Polynomial Liveness*

Michael Backes Birgit Pfitzmann Michael Waidner

IBM Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland
{mbc,bpf,wmi}@zurich.ibm.com

Michael Steiner

IBM T.J. Watson Research Center
Yorktown Heights, NY 10598, USA
msteiner@watson.ibm.com

Abstract

Important properties of many protocols are liveness or availability, i.e., that something good happens now and then. In asynchronous scenarios, these properties depend on the scheduler, which is usually considered to be fair in this case. The standard definitions of fairness and liveness are based on infinite sequences. Unfortunately, this cannot be applied to most cryptographic protocols since one must restrict the adversary and the runs as a whole to length polynomial in the security parameter. We present the first general definition of polynomial fairness and liveness in asynchronous scenarios which can cope with cryptographic protocols. Furthermore, our definitions provide a link to the common approach of simulatability which is used throughout modern cryptography: We show that polynomial liveness is maintained under simulatability. As an example, we present an abstract specification and a secure implementation of secure message transmission with reliable channels, and prove them to fulfill the desired liveness property, i.e., reliability of messages.

1 Introduction

When we analyze arbitrary protocols, two important classes of properties to consider are liveness and safety [1]. Informally, a liveness property states that something good eventually happens. A common problem in asynchronous scenarios is that liveness depends on the scheduler: If the scheduler never schedules certain messages, the good event cannot happen. The standard solution (see, e.g., [8]) is to concentrate on so-called fair schedulers. Roughly, those guarantee that every message is delivered at some point in time in every infinite run of the system.

*A preliminary version of this article appeared in the Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW), pages 160–174.

For most cryptographic protocols, definitions based on infinite runs cannot be used since one must restrict the adversary and the runs as a whole to polynomial and, therefore, finite length. By polynomial we mean here, as well as in the sequel, polynomial in the security parameter k.

Another problem in the cryptographic case is that one typically assumes that an adversary can modify messages arbitrarily in transit. For those cases, one can certainly not require that anything good happens (e.g., the parties agree on a key [10]). Nevertheless, even for protocols that do consider such arbitrary corruptions, one typically wants a guarantee of the following form: *If* certain messages get through unmodified, then certain good things happen. For the other cases, e.g., the honest participants make progress but do not get each others' messages, one may still want at least local termination (in the sense of a timeout message, or a positive reaction to an abort by the user). In order to cope with these problems, we introduce the notion of *polynomial fairness* and *polynomial liveness*. Polynomial fairness states that the scheduler will schedule each message after a polynomially bounded number of steps. Now, polynomial liveness captures that something good will happen after a polynomial number of steps subject to the condition that the considered scheduler is polynomially fair.

We are aware of only one other approach that handles polynomial liveness properties for security protocols: Cachin et al. presented an elegant solution for a specific protocol, asynchronous Byzantine agreement in [6]. They show that an adversary cannot make the honest users (altogether) generate a super-polynomial number of messages for any particular subprotocol run and that the protocol ensures "deadlock-freeness", i.e., some progress will eventually be made. However, their definition is limited to this specific protocol, and applying the approach more generally presumes a fixed number of subprotocols and the use of session IDs (so that one can say "all messages associated to an event have been delivered"). We aim at a more general definition. Thus, the fact that certain messages get through is defined by letting the system "run empty", i.e., we consider one particular point in time, so that neither the honest user nor the adversary produce any outputs after that time. From then on, the only active machines are the scheduler and the internal machines of the system. The scheduler can then deliver all messages that have already passed through the adversary (i.e., have not been interrupted in transit), all messages that have been sent over reliable channels, etc. We speak of polynomial liveness if the good event happens in a polynomially bounded number of steps of the honest user counted from the beginning of the "run-empty phase". This models our intuition that in real life, every message which has not been interrupted by the adversary will eventually be delivered to its recipient.

To the best of our knowledge, the approach of letting the system run empty has not been used before to prove liveness for security protocols. However, if we leave the security community and take a look at the verification of microprocessors we meet similar techniques. Roughly speaking, Burch and Dill showed in [5] that certain safety properties of a pipelined microprocessor hold by letting the pipelines run empty now and then, which they denoted as "flushing the pipeline". In the essence, their approach is quite similar to ours, but applied to a completely different problem.

Moreover, we will show that our definition of liveness behaves well under the concept of simulatability which has asserted its position as a fundamen-

tal concept of cryptography. Precisely, we show that liveness properties are preserved under simulatability under certain circumstances, i.e., liveness properties proved for abstract specifications automatically carry over to the concrete implementations in this case.

As an example fitting our definition, we present an abstract specification and a concrete implementation of reliable secure message transmission. The reliability is considered as the desired liveness property, i.e., roughly speaking, every message sent will eventually be delivered. We prove this liveness property for the abstract specification and transfer it to the implementation with the preservation theorem.

Outline of the paper. In Section 2 we review the underlying model from [11]. In Section 3 we present our notion of polynomial fairness and polynomial liveness, starting with the basic intuition and moving on to the rigorous definitions. In Section 4 we show that liveness is preserved under simulatability. The example of reliable cryptographic channels is presented in Section 5. Section 6 summarizes the results.

2 The Model for Reactive Systems

In this section, we recapitulate the model for asynchronous probabilistic reactive systems as introduced by Pfitzmann and Waidner in [11].

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

In particular, we repeat the scheduling model in full detail because it is important for the fair schedulers. The specific scheduling aspects needed for cryptographic asynchronous systems are that schedulers are "normal" system machines, so that they schedule with realistic knowledge, and that different channels may be scheduled by different machines, e.g., so that local submachines can be represented. This is also useful flexibility for liveness because we only need to let the fair scheduler schedule certain important channels.

2.1 General System Model

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

Communication between different machines is done via ports using messages composed from an alphabet Σ. Inspired by the CSP-Notation [7], we write output and input ports as $\mathsf{q}!$ and $\mathsf{q}?$ respectively. The ports of a machine M are denoted by $\mathsf{ports}(\mathsf{M})$. The subset of input and output ports are denoted by $\mathsf{in}(\mathsf{ports}(\mathsf{M}))$ and $\mathsf{out}(\mathsf{ports}(\mathsf{M}))$, respectively. Channels are defined implicitly by naming convention, that is port $\mathsf{q}!$ sends messages to $\mathsf{q}?$. To achieve asynchronous timing, a message is not directly sent to its recipient, but it is first stored in a special machine $\widetilde{\mathsf{q}}$ called a buffer and waits to be scheduled. If a machine wants to schedule the i-th message of buffer $\widetilde{\mathsf{q}}$ (this machine must have the unique clock-out port $\mathsf{q}^{\triangleleft}!$) it simply sends i at $\mathsf{q}^{\triangleleft}!$, see Figure 1. The buffer then schedules the i-th message and removes it from its internal list. In our

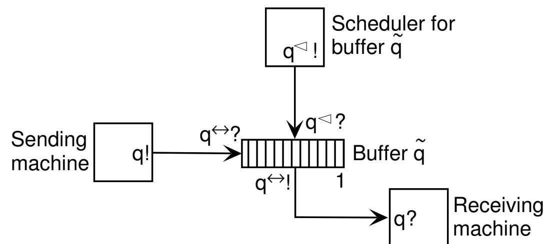

Figure 1: Ports and buffers.

case, most buffers are either scheduled by a master scheduler or the adversary, i.e., one of those has the clock-out port. Note that in [11] the adversary and the master scheduler are mostly the same entity. This gives the adversary complete control over the overall scheduling of network traffic and models the worst-case behavior we usually have to expect in an asynchronous system. However, to consider liveness we have to make certain assumptions on the well-behavior of scheduling. Therefore, we separate it here into two entities, a master scheduler enforcing our assumptions and an adversary with arbitrary behavior.

After introducing ports, we now focus on the definition of machines. Our machine model is probabilistic state-transition machines, similar to probabilistic I/O automata as sketched by Lynch [8]. If a machine is switched, it receives an input tuple at its input ports and performs its transition function yielding a new state and an output tuple in the deterministic case, or a finite distribution over the set of states and possible outputs in the probabilistic case. At each switching step of one particular machine, at most one value can arrive at every input port and the machine can produce at most one output per port. Furthermore, each machine has a bound on the length of the considered inputs which allows time bounds independent of the environment.

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

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

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

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

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

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

Definition 2.2 *(Runs) Given a closed collection $\hat{C}$ with master scheduler X and a tuple $ini \in Ini_{\hat{C}} := \times_{\mathsf{M}\in\hat{C}} Ini_{\mathsf{M}}$ of initial states, the probability space of* runs *is defined inductively by the following algorithm. It has a variable r for the resulting run, an initially empty list, a variable $\mathsf{M_{CS}}$ ("current scheduler") over machine names, initially $\mathsf{M_{CS}} := \mathsf{X}$, and treats each port as a variable over Σ^*, initialized with ϵ except for $\mathsf{clk}^{\triangleleft}? := 1$. Probabilistic choices only occur in Step (1).*

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

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

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

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

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

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

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

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

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

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

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

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

2.2 Security-specific System Model

For security purposes, special collections are needed, because an adversary may have taken over parts of the initially intended system. Therefore, a system consists of several possible remaining structures. An example of the typical derivation of these structures from an intended structure and a trust model will be seen in Section 5. First, the system part is defined and then the environment, consisting of users and adversaries.

Definition 2.4 *(Structures and Systems)*

a) *A* structure *is a pair $struc = (\hat{M}, S)$ where $\hat{M}$ is a collection of simple machines (i.e., with only normal in- and output ports and clock-out ports) called* correct machines*, and $S \subseteq \mathsf{free}([\hat{M}])$ is called* specified ports*. If $\hat{M}$ is clear from the context, let $\bar{S} := \mathsf{free}([\hat{M}]) \setminus S$. We call $\mathsf{forb}(\hat{M}, S) := \mathsf{ports}(\hat{M}) \cup \bar{S}^c$ the* forbidden ports.

