

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

[47] J. K. Millen. The interrogator: A tool for cryptographic protocol security. In *Proc. 5th IEEE Symposium on Security & Privacy*, pages 134–141, 1984.

[48] J. Mitchell, M. Mitchell, and A. Scedrov. A linguistic characterization of bounded oracle computation and probabilistic polynomial time. In *Proc. 39th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 725–733, 1998.

[49] J. Mitchell, M. Mitchell, A. Scedrov, and V. Teague. A probabilistic polynominal-time process calculus for analysis of cryptographic protocols (preliminary report). *Electronic Notes in Theoretical Computer Science*, 47:1–31, 2001.

[50] J. Neveu. *Mathematical Foundations of the Calculus of Probability*. Holden-Day, 1965.

[51] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks. In *Proc. 10th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 51–59, 1991.

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

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

[54] B. Pfitzmann and M. Waidner. A general framework for formal notions of "secure" systems. Research Report 11/94, University of Hildesheim, Apr. 1994. `http://www.semper.org/sirene/lit/abstr94.html#PfWa_94`.

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

[56] 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. Extended version in Cryptology ePrint Archive, Report 2000/066, `http://eprint.iacr.org/`.

[57] R. Segala and N. Lynch. Probabilistic simulation for probabilistic processes. In *Proc. 5th International Conference on Concurrency Theory (CONCUR)*, volume 836 of *Lecture Notes in Computer Science*, pages 481–497. Springer, 1994.

[58] R. Segala and N. Lynch. Probabilistic simulation for probabilistic processes. *Nordic Journal of Computing*, 2(2):250–273, 1995.

[59] M. Steiner. *Secure Group Key Agreement*. PhD thesis, Universität des Saarlandes, 2002. `http://www.semper.org/sirene/publ/Stei_02.thesis-final.pdf`.

[60] F. J. Thayer Fabrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Why is a security protocol correct? In *Proc. 19th IEEE Symposium on Security & Privacy*, pages 160–171, 1998.

[61] 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.

[62] A. C. Yao. Protocols for secure computations. In *Proc. 23rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 160–164, 1982.

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