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

$\diamond$

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

A structure can be completed to a *configuration* by adding machines H and A, modeling the joint honest users and the adversary, respectively. The machine

H is restricted to the specified ports S, A connects to the remaining free ports of the structure and both machines can interact, e.g., in order to model active attacks.

Definition 2.5 *(Configurations)*

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

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

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

$\diamond$

We only briefly state here that several machines can be combined into one single machine (which has the original machines as submachines), cf. [11] for more details. Moreover, the view of every submachine remains unchanged by this combination. Hence, we can consider configurations with a *set* of users instead of a single user-machine H as well, and the upcoming definitions work as well with these modified configurations.

2.3 Defining Security with Simulatability

As we will see below, the system model provides a powerful instrument to compare two systems and to assess whether one system securely implements another one. Based on this, our approach in defining security is as follows: (1) We define the abstract specification of a secure service as an ideal system Sys_{id} consisting of a single machine TH. Given the simplicity of the idealized machine, the correctness of the specification is often intuitively clear. Furthermore, we can gain additional confidence by analyzing TH using formal methods and automated tools [3]. The power of the model allows us to specify arbitrary integrity and strong confidentiality properties. (2) Given any concrete real system Sys_{real} implementing the desired service, we then prove its security by showing that it securely implements Sys_{id}.

The definition of one system securely implementing another one is based on the common concept of *simulatability.* The notion of simulatability was introduced in [12] and has asserted its position as a fundamental concept of modern cryptography. Simulatability essentially means that whatever might happen to an honest user in a concrete system Sys_{real} can also happen in an ideal system Sys_{id}. As by definition only good things can happen in the ideal

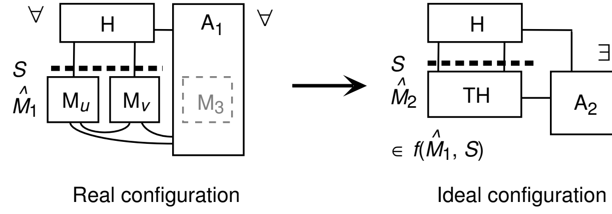

Figure 2: Example of simulatability. The view of H is compared.

system, simulatability guarantees that no bad things can happen in the real system. More precisely, for every configuration $conf_1 \in \mathsf{Conf}(Sys_{\mathsf{real}})$, there exists a configuration $conf_2 \in \mathsf{Conf}(Sys_{\mathsf{id}})$ yielding indistinguishable views of the same user in both configurations. We abbreviate this by $Sys_{\mathsf{real}} \geq_{\mathsf{sec}} Sys_{\mathsf{id}}$ and we say that Sys_{real} is "at least as secure" as the system Sys_{id}. A typical situation is illustrated in Figure 2.

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

The upcoming simulatability definition is based on indistinguishability of views.

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

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

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

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

(as a function of k*). SMALL should be closed under addition, and with a function* g *also contain every function* $g' \leq g$*. Typical classes are EXPSMALL containing all functions bounded by* $Q(k) \cdot 2^{-k}$ *for a polynomial* Q*, and the (larger) class NEGL, cf. part c) of the definition.*

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

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

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

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

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

We now present the simulatability definition.

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

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

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

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

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

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

3 Expressing Polynomial Fairness and Liveness

In this section, we introduce our definitions of polynomial fairness and polynomial liveness in asynchronous reactive systems. At first, we concentrate on fairness.

3.1 Polynomial Fairness

Usually, a scheduler is called *fair* if it schedules every process infinitely often unless this process is only finitely often enabled (see, e.g., [8]). As we already stated in the introduction, this definition is not suited for most cryptographic protocols, since both the adversary and the honest user are polynomially bounded, and, therefore the runs are finite.

What we would like to express is that an enabled process will be scheduled by the master scheduler X after at most $J(k)$ of the scheduler's steps where $J(k)$ is a polynomial in the security parameter k. This should hold all the time unless the scheduler reaches its runtime bound. We start with an intuitive description of how this can be formalized.

Starting from the t-th switching step of X in one particular run r, we search for the first future scheduling step $m > t$ of X such that the first message of $\widetilde{\mathsf{q}}$ is scheduled. Thus, if the buffer is non-empty, a message will be scheduled from it. Moreover, we always demand that it is the first clock-out port with non-empty value (so it will not be ignored by the run algorithm, cf. Definition 2.2).

We denote this number of switching steps of X (i.e., $m - t$) by $\mathsf{wait}(t, r, \mathsf{q}^{\triangleleft}!)$. Moreover, in order to cope with the final state of X, we explicitly define this number to be infinite if the master scheduler never enters a final state, and if there exists no such output at $\mathsf{q}^{\triangleleft}!$. If the master scheduler enters a final state after its m-th switching step without ever outputting 1 at $\mathsf{q}^{\triangleleft}!$ we define this number to be $m - t$.

We denote the scheduler as J-fair for a function $J : \mathbb{N} \to \mathbb{N}$ if the maximum of these waiting times is bounded by J as a function of k.

Moreover, we demand that X does not connect to an unspecified port of the system, i.e., we have $\mathsf{ports}(\mathsf{X}) \cap \mathsf{forb}(\hat{M}, S) = \emptyset$. This condition is essential for relating the definition of polynomial fairness to simulatability, since the master scheduler can be defined as part of the honest user in this case, and hence, can remain unchanged in simulatability. However, note that this restriction does not limit the expressive power of the model.

Let us now turn to the formal definition.

Definition 3.1 *(Polynomial Fairness) Let an arbitrary system Sys be given. Then a master scheduler* X *is called* J-fair *for a structure* $(\hat{M}, S) \in \mathit{Sys}$ *and a function* $J : \mathbb{N} \to \mathbb{N}$ *if and only if the following holds:*

- X *does not connect to an unspecified port of the system, i.e., we have* $\mathsf{ports}(\mathsf{X}) \cap \mathsf{forb}(\hat{M}, S) = \emptyset$.
- *Let* $O_{t,r}$ *and* $S_{t,r}$ *be* X*'s output and state after the t-th switching of* X*, respectively, in every considered run r of the configuration. Moreover, let* $\mathit{Fin}_{\mathsf{X}}$ *denote the set of final states of* X *and let the predicate* $O_{t,r}\lceil_{\mathsf{q}^{\triangleleft}!}$ *denote whether the buffer* $\widetilde{\mathsf{q}}$ *will be scheduled in the next round. Now we define*

  $$\mathsf{wait}(t, r, \mathsf{q}^{\triangleleft}!) := \infty$$

  if for all $m \in \mathbb{N}$: $O_{t+m,r}\lceil_{\mathsf{q}^{\triangleleft}!} \neq 1$ *and* $S_{t+m,r} \notin \mathit{Fin}_{\mathsf{X}}$*, and otherwise*

  $$\mathsf{wait}(t, r, \mathsf{q}^{\triangleleft}!) := \mathsf{min}_{m \in \mathbb{N}}\{O_{t+m,r}\lceil_{\mathsf{q}^{\triangleleft}!} = 1 \vee S_{t+m,r} \in \mathit{Fin}_{\mathsf{X}}\}.$$

  Then we require that

  $$\mathsf{wait}(t, r, \mathsf{q}^{\triangleleft}!) \leq J(k)$$

  holds for the security parameter k, all runs r of the configuration, all $t \in \mathbb{N}$ *and* $\mathsf{q}^{\triangleleft}! \in \mathsf{ports}(\mathsf{X})$.

If a master scheduler is J-fair for a polynomial J, we call it polynomially fair*.* $\diamond$

Remark 3.1. Our definition of J-fairness implies the common definition of fairness based on infinite sequences (i.e., processes which are infinitely often enabled have to be scheduled infinitely often). By "a process or machine M is enabled", we define that an ingoing buffer $\widetilde{\mathsf{q}}$ of the machine M has non-empty content, and

the master scheduler could schedule this buffer at the moment (i.e., the master scheduler is switched, and it has the corresponding clock-out port $\mathsf{q}^{\triangleleft}!$ for scheduling this buffer). Now according to our definition, this buffer will always be scheduled after at most $J(k)$ steps, i.e., after a finite number of steps. Moreover, if the machine is enabled infinitely often, then the master scheduler must also be switched infinitely often, so it will schedule the corresponding buffer, and therefore the machine infinite times. ◦

3.2 Polynomial Liveness

After introducing the notion of polynomially fair master schedulers, we can now turn our attention to expressing polynomial liveness. Intuitively speaking, polynomial liveness means that some good things will happen after a polynomial number of steps of the honest user.

However, we cannot expect this to hold for arbitrary configurations. Imagine an honest user and an adversary which are communicating over the system all the time, without ever giving control to the (fair) master scheduler. In this case we cannot guarantee that good things happen since the fairness condition of the master scheduler is irrelevant. Instead, we define that certain good things happen if the system is "run empty". More precisely, we consider situations where neither the user nor the adversary produce any outputs any further from one particular point in time. We then require that there exists a polynomial Q_{H} such that the good event will happen in at most $Q_{\mathsf{H}}(k)$ switching steps of H, counted from that special point in time.

Thus, in order to define polynomial liveness for the overall system, we restrict ourselves to those configurations which prevent outputs of both the user and the adversary after a particular number of switching steps of H. More precisely, we let the honest user count the number of times it has been switched. If it reaches the "critical" time $t_{\mathsf{stop}}(k)$, it outputs a command (stop) to the adversary and both machines do not produce outputs any further. We call these configurations *liveness configurations.*

From that special point in time, the master scheduler and the machines of the system are the only active machines in the configuration, so the master scheduler can try to empty the system, e.g., to deliver those messages that have already passed through the adversary and now wait to be scheduled to their recipient, and those that have been sent over reliable channels. In the real world, this models that all messages that the adversary has already let through will eventually be delivered, just what we expect if we consider sending messages over the net: If the adversary explicitly let the message through (or he does not, or cannot, interrupt the transmission) then the message will be delivered.

However, there is one more problem we have to take care of. Clearly, we cannot expect the event to happen if the master scheduler or the honest user enter a final state too early. Therefore, we assume that the master scheduler and the honest user run sufficiently long, i.e., we augment our definition of liveness configurations with lower bounds Q_{X}, Q_{H} on the number of switching steps of the master scheduler and the honest user.

Definition 3.2 *(Liveness Configuration) Let an arbitrary system Sys and four functions* $t_{\mathsf{stop}}, J, Q_{\mathsf{X}}, Q_{\mathsf{H}} \colon \mathbb{N} \to \mathbb{N}$ *be given. Furthermore, let a configuration* $conf = (\hat{M}, S, \{\mathsf{H}\} \cup \{\mathsf{X}_{\mathsf{fair}}\}, \mathsf{A}) \in \mathsf{Conf}(Sys)$ *be given where* $\mathsf{X}_{\mathsf{fair}}$ *is a J-fair*

scheduler for $(\hat{M}, S)$. *We call this configuration a* (t_{stop}, J, Q_{X}, Q_{H})-liveness configuration *if the following holds:*

- *The honest user* H *has special ports* $stop_{\mathsf{H}}!$, $stop_{\mathsf{H}}{}^{\triangleleft}!$ *which are connected to the adversary, i.e.,* $\{stop_{\mathsf{H}}!, stop_{\mathsf{H}}{}^{\triangleleft}!\} \subseteq \mathsf{ports}(\mathsf{H})$ *and* $stop_{\mathsf{H}}? \in \mathsf{ports}(\mathsf{A})$.
- *The user* H *has an internal counter count over the naturals initialized with* 0. *The user first increases this counter every time it switches, and then checks whether the counter equals* t_{stop} *(k). In this case it outputs* (stop) *at* $stop_{\mathsf{H}}!$, 1 *at* $stop_{\mathsf{H}}{}^{\triangleleft}!$. *From now on, the user only reads its inputs, but no longer produces outputs. Similarly, if the adversary gets an input* (stop) *at* $stop_{\mathsf{H}}?$, *it only reads its inputs in the future without producing any outputs.*
- *The user* H *does not output anything at* $stop_{\mathsf{H}}!$ *except in the above case.*
- *The number of switching steps of the master scheduler and the honest user is lower-bounded by* $Q_{\mathsf{X}}(k)$ *and* $Q_{\mathsf{H}}(k)$, *respectively, for every run of the configuration.*

Liveness configurations will be denoted by $conf^{\mathsf{live}} = (\hat{M}, S, \{\mathsf{H}\} \cup \{\mathsf{X}_{\mathsf{fair}}\}, \mathsf{A})^{\mathsf{live}}$ *and the set of all liveness configurations of a system Sys by* $\mathsf{Conf}^{\mathsf{live}}(Sys)$. ◇

Thus, for a given run r of a (t_{stop}, J, Q_{X}, Q_{H})-liveness configuration, we will not have any further outputs of both the user and the adversary after the $t_{\mathsf{stop}}(k)$-th switching of the user. We use this point in time to split the run into two parts, a prefix $r_{\leq i}$ and a tail $r_{>i}$, so that we obtain $r = r_{\leq i} \circ r_{>i}$. The tail $r_{>i}$ is called the *extended run* or the *run-empty phase*.

We can now turn our attention to the actual definition of polynomial liveness.

Definition 3.3 *(Polynomial Liveness Properties) A polynomial liveness property of a structure* $(\hat{M}, S) \in Sys$ *consists of three components:*

1. *First, we have an integrity property Req (i.e., the good event which we would like to happen) which is represented as a function that assigns to each set* S *with* $(\hat{M}, S) \in Sys$ *a set of runs at the ports in* S. *Informally speaking,* $Req(S)$ *states which are the "good" runs for the given structure* $(\hat{M}, S)$. *More precisely, such a run is a sequence* $(v_i)_{i \in I}$ *of values over port names and* Σ^* *with* $I = \{1, \ldots, l\}$ *for* $l \in \mathbb{N}$ *or* $I = \mathbb{N}$, *i.e., sets of port-value pairs so that* v_i *is of the form* $v_i := \bigcup_{p \in S'} \{p : v_{p,i}\}$ *for a subset* $S' \subseteq S$ *and* $v_{p,i} \in \Sigma^*$. *For more details on how integrity properties are expressed and how they behave under simulatability in the asynchronous case, we refer the reader to [2].*
2. *The second component* $\{\mathsf{p}_1{}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\}$ *is a subset of the complement of the specified ports of the structure, i.e.,* $\{\mathsf{p}_1{}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\} \subseteq S^c$. *It indicates which ports have to be scheduled by the fair master scheduler such that the event Req will eventually happen.*
3. *The third component is a function* $t_s \colon \mathbb{N} \to \mathbb{N}$. *Intuitively, a system can only run empty and finally fulfill the desired property if it has not yet run for too long. In this case, it might exceed its runtime bounds during the extended run of the configuration before the event Req occurs. Therefore, we have to bound the point in time at which the extended run may begin.*

Obviously, the runtime of the system depends on the security parameter k, so this bound is represented as a function of k, too.

Finally, a liveness property of a system Sys is a mapping

$$\begin{aligned} \varphi_{Sys} : Sys &\rightarrow LiveProp \\ (\hat{M}, S) &\mapsto (Req, \{\mathsf{p_1}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\}, t_s)_{(\hat{M},S)} \end{aligned}$$

that assigns each structure $(\hat{M}, S)$ a liveness property which is defined on $(\hat{M}, S)$. $\diamond$

Remark 3.2. This definition might look surprising at first sight as the polynomial-time aspects are somewhat missing. In particular, *Req* contains potentially arbitrarily long runs and basically corresponds to runs of a general liveness property. However, the purpose of the polynomial liveness property is only to list *all* "good" traces and define some technicalities. The polynomial-time aspects and restrictions come mainly into play in the following definition of fulfillment. $\circ$

After introducing what polynomial liveness properties are, we have to define what it means that a system fulfills them. Essentially, our subsequent definition states that a structure fulfills the liveness property $(Req, \{\mathsf{p_1}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\}, t_s)$ if the following holds: If the ports $\{\mathsf{p_1}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\}$ are scheduled by a J-fair master scheduler for an arbitrary polynomial J, and if we do not proceed too far in time (i.e., we only consider t_{stop}-liveness configurations for $t_{\mathsf{stop}}(k) < t_s(k)$) then there are polynomials Q_{X}, Q_{H} such that the event *Req* will happen within a polynomial number of switching steps of the honest user.

This will be expressed by using prefixes of the whole run restricted to those ports that connect to the honest user in the considered configuration, i.e., we write $run_{conf,k,l(k)}\lceil_{S^{\mathsf{H}}}$ with $S^{\mathsf{H}} := \{\mathsf{q} \in S \mid \mathsf{q}^c \in \mathsf{ports}(\mathsf{H})\}$ and a polynomial l (cf. Definition 2.3). In slight abuse of notation, we write $Req(S^{\mathsf{H}})$ instead of $Req(S)\lceil_{S^{\mathsf{H}}}$, i.e., we restrict the run to the ports in S^{H}.

A system fulfills the overall liveness property if all of its structures fulfill their liveness properties. Moreover, we will see that there are different grades of fulfillment. We distinguish between *perfect*, *statistical* and *computational* fulfillment depending on whether the good event will always happen, or only with overwhelming probability, i.e., the probability of failure should be statistically small or negligible in polynomial-time configurations, respectively.

There is one more technicality that has to be taken care of. Consider a system that initially needs several steps for distributing keys. Now an integrity property might state that certain properties hold subject to the condition that the keys have already been distributed. However, keeping the polynomials Q_{H} and Q_{X} small ensures that each run will always stop before the key distribution can be completed, hence any system would then fulfill this property. Obviously, a meaningful definition of liveness should exclude this. The solution is that both polynomials have to be chosen large enough such that the integrity property indeed could be violated. For an integrity property *Req* for a structure $(\hat{M}, S)$, we hence define $\mathsf{min}(Req, S) := \mathsf{min}(|I| \mid v = (v_i)_{i \in I} \wedge v \notin Req(S)\}$, with v_i as in Definition 3.3.

Definition 3.4 *(Fulfillment of Polynomial Liveness) Let an arbitrary system Sys and a polynomial liveness property φ_{Sys} for Sys be given. Then a structure $(\hat{M}, S) \in Sys$ fulfills its polynomial liveness property $LiveReq := (Req, \{\mathsf{p}_1{}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\}, t_s) := \varphi_{Sys}(\hat{M}, S)$*

- ***perfectly*** *($(\hat{M}, S) \models^{\mathsf{perf}} LiveReq$) iff $\forall$ polynomials J, t_{stop} with $t_{\mathsf{stop}} < t_s$ $\exists$ polynomials Q_{X}, $Q_{\mathsf{H}} \geq \mathsf{min}(Req, S)$ such that for all (t_{stop}, J, Q_{X}, Q_{H})-liveness configurations $conf_{\mathsf{live}} = (\hat{M}, S, \{\mathsf{H}\} \cup \{\mathsf{X}_{\mathsf{fair}}\}, \mathsf{A})^{\mathsf{live}} \in \mathsf{Conf}^{\mathsf{live}}(Sys)$ with $\{\mathsf{p}_1{}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\} \subseteq \mathsf{ports}(\mathsf{X}_{\mathsf{fair}})$ the following holds: all $Q_{\mathsf{H}}(k)$-prefixes of the restriction of the run to the ports in S^{H} lie in $Req(S^{\mathsf{H}})$. In formulas,*

  $$[(run_{conf_{\mathsf{live}}, k, Q_{\mathsf{H}}(k)} \lceil_{S^{\mathsf{H}}})] \subseteq Req(S^{\mathsf{H}})$$

  for all k, where $[\cdot]$ denotes the carrier set of a probability distribution.

- ***statistically*** *($(\hat{M}, S) \models^{SMALL} LiveReq$) iff $\forall$ polynomials J, t_{stop} with $t_{\mathsf{stop}} < t_s$ $\exists$ polynomials Q_{X}, $Q_{\mathsf{H}} \geq \mathsf{min}(Req, S)$ such that for all (t_{stop}, J, Q_{X}, Q_{H})-liveness configurations $conf_{\mathsf{live}} = (\hat{M}, S, \{\mathsf{H}\} \cup \{\mathsf{X}_{\mathsf{fair}}\}, \mathsf{A})^{\mathsf{live}} \in \mathsf{Conf}^{\mathsf{live}}(Sys)$ with $\{\mathsf{p}_1{}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\} \subseteq \mathsf{ports}(\mathsf{X}_{\mathsf{fair}})$ the following holds: the probability that $Req(S^{\mathsf{H}})$ is not fulfilled after $Q_{\mathsf{H}}(k)$ steps is small, i.e.,*

  $$P(run_{conf_{\mathsf{live}}, k, Q_{\mathsf{H}}(k)} \lceil_{S^{\mathsf{H}}} \notin Req(S^{\mathsf{H}})) \in SMALL.$$

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

- ***computationally*** *($(\hat{M}, S) \models^{\mathsf{poly}} LiveReq$) iff $\forall$ polynomials J, t_{stop} with $t_{\mathsf{stop}}(k) < t_s(k)$ $\exists$ polynomials Q_{X}, $Q_{\mathsf{H}} \geq \mathsf{min}(Req, S)$ such that for all polynomial-time (t_{stop}, J, Q_{X}, Q_{H})-liveness configurations with $\{\mathsf{p}_1{}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\} \subseteq \mathsf{ports}(\mathsf{X}_{\mathsf{fair}})$ the following holds: The probability, that $Req(S^{\mathsf{H}})$ is not fulfilled after $Q_{\mathsf{H}}(k)$ steps is negligible, i.e.,*

  $$P(run_{conf_{\mathsf{live}}, k, Q_{\mathsf{H}}(k)} \lceil_{S^{\mathsf{H}}} \notin Req(S^{\mathsf{H}})) \in NEGL.$$

We write $(\hat{M}, S) \models LiveReq$ if we want to treat all three cases together.

Finally, a system Sys fulfills a liveness property φ_{Sys} perfectly, statistically, or computationally iff each $(\hat{M}, S) \in Sys$ fulfills $\varphi_{Sys}(\hat{M}, S)$ perfectly, statistically, or computationally. In this case, we write $Sys \models^{\mathsf{perf}} \varphi_{Sys}$, etc. $\diamond$

Remark 3.3. Our definition of polynomial liveness (fulfillment) closely resembles the common definition of liveness, i.e., for any finite run prefix r_1 there exists an extended run r_2 such that $r_1 \circ r_2$ is a valid run of the system. It should become clear if we ignore for a moment the polynomials J, Q_{X}, and Q_{H}, which are technicalities necessary to deal with the underlying system model, and focus on $t_{\mathsf{stop}}(k)$. By all-quantifying over all $t_{\mathsf{stop}}(k)$ and considering the point in time when H sends (stop) to A, we consider all, in our case not only finite but also polynomially-bounded, prefixes r_1. Similar to the common definition, our definition also guarantees valid runs by requiring that $r_1 \circ r_2$ must be contained in $Req(S^{\mathsf{H}})$ for all extended runs r_2. $\circ$

Remark 3.4. Our definition requires that a user will never be able to produce any outputs again after exceeding the time $t_{\mathsf{stop}}(k)$. In real life, the user will usually resume outputting messages after the good event has happened, e.g., to send reply-messages. Our definitions could be modified such that both the honest user and the adversary are "switched on" again after the good event has happened. However, on the one hand, this task is tedious because it significantly complicates our definitions and the preservation theorem of Section 4. On the other hand, this extension is also not really necessary. The "run-empty" phase is a conceptual artifact to express that at any given point in time t (the point H sends (stop)), the good events could happen eventually when progress can be guaranteed, e.g., the network schedules fairly. Insofar, it looks rather unnatural to "switch-on" adversaries and users afterwards. Furthermore, our model already covers "continuous liveness" implicitly. Take a particular liveness configuration where we show that the good thing has happened after a trigger at time $t_{\mathsf{stop}}(k)$ and where we would desire a service resumption, e.g., for a reply. There will be another similar liveness configuration which differs only as follows: (1) $t_{\mathsf{stop}}(k)$ is replaced by a larger $t'_{\mathsf{stop}}(k)$; (2) H does not send (stop) at time $t_{\mathsf{stop}}(k)$ but H (and A) behave like (stop) would have been sent until the good thing has happened; (3) From then on A behave arbitrarily awaiting the real (stop), and H restarts its activity, e.g., by sending a reply, and eventually at time $t'_{\mathsf{stop}}(k)$ sends the real $t_{\mathsf{stop}}(k)$. Clearly, this configuration provides exactly the extended service, i.e, sending a reply. Furthermore, by essentially all-quantifying over all H, A and $t_{\mathsf{stop}}(k)$, our definition should also guarantee that our system provides liveness in delivering the reply. ◦

4 Preservation of Polynomial Liveness under Simulatability

In this section, we show that our definition of polynomial liveness behaves well under simulatability under certain circumstances. Usually, defining a cryptographic system starts with an abstract specification stating what the system should do. After that, this specification can be refined stepwise with respect to simulatability, which finally yields a secure implementation. Such a specification is usually monolithic, i.e., it consists of only one idealized machine which does not contain any probabilism and has the desired properties by construction. Thus, it can be validated by formal proof systems, at least if is not too complex. At this time, we may wonder whether or not the verification of these properties made for the ideal specification carries over to the concrete implementation. This is essential for modular proofs. We can answer this question in the affirmative under reasonable assumptions yielding the preservation theorem presented below.

In the following, we assume two systems Sys_1, Sys_2 to be given, such that $Sys_1 \geq^f Sys_2$ holds for a valid mapping f. Moreover, we assume that Sys_2 fulfills an arbitrary liveness property φ_{Sys_2}. Unfortunately, we cannot expect the liveness property to automatically carry over to Sys_1 if both systems are completely unrestricted. Let us assume that we are given a valid $(t_{\mathsf{stop}},\cdot,\cdot,\cdot)$-liveness configuration[1] $conf_1^{\mathsf{live}} \in \mathsf{Conf}^{\mathsf{live}}(Sys_1)$. Therefore, there has to be an

[1] As the last three parameters are of no importance in this discussion, we omit them here.

indistinguishable configuration $conf_2 \in \mathsf{Conf}(Sys_2)$ because of $Sys_1 \geq^f Sys_2$. However, it is clear that $conf_2$ is not necessarily a $(t_{\mathsf{stop}},\cdot,\cdot,\cdot)$-liveness configurations again, since the adversary is not forced to stop outputting messages at that particular point in time; in fact, he is not forced to do so at all. Given this observation, we have to restrict our attention to those systems in which simulatability respects $(t_{\mathsf{stop}},\cdot,\cdot,\cdot)$-liveness configurations, i.e., simulatability yields indistinguishable $(t_{\mathsf{stop}},\cdot,\cdot,\cdot)$-liveness configurations by construction. We speak of *liveness simulatability* in this case.

At first glance, this seems to be a quite severe restriction to the considered set of possible systems. However, indistinguishable configurations are typically derived using the simulatability variant of *blackbox* simulatability. This means that the adversary A' of the indistinguishable configuration is derived by the original adversary A and a simulator Sim which is inserted between the original adversary and the system. This does not change the communication between A and the honest user, so A' handles incoming (stop)-signals just as the original adversary A. Moreover, the machine Sim usually only transmits values from the adversary to the system and vice versa; especially it does not produce any outputs alone by itself. Thus, the complete adversary A', i.e., A and Sim, will not produce outputs any further if the original adversary does not, yielding the desired liveness configuration. Our example of Section 5 belongs to that kind of system. All other examples which have been proved so far, e.g., secure channels [4], fair exchange protocols [9], and secure group key exchange [10], belong to that kind of system as well. Formally, liveness simulatability is introduced as follows.

Definition 4.1 *(Liveness Simulatability) Let two arbitrary systems Sys_1 and Sys_2 be given such that $Sys_1 \geq^f Sys_2$ holds for a valid mapping f. We then call Sys_1 "at least as secure as" Sys_2 "with respect to liveness" (written $Sys_1 \geq^{f,\mathsf{live}} Sys_2$) if the following holds: For a given $(t_{\mathsf{stop}}, J, Q_{\mathsf{X}}, Q_{\mathsf{H}})$-liveness configuration $conf_1^{\mathsf{live}} = (\hat{M}_1, S_1, \{\mathsf{H}\} \cup \{\mathsf{X}_{\mathsf{fair}}\}, \mathsf{A}_1)^{\mathsf{live}} \in \mathsf{Conf}^{\mathsf{live}}(Sys_1)$ there exists a $(t_{\mathsf{stop}}, J, Q_{\mathsf{X}}, Q_{\mathsf{H}})$-liveness configuration $conf_2^{\mathsf{live}} = (\hat{M}_2, S_2, \{\mathsf{H}\} \cup \{\mathsf{X}_{\mathsf{fair}}\}, \mathsf{A}_2)^{\mathsf{live}} \in \mathsf{Conf}^{\mathsf{live}}(Sys_2)$ yielding indistinguishable views for the honest user. As usual, we distinguish between perfect, statistical, and computational indistinguishability.* $\diamond$

We will now show that liveness properties automatically carry over in case of liveness simulatability. Before we turn our attention to the actual preservation theorem, we state the following well-known lemma which we will need during the theorem's proof.

Lemma 4.1 *The statistical distance $\Delta(\phi(\mathsf{var}_k), \phi(\mathsf{var}'_k))$ for any function ϕ of random variables is at most $\Delta(\mathsf{var}_k, \mathsf{var}'_k)$.* $\Box$

Theorem 4.1 (Preservation of Polynomial Liveness) *Let an arbitrary system Sys_2 and a polynomial liveness property φ_{Sys_2} be given such that $Sys_2 \models \varphi_{Sys_2}$ holds. Furthermore, let a system Sys_1 and a valid mapping f be given with $Sys_1 \geq^{f,\mathsf{live}} Sys_2$. Then $Sys_1 \models \varphi_{Sys_1}$ for all φ_{Sys_1} with $\varphi_{Sys_1}(\hat{M}_1, S_1) := \varphi_{Sys_2}(\hat{M}_2, S_2)$ for an arbitrary structure $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$.*

Furthermore, note that all parameters of a liveness configuration must remain unchanged under simulatability since they are part of the honest user.

This holds in the perfect case and in the statistical case. It holds in the computational case if additionally, membership in $Req(S)$ is decidable in polynomial time for all S. □

Proof. At first, we show that φ_{Sys_1} is a well-defined liveness property for Sys_1. Let an arbitrary structure $(\hat{M}_1, S_1) \in Sys_1$ be given. Simulatability implies that for every structure $(\hat{M}_1, S_1) \in Sys_1$ there exists $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$. φ_{Sys_2} is a well-defined liveness property for Sys_2, so $\varphi_{Sys_2}(\hat{M}_2, S_2) =: (Req, \{\mathsf{p_1}^{\lhd}!, \ldots, \mathsf{p}_n{}^{\lhd}!\}, t_s)$ is a liveness property for $(\hat{M}_2, S_2)$. By the definition of valid mappings, we have $S_1 = S_2$, so the integrity requirement Req is well-defined on $(\hat{M}_1, S_1)$. Moreover, we have $\{\mathsf{p_1}^{\lhd}!, \ldots, \mathsf{p}_n{}^{\lhd}!\} \subseteq S_1^c$ if and only if $\{\mathsf{p_1}^{\lhd}!, \ldots, \mathsf{p}_n{}^{\lhd}!\} \subseteq S_2^c$. Therefore, the liveness property $\varphi_{Sys_1}(\hat{M}_1, S_1)$ is defined on the structure $(\hat{M}_1, S_1)$, so φ_{Sys_1} is a well-defined liveness property of Sys_1. We now have to show that Sys_1 fulfills φ_{Sys_1}. The actual proof will be done by contradiction, i.e., we will show that if Sys_1 did not fulfill the liveness property, the two systems could be distinguished.

Assume that Sys_1 does not fulfill its liveness property. Thus, there exist polynomials J, t_{stop} with $t_{\mathsf{stop}} < t_s$ such that for every polynomials Q_{X}, $Q_{\mathsf{H}} \geq \mathsf{min}(Req, S_1)$ there exists a $(t_{\mathsf{stop}},\ J,\ Q_{\mathsf{X}},\ Q_{\mathsf{H}})$-liveness configuration $conf_1^{\mathsf{live}}$ so that the good event does not occur within $Q_{\mathsf{H}}(k)$ switching steps of the honest user. Because of $Sys_1 \geq^{f,\mathsf{live}} Sys_2$ there is a $(t_{\mathsf{stop}},\ J,\ Q_{\mathsf{X}},\ Q_{\mathsf{H}})$-liveness configuration $conf_2^{\mathsf{live}} = (\hat{M}_2, S_2, \{\mathsf{H}\} \cup \{\mathsf{X_{fair}}\}, \mathsf{A}_2)^{\mathsf{live}} \in \mathsf{Conf}^{\mathsf{live}}(Sys_2)$ for $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$ such that

$$view_{conf_1^{\mathsf{live}}}(\{\mathsf{H}\} \cup \{\mathsf{X_{fair}}\}) \approx view_{conf_2^{\mathsf{live}}}(\{\mathsf{H}\} \cup \{\mathsf{X_{fair}}\})$$

holds. For the sake of readability we abbreviate $\{\mathsf{H}\} \cup \{\mathsf{X_{fair}}\}$ by H' and set $S := S_1 := S_2$. Since H is a submachine of H', we can apply Lemma 4.1 which yields $view_{conf_1^{\mathsf{live}}}(\mathsf{H}) \approx view_{conf_2^{\mathsf{live}}}(\mathsf{H})$. Moreover, the view of H in both configurations contains the run at S^{H}, i.e., the run is a function of the view, so we finally obtain

$$run_{conf_1^{\mathsf{live}}}\lceil_{S^{\mathsf{H}}} \approx run_{conf_2^{\mathsf{live}}}\lceil_{S^{\mathsf{H}}}.$$

As usual we have to distinguish between the perfect, statistical and computational case. In the computational case, both configurations have to be polynomial-time.

In the perfect case we have $view_{conf_1^{\mathsf{live}}}(\mathsf{H}') = view_{conf_2^{\mathsf{live}}}(\mathsf{H}')$ because of $Sys_1 \geq^{f,\mathsf{live},\mathsf{perf}} Sys_2$, i.e., the distributions of the views are identical which yields $run_{conf_1^{\mathsf{live}}}\lceil_{S^{\mathsf{H}}} = run_{conf_2^{\mathsf{live}}}\lceil_{S^{\mathsf{H}}}$. Because of $(\hat{M}_2, S_2) \models^{\mathsf{perf}} (Req, \{\mathsf{p_1}^{\lhd}!, \ldots, \mathsf{p}_n{}^{\lhd}!\}, t_s)$, there exist two polynomials Q'_{X}, $Q'_{\mathsf{H}} \geq \mathsf{min}(Req, S)$ such that

$$[(run_{conf_2^{\mathsf{live}},k,Q'_{\mathsf{H}}(k)}\lceil_{S^{\mathsf{H}}})] \subseteq Req(S^{\mathsf{H}})$$

holds for every $(t_{\mathsf{stop}},\ J,\ Q'_{\mathsf{X}},\ Q'_{\mathsf{H}})$-liveness configuration $conf_2^{\mathsf{live}}$. Thus, for every given $(t_{\mathsf{stop}},\ J,\ Q'_{\mathsf{X}},\ Q'_{\mathsf{H}})$-liveness configuration $conf_1^{\mathsf{live}}$, we have an indistinguishable $(t_{\mathsf{stop}},\ J,\ Q'_{\mathsf{X}},\ Q'_{\mathsf{H}})$-liveness configuration $conf_2^{\mathsf{live}}$ of $(\hat{M}_2, S_2)$ with the above property. Assume now that $(\hat{M}_1, S_1)$ does not fulfill $(Req, \{\mathsf{p_1}^{\lhd}!, \ldots, \mathsf{p}_n{}^{\lhd}!\}, t_s)$. This immediately contradicts the assumption that $[(run_{conf_1^{\mathsf{live}},k,Q'_{\mathsf{H}}(k)}\lceil_{S^{\mathsf{H}}})] \not\subseteq Req(S^{\mathsf{H}})$ while $[(run_{conf_2^{\mathsf{live}},k,Q'_{\mathsf{H}}(k)}\lceil_{S^{\mathsf{H}}})] \subseteq Req(S^{\mathsf{H}})$, since $run_{conf_1^{\mathsf{live}}}\lceil_{S^{\mathsf{H}}} = run_{conf_2^{\mathsf{live}}}\lceil_{S^{\mathsf{H}}}$ holds.

In the statistical case, we have $view_{conf_1^{\mathsf{live}}}(\mathsf{H}') \approx_{SMALL} view_{conf_2^{\mathsf{live}}}(\mathsf{H}')$, which again yields $run_{conf_1^{\mathsf{live}}} \lceil_{S^\mathsf{H}} \approx_{SMALL} run_{conf_2^{\mathsf{live}}} \lceil_{S^\mathsf{H}}$. Thus, the statistical distance $\Delta(run_{conf_1^{\mathsf{live}},k,l(k)} \lceil_{S^\mathsf{H}},\ run_{conf_2^{\mathsf{live}},k,l(k)} \lceil_{S^\mathsf{H}})$ is a function $g(k) \in SMALL$ for all polynomials l. We apply Lemma 4.1 to the characteristic function $1_{v\lceil_{S^\mathsf{H}} \notin Req(S^\mathsf{H})}$ on such views v. This gives

$$\begin{aligned} & |P(run_{conf_1^{\mathsf{live}},k,l(k)} \lceil_{S^\mathsf{H}} \notin Req(S^\mathsf{H})) \\ - \quad & P(run_{conf_2^{\mathsf{live}},k,l(k)} \lceil_{S^\mathsf{H}} \notin Req(S^\mathsf{H}))| \\ \leq \quad & g(k). \end{aligned}$$

for every polynomial l. If we use the above inequality with $l := Q'_\mathsf{H}$ we obtain

$$\begin{aligned} & |P(run_{conf_1^{\mathsf{live}},k,Q'_\mathsf{H}(k)} \lceil_{S^\mathsf{H}} \notin Req(S^\mathsf{H})) \\ - \quad & P(run_{conf_2^{\mathsf{live}},k,Q'_\mathsf{H}(k)} \lceil_{S^\mathsf{H}} \notin Req(S^\mathsf{H}))| \\ \leq \quad & g(k). \end{aligned}$$

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

In the computational case, we define a distinguisher Dis as follows: Given a view of machine H, it extracts the $Q'_\mathsf{H}(k)$ prefix of the user's view restricted to S^H and verifies if the result lies in $Req(S^\mathsf{H})$. If yes, it outputs 0, otherwise 1. This distinguisher is polynomial-time (in the security parameter k) because the view of H is of polynomial length, and membership in $Req(S)$ (and therefore also in $Req(S^\mathsf{H})$) was required to be polynomial-time decidable. Its advantage in distinguishing is

$$\begin{aligned} & |P(\mathsf{Dis}(1^k, view_{conf_1^{\mathsf{live}},k}) = 1) \\ - \quad & P(\mathsf{Dis}(1^k, view_{conf_2^{\mathsf{live}},k}) = 1)| \\ = \quad & |P(run_{conf_1^{\mathsf{live}},k,Q'_\mathsf{H}(k)} \lceil_{S^\mathsf{H}} \notin Req(S^\mathsf{H})) \\ - \quad & P(run_{conf_2^{\mathsf{live}},k,Q'_\mathsf{H}(k)} \lceil_{S^\mathsf{H}} \notin Req(S^\mathsf{H}))|. \end{aligned}$$

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

5 An Example: Secure Message Transmission with Reliable Channels

In the following, we present a specification for secure message transmission with reliable channels. Here, reliability is considered as a liveness property which the system will be proved to fulfill. Moreover, we present a secure implementation.

We start with a brief review on standard cryptographic systems and composition (cf. [11] for more details). We model the real life by assigning to every user u a single machine M_u and assume this machine to be correct if and only if the user is honest. The machine M_u of user u has special ports $\mathsf{in}_u?$ and $\mathsf{out}_u!$ for connecting to the user u. A standard cryptographic system *Sys* can now

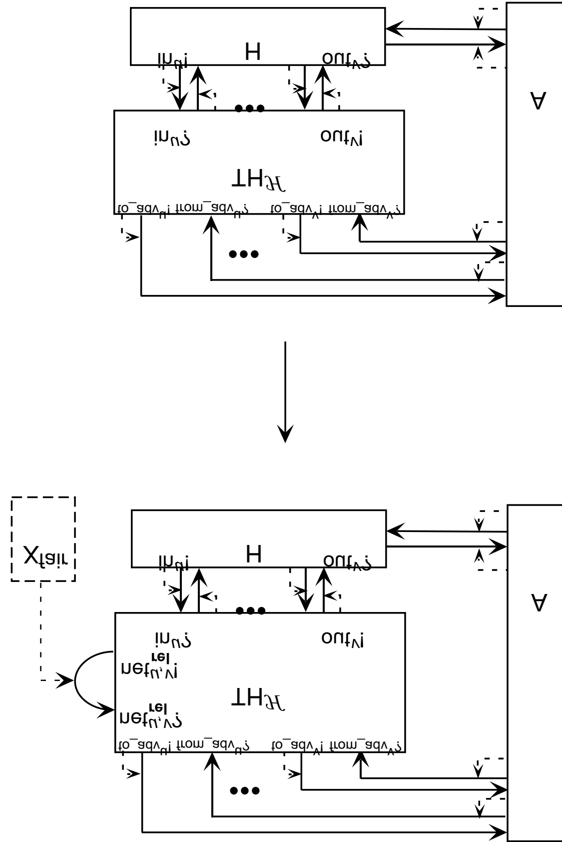

Figure 3: Transformation of ideal system Sys_{id} into modified system $Sys_{\mathsf{id}}^{\mathsf{rel}}$. (Exemplarily, we only consider one self-loop port $\mathsf{rel}_{u,v}$, and we sketched that this port is scheduled by the master scheduler $\mathsf{X}_{\mathsf{fair}}$.)

be derived by a *trust model.* The trust model consists of an access structure $\mathcal{ACC}$ and a channel model χ. If n denotes the number of all participants, then $\mathcal{ACC}$ is a set of subsets $\mathcal{H} \subseteq \{1, \ldots, n\}$ denoting the possible sets of correct machines. For each set $\mathcal{H}$ there will be exactly one structure consisting of the machines belonging to the set $\mathcal{H}$. The channel model classifies every connection as either secure (private and authentic), authenticated, or insecure and derives the correspondent network connectivity. These changes can easily be done via port renaming and duplication (cf. [11]).

For a fixed set $\mathcal{H}$ and a fixed channel model we obtain modified machines for every machine M_u which we refer to as $\mathsf{M}_{u,\mathcal{H}}$. We denote the set of them by $\hat{M}_{\mathcal{H}}$ (i.e., $\hat{M}_{\mathcal{H}} := \{\mathsf{M}_{u,\mathcal{H}} \mid u \in \mathcal{H}\}$), so real systems are given by $Sys_{\mathsf{real}} = \{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}}) \mid \mathcal{H} \in \mathcal{ACC}\}$. Ideal systems typically are of the form $Sys_{\mathsf{id}} = \{(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}}) \mid \mathcal{H} \in \mathcal{ACC}\}$ with the same sets $S_{\mathcal{H}}$ as in the corresponding real system Sys_{real}. The machine $\mathsf{TH}_{\mathcal{H}}$ is called *trusted host.*

After this brief review we can turn our attention to the actual system. Both the ideal and real system are based on the systems for secure message transmission introduced in [11] that we will modify to fit our requirements.

5.1 The Ideal System

We start with a brief description of the ideal system for secure message transmission. The system is of the typical form $Sys_{\mathsf{id}} = \{(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}}) | \mathcal{H} \in \mathcal{ACC}\}$, $\mathcal{ACC}$ is the powerset of $\{1, \ldots, n\}$. The system is illustrated in the upper part of Figure 3. The ideal machine $\mathsf{TH}_{\mathcal{H}}$ models initialization, sending and receiving of messages. A user u can initialize communications with other users by inputting a command of the form $(\mathsf{snd_init})$ to the port $\mathsf{in}_u?$ of $\mathsf{TH}_{\mathcal{H}}$. In real systems, initialization corresponds to key generation and authenticated key exchange. Sending of messages to a user v is triggered by a command (send, m, v). If v is honest, the message is stored in an internal array of $\mathsf{TH}_{\mathcal{H}}$ and a command $(\mathsf{send_blindly}, i, l, v)$ is output to the adversary, l and i denote the length of the message m and its position in the array, respectively. This models that the adversary will see that a message has been sent and he might also be able to know the length of that message. The authors speak of tolerable imperfections that are explicitly granted to the adversary. Because of the underlying asynchronous timing model, $\mathsf{TH}_{\mathcal{H}}$ has to wait for a special term $(\mathsf{receive_blindly}, u, i)$ or $(\mathsf{rec_init}, u)$ sent by the adversary signaling that the i-th stored message sent by u to v should be delivered or that a channel between u and v should be initialized, respectively. The user v will receive inputs of the form $(\mathsf{receive}, u, m)$ and $(\mathsf{rec_init}, u)$, respectively. If v is dishonest, $\mathsf{TH}_{\mathcal{H}}$ will simply output (send, m, v) to the adversary. Finally, the adversary can send a message m to a user v by sending a command $(\mathsf{receive}, u, m)$ to the port $\mathsf{from_adv}_v?$ of $\mathsf{TH}_{\mathcal{H}}$ for a corrupted user v, and he can also stop the machine of any user by sending a command (stop)[2] to a corresponding port of $\mathsf{TH}_{\mathcal{H}}$ which corresponds to exceeding the machine's runtime bounds in the real world.

Necessary Modification of the System. Roughly speaking, we model reliable channels by providing the trusted host with additional self-loop channels $\mathsf{rel}_{u,v}$, modeling the "reliable net" in the real world. This is illustrated in the lower part of Figure 3.

More precisely, we assume a set $\mathcal{I}_{\mathcal{H}} \subseteq \mathcal{H}^2$ of pairs of users where $(u, v) \in \mathcal{I}_{\mathcal{H}}$ states that there is a reliable channel for sending messages from user u to user v. Note, that we do not restrict ourselves to a symmetric relation $\mathcal{I}_{\mathcal{H}}$.

Coming back to our specification, we model reliable channels for every pair $(u, v) \in \mathcal{I}_{\mathcal{H}}$ by providing the trusted host with additional self-loop channels $\{\mathsf{rel}_{u,v}!, \mathsf{rel}_{u,v}?\} \subseteq \mathsf{ports}(\mathsf{TH}_{\mathcal{H}})$. The corresponding clock port $\mathsf{rel}_{u,v}{}^{\triangleleft}!$ remains free at first; it will later be connected to the fair master scheduler to achieve the desired liveness property. If now user u sends a message to user v, $\mathsf{TH}_{\mathcal{H}}$ outputs this message at $\mathsf{rel}_{u,v}!$. This might also be an initialization command, since its corresponding part in the real world is key exchange which obviously requires sending the keys to user v. Moreover, the (blinded) message is additionally output to the adversary as usual. By now, the blinded message resides in the self-loop channel buffer and waits to be scheduled. If it is eventually scheduled by the master scheduler, the trusted host outputs the message to its recipient. Intuitively, we can expect some kind of liveness property, since all of these messages contained in the self-loops will eventually be delivered, at least if the master scheduler is fair and runs sufficiently long.

[2]Note that this stop is completely unrelated to the stop which a H contained in a liveness configuration sends to A after $t_{\mathsf{stop}}(k)$ scheduling steps!

After presenting the main ideas, we can now turn our attention to the actual modifications of the system. Let $n \in \mathbb{N}$ and $\mathcal{M} := \{1, \ldots, n\}$ denote the number of participants and the set of indices, respectively. Throughout the following, let an arbitrary $\mathcal{H} \in \mathcal{ACC}$ together with the set $\mathcal{I}_\mathcal{H}$ be given. As we already stated above, the standard trusted host $\mathsf{TH}_\mathcal{H}$ mainly has to be modified in the "send message" and "send initialization" transitions. Moreover, we have to define what $\mathsf{TH}_\mathcal{H}$ does if it receives an input at one of the special new ports $\mathsf{rel}_{u,v}?$ yielding the following modifications:

- If $\mathsf{TH}_\mathcal{H}$ receives an input ($\mathsf{snd_init}$) at port $\mathsf{in}_u?$, it additionally outputs ($\mathsf{snd_init}$) at $\mathsf{rel}_{u,v}!$ for all v with $(u, v) \in \mathcal{I}_\mathcal{H}$. However, it still schedules the output intended for the adversary.
- If $\mathsf{TH}_\mathcal{H}$ receives an input (send, m, v) at port $\mathsf{in}_u?$, it checks whether $(u, v) \in \mathcal{I}_\mathcal{H}$. In this case it additionally outputs $(\mathsf{send_blindly}, i, l, v)$ at $\mathsf{rel}_{u,v}!$, but it still schedules the output intended for the adversary.
- If $\mathsf{TH}_\mathcal{H}$ receives an input ($\mathsf{snd_init}$) at $\mathsf{rel}_{u,v}?$ it does the same checks as in the "receive-initialization" stage (the adversary tries to schedule the keys), i.e., it checks that the machine of user v has not been stopped so far, that the connection has already been initialized and so on. If all these checks succeed, it outputs $(\mathsf{rec_init}, u)$ at $\mathsf{out}_v!$.
- If $\mathsf{TH}_\mathcal{H}$ receives an input $(\mathsf{send_blindly}, i, l, v)$ at $\mathsf{rel}_{u,v}?$, it acts similarly as in the above case, i.e., it performs the test of the "receive-message" stage, and if all tests succeed it outputs $(\mathsf{receive}, u, m)$ at $\mathsf{out}_v!$.

After this rather informal definition which we hope to increase basic understanding, we now rigorously define our system. After that, we briefly sketch the concrete implementation for secure message transmission along with the necessary modification to preserve the "at least as secure as" relation.

Scheme 5.1 (Reliable Secure Message Transmission) Let $n \in \mathbb{N}$, a finite index set Σ, and a polynomial $L \in \mathbb{N}[x]$ be given. $L(k)$ bounds the length of the messages for the security parameter k. Let $\mathcal{M} := \{1, \ldots, n\}$ denote the set of possible participants, and let the access structure $\mathcal{ACC}$ be the powerset of $\mathcal{M}$. Moreover, let a family of sets $(\mathcal{I}_\mathcal{H})_{\mathcal{H} \in \mathcal{ACC}}$ be given such that $\mathcal{I}_\mathcal{H} \subseteq \mathcal{H} \times \mathcal{H}$. Our specification for secure message transmission with reliable channels is now a standard ideal system

$$Sys_{\mathsf{id}}^{\mathsf{rel}} = \{(\{\mathsf{TH}_\mathcal{H}\}, S_\mathcal{H}) \mid \mathcal{H} \in \mathcal{ACC}\}.$$

As in the standard definition, we have $S_\mathcal{H}^c \supseteq \{\mathsf{in}_u!, \mathsf{out}_u?, \mathsf{in}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\}$, but additionally, there are specified ports $\mathsf{rel}_{u,v}{}^{\triangleleft}!$ for every pair $(u, v) \in \mathcal{I}_\mathcal{H}$. Thus, we obtain

$$\begin{aligned} S_\mathcal{H}^c &= \{\mathsf{in}_u!, \mathsf{out}_u?, \mathsf{in}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\} \\ &\cup \{\mathsf{rel}_{u,v}{}^{\triangleleft}! \mid (u, v) \in \mathcal{I}_\mathcal{H}\}. \end{aligned}$$

The machine $\mathsf{TH}_\mathcal{H}$ is defined as follows. When $\mathcal{H}$ is clear from the context, let $\mathcal{A} := \mathcal{M} \setminus \mathcal{H}$ denote the indices of corrupted machines. The ports

of $\mathsf{TH}_\mathcal{H}$ are $\{\mathsf{in}_u?, \mathsf{out}_u!, \mathsf{out}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\} \cup \{\mathsf{rel}_{u,v}!, \mathsf{rel}_{u,v}? \mid (u,v) \in \mathcal{I}_\mathcal{H}\} \cup \{\mathsf{from_adv}_u?, \mathsf{to_adv}_u!, \mathsf{to_adv}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\}$.

Internally, $\mathsf{TH}_\mathcal{H}$ maintains arrays $(\mathit{init}^*_{u,v})_{u,v\in\mathcal{M}}$ and $(\mathit{stopped}^*_u)_{u\in\mathcal{M}}$ over $\{0,1\}$, both initialized with 0 everywhere, and an array $(\mathit{deliver}^*_{u,v})_{u,v\in\mathcal{M}}$ of lists, all initially empty. The state-transition function of $\mathsf{TH}_\mathcal{H}$ is defined by the following rules, written in a pseudo-code language. For the sake of readability, we exemplarily annotate the first part of the definition, the "send initialization" transition, i.e., key generation in the real world.

Initialization.

- **Send initialization:** Assume, that the user u wants to generate its encryption and signature keys and distribute the corresponding public keys over authenticated channels. He can do so by sending a command $(\mathsf{snd_init})$ to $\mathsf{TH}_\mathcal{H}$. Now, the system checks that the user's machine itself has not reached its runtime bound (i.e., it has not been stopped), and that no key generation of this user has already occurred in the past. These two checks correspond to $\mathit{stopped}^*_u = 0$, and $\mathit{init}^*_{u,u} = 0$, respectively. If both checks hold, the keys are distributed over authenticated channels, modeled by an output $(\mathsf{snd_init})$ to the adversary. After receiving this command, the adversary can decide whether it schedules the keys immediately, later on, or even leave them on the channels forever. Moreover, the keys are additionally sent over all reliable channels $\mathsf{rel}_{u,v}$ for $(u,v) \in \mathcal{I}_\mathcal{H}$. In our pseudo-code language this is expressed as follows:

  On input $(\mathsf{snd_init})$ at $\mathsf{in}_u?$: If $\mathit{stopped}^*_u = 0$ and $\mathit{init}^*_{u,u} = 0$, set $\mathit{init}^*_{u,u} := 1$. After that, output $(\mathsf{snd_init})$ at $\mathsf{rel}_{u,v}!$ for every $(u,v) \in \mathcal{I}_\mathcal{H}$, $(\mathsf{snd_init})$ at $\mathsf{to_adv}_u!$, and 1 at $\mathsf{to_adv}_u{}^{\triangleleft}!$.

- **Receive initialization.** On input $(\mathsf{rec_init}, u)$ at $\mathsf{from_adv}_v?$ with $u \in \mathcal{M}, v \in \mathcal{H}$ or $(\mathsf{snd_init})$ at $\mathsf{rel}_{u,v}?$: If $\mathit{stopped}^*_v = 0$ and $\mathit{init}^*_{u,v} = 0$ and $[u \in \mathcal{H} \Rightarrow \mathit{init}^*_{u,u} = 1]$, set $\mathit{init}^*_{u,v} := 1$ and output $(\mathsf{rec_init}, u)$ at $\mathsf{out}_v!$.

Sending and receiving messages.

- **Send.** On input (send, m, v) at $\mathsf{in}_u?$ with $m \in \Sigma^+$, $l := \mathsf{len}(m) \leq L(k)$, and $v \in \mathcal{M} \setminus \{u\}$: If $\mathit{stopped}^*_u = 0$, $\mathit{init}^*_{u,u} = 1$, and $\mathit{init}^*_{v,u} = 1$: If $v \in \mathcal{A}$ then { output (send, m, v) at $\mathsf{to_adv}_u!$ } else { $i := \mathsf{size}(\mathit{deliver}^*_{u,v}) + 1$; $\mathit{deliver}^*_{u,v}[i] := m$. If $(u,v) \in \mathcal{I}_\mathcal{H}$ it outputs $(\mathsf{send_blindly}, i, l, v)$ at $\mathsf{rel}_{u,v}!$, $(\mathsf{send_blindly}, i, l, v)$ at $\mathsf{to_adv}_u!$ and 1 at $\mathsf{to_adv}_u{}^{\triangleleft}!$. Otherwise it outputs $(\mathsf{send_blindly}, i, l, v)$ at $\mathsf{to_adv}_u!$, 1 at $\mathsf{to_adv}_u{}^{\triangleleft}!$. }.

- **Receive from honest party u.** On input $(\mathsf{receive_blindly}, u, i)$ at $\mathsf{from_adv}_v?$ with $u, v \in \mathcal{H}$, $i \in \mathbb{N}$ or $(\mathsf{send_blindly}, i, l, v)$ at $\mathsf{rel}_{u,v}?$: If $\mathit{stopped}^*_v = 0$, $\mathit{init}^*_{v,v} = 1$, $\mathit{init}^*_{u,v} = 1$, and $m := \mathit{deliver}^*_{u,v}[i] \neq \downarrow$, then output $(\mathsf{receive}, u, m)$ at $\mathsf{out}_v!$.

- **Receive from dishonest party u.** On input $(\mathsf{receive}, u, m)$ at $\mathsf{from_adv}_v?$ with $u \in \mathcal{A}$, $m \in \Sigma^+$, $\mathsf{len}(m) \leq L(k)$, and $v \in \mathcal{H}$: If $\mathit{stopped}^*_v = 0$, $\mathit{init}^*_{v,v} = 1$ and $\mathit{init}^*_{u,v} = 1$, output $(\mathsf{receive}, u, m)$ at $\mathsf{out}_v!$.

- **Stop.** On input (stop) at $\mathsf{from_adv}_u?$ with $u \in \mathcal{H}$, set $stopped_u^* = 1$ and output (stop) at $\mathsf{out}_u!$.

$\diamond$

After presenting the abstract specification, we now briefly sketch the concrete implementation for secure message transmission, and the necessary modifications to preserve the "at least as secure as" relation.

5.2 The Real System

To understand the following reasoning, it is sufficient to give a brief review of Sys_{real}. The system is a standard cryptographic system of the form $Sys_{\mathsf{real}} = \{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}}) \mid \mathcal{H} \in \mathcal{ACC}\}$. $\mathcal{ACC}$ is the powerset of $\mathcal{M}$, i.e., any subset of participants may be dishonest. It uses asymmetric encryption and digital signatures as cryptographic primitives. A user u can let his machine create signature and encryption keys that are sent to other users over authenticated channels $\mathsf{aut}_{u,v}$. Furthermore, messages sent from user u to user v will be signed and encrypted by M_u and sent to M_v over an insecure channel $\mathsf{net}_{u,v}$, representing the net in the real world. The adversary is able to schedule the communication between the users, and he can furthermore send arbitrary messages m to arbitrary users u for a dishonest sender v.

We now have to implement the modification of the ideal system in our concrete implementation. This can simply be achieved by providing additional channels between every pair $(\mathsf{M}_u, \mathsf{M}_v)$ of machines with $(u, v) \in \mathcal{I}_{\mathcal{H}}$, and let the master scheduler schedule them. Now the machine M_u behaves just as $\mathsf{TH}_{\mathcal{H}}$ does, i.e., it additionally sends messages over these reliable channels. Obviously, these channels precisely correspond to the channels $\mathsf{rel}_{u,v}$ of the ideal specification, so both system still provide the same functional behavior for their environment, i.e., for the honest user and the adversary. It would even be closer to the real world if we distinguished between reliable authentic channels for initialization commands and reliable but non-authentic channels for usual message transmission. We already succeeded in this task, cf. [2], but we omit it here since quite a lot preparatory work (e.g, a new channel type representing reliable non-authentic channels) is additionally needed for that. We will denote the modified real system by $Sys_{\mathsf{real}}^{\mathsf{rel}}$.

Moreover, if we take a look at the security proof of [11] we can see that the concrete implementation is derived using blackbox simulatability of the right form (see Section 4), so our preservation theorem can be applied. Looking at the proof, it is immediately obvious that this still holds for our modified systems.

5.3 Proof of Liveness

After introducing the specification, we now want to show that it in fact fulfills the desired liveness property, i.e., that messages sent by the honest user over reliable channels will eventually be received unless some internal checks of $\mathsf{TH}_{\mathcal{H}}$ fail (e.g., the user has not initialized itself, its machine has been stopped, etc.).

At first, we have to rigorously define the liveness property $\varphi_{Sys_{\mathsf{id}}^{\mathsf{rel}}}$. Let an arbitrary structure $(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}}) \in Sys_{\mathsf{id}}^{\mathsf{rel}}$ be given. Then the three components of $\varphi_{Sys_{\mathsf{id}}^{\mathsf{rel}}}(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}})$ are defined as follows.

If there exists $l_1, l_2 \in \mathbb{N}$, such that

$$\{\mathsf{in}_u!^c\colon (\mathsf{snd_init}), \mathsf{in}_u{}^{\triangleleft}!^c\colon 1\} \subseteq r_{l_1} \qquad \#\ \textit{Key generation of } u \textit{ in } r_{l_1}$$
$$\{\mathsf{in}_v!^c\colon (\mathsf{snd_init}), \mathsf{in}_v{}^{\triangleleft}!^c\colon 1\} \subseteq r_{l_2} \qquad \#\ \textit{Key generation of } v \textit{ in } r_{l_2}$$

and $l_3 > l_1$, $l_4 > l_2$ such that

$$(\mathsf{out}_v?^c\colon (\mathsf{rec_init}, u)) \in r_{l_3} \qquad \#\ \textit{Connection established from } u \textit{ to } v \textit{ in } r_{l_3}$$
$$(\mathsf{out}_u?^c\colon (\mathsf{rec_init}, v)) \in r_{l_4} \qquad \#\ \textit{Connection established from } v \textit{ to } u \textit{ in } r_{l_4}$$

then the following must hold. At first, set $l := \max\{l_3, l_4\}$. Then for every $t \in \mathbb{N}$,

$$\begin{array}{ll}
\{\mathsf{in}_u!^c\colon (\mathsf{send}, m, v), \mathsf{in}_u{}^{\triangleleft}!^c\colon 1\} \subseteq r_t & \#\ \textit{If } u \textit{ sends and schedules a message } m \textit{ to } v \\
\wedge\, t > l & \#\ \textit{after the connection has been established} \\
\wedge\, m \in \Sigma^+ \wedge \mathit{len}(m) < L(k) & \#\ \textit{and the message is valid} \\
\wedge\, \forall t_1 < t\colon (\mathsf{out}_u?^c\colon (\mathsf{stop})) \notin r_{t_1} & \#\ \textit{and the sender's machine is not stopped} \\
\Longrightarrow & \#\ \textit{then} \\
\big(\exists t_2 \in \mathbb{N}\colon (\mathsf{out}_v?^c\colon (\mathsf{stop})) \in r_{t_2} & \#\ \textit{the receiver's machine is either stopped} \\
\vee\, \exists t_3 > t\colon & \#\ \textit{or there is a future time } t_3 \textit{ such that} \\
\quad (\mathsf{out}_v?^c\colon (\mathsf{receive}, u, m)) \in r_{t_3}\big) & \#\ \textit{the message } m \textit{ will be delivered in } r_{t_3}
\end{array}$$

Figure 4: The formula for $\mathit{Req}(S_{\mathcal{H}})$.

- Its first component, the integrity requirement Req, is defined as follows. Consider two users u, v, $u \neq v$ such that $(u, v) \in \mathcal{I}_{\mathcal{H}}$. Informally, a run is contained in the requirement if the following holds: if both users u and v have established a connection (i.e., they have initialized themselves, and both users have received the corresponding keys), and the user u has not been stopped so far, then a valid message sent from u to v will either be eventually delivered, or the recipient v has been or will be stopped.

  Formally, this is captured as follows. As usual, the lth step of the run r is denoted by r_l. For the considered set $S_{\mathcal{H}}$ of specified ports, we define that a run r of an arbitrary configuration is contained in $\mathit{Req}(S_{\mathcal{H}})$ if the formula of Figure 4 holds.

- Secondly, we have to specify the set of ports that should be scheduled by the fair master scheduler. For a given set $S_{\mathcal{H}}$, we define this set to be $\{\mathsf{rel}_{u,v}{}^{\triangleleft}! \mid (u, v) \in \mathcal{I}_{\mathcal{H}}\}$.

- At last, the function t_s can be chosen arbitrarily as long as it is bounded by a polynomial, i.e., $t_s(k) \in O(k^c)$ for a natural number c.

We can now state our main theorem.

Theorem 5.1 *The system $\mathit{Sys}^{\mathsf{rel}}_{\mathsf{id}}$ fulfills the polynomial liveness property $\varphi_{\mathit{Sys}^{\mathsf{rel}}_{\mathsf{id}}}$ perfectly, i.e., in formulas $\mathit{Sys}^{\mathsf{rel}}_{\mathsf{id}} \models^{\mathsf{perf}} \varphi_{\mathit{Sys}^{\mathsf{rel}}_{\mathsf{id}}}$.* □

Proof. Let an arbitrary structure $(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}}) \in Sys^{\mathsf{rel}}_{\mathsf{id}}$, arbitrary polynomials J, t_{stop} with $t_{\mathsf{stop}} < t_s$ be given. Note that $\mathsf{min}(Req, S_{\mathcal{H}})$ is constant since there exists a sequence of length five which is not contained in $Req(S_{\mathcal{H}})$. (Four steps are needed for key generation and distribution, the last step contains a send-command for a valid message.) We then define $Q_{\mathsf{X}}(k) := Q_{\mathsf{H}}(k) := t_{\mathsf{stop}}(k) + J(k) \cdot k^x + \mathsf{min}(Req, S_{\mathcal{H}})$. Now, let $conf^{\mathsf{live}} = (\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}}, \{\mathsf{H}\} \cup \{\mathsf{X}_{\mathsf{fair}}\}, \mathsf{A})^{\mathsf{live}} \in \mathsf{Conf}^{\mathsf{live}}(Sys^{\mathsf{rel}}_{\mathsf{id}})$ denote an arbitrary $(t_{\mathsf{stop}}, J, Q_{\mathsf{X}}, Q_{\mathsf{H}})$-liveness configuration. We have to show that all $Q_{\mathsf{H}}(k)$ prefixes of the restriction of the run to the ports $S^{\mathsf{H}}_{\mathcal{H}}$ lie in $Req(S^{\mathsf{H}}_{\mathcal{H}})$.

In our particular case, this means the following: let an arbitrary pair $(u, v) \in \mathcal{I}_{\mathcal{H}}$ be given, and assume that the preconditions of $Req(S_{\mathcal{H}})$ are fulfilled, i.e., both users u and v have initialized themselves and a connection has been established between them at time l. Now, user u sends a command (send, m, v) at $\mathsf{TH}_{\mathcal{H}}$ at time $t > l$ and schedules it. Moreover, the message is valid and the machine of user u has not been stopped so far.

If we take a look at the "send" transition of the machine $\mathsf{TH}_{\mathcal{H}}$, we can see that all of its internal checks will succeed by our above preconditions. Thus, it will output $(\mathsf{send_blindly}, i, l, v)$ at $\mathsf{rel}_{u,v}!$. By construction of $\mathsf{TH}_{\mathcal{H}}$ it only outputs anything at $\mathsf{rel}_{u,v}!$ if it obtains inputs of the form $(\mathsf{snd_init})$ or $(\mathsf{send}, \cdot, v)$ at $\mathsf{in}_u?$. Since these inputs must come from the user u and the overall honest user H will stop outputting messages after its $t_{\mathsf{stop}}(k)$-th switching step, there can be at most $t_{\mathsf{stop}}(k)$ messages stored in the buffer $\widetilde{\mathsf{rel}_{u,v}}$. By precondition, the function t_s (as a function of k) is bounded by k^x for a natural number x, so $t_{\mathsf{stop}}(k) < t_s(k)$ implies that the number of messages stored in $\widetilde{\mathsf{rel}_{u,v}}$ is also bounded by k^x.

We now distinguish two cases. First, let us assume that the user v is stopped before the overall user H stops outputting messages, i.e., before the run-empty phase begins. In this case, $\mathsf{TH}_{\mathcal{H}}$ will output a command (stop) at $\mathsf{out}_v!$ and schedule it by construction, so we have $(\mathsf{out}_v?^c : (\mathsf{stop})) \in r_{t_2}$ for one particular $t_2 \in \mathbb{N}$. Hence, the requirement is fulfilled in this case, even before the run-empty phase begins. Second, let us assume that the machine of v has not been stopped before the run-empty phase begins. Since the adversary does not produce outputs any further, and $\mathsf{TH}_{\mathcal{H}}$ may only stop a machine after it has been scheduled by the adversary, the machine of v will not be stopped during the whole run. In this case, we have to prove that the message m will in fact be delivered.

As we already stated above, the desired message m (more precisely, the term $(\mathsf{send_blindly}, i, l, v)$, which corresponds to m) has been stored in the buffer $\widetilde{\mathsf{rel}_{u,v}}$. Moreover, this buffer can contain at most a polynomial number of messages (bounded by k^x). By precondition, the master scheduler $\mathsf{X}_{\mathsf{fair}}$ is J-fair, hence it schedules the first message of every of its connected buffers again and again, each one always after at most $J(k)$ steps. Since $\mathsf{rel}_{u,v}{}^{\triangleleft}! \in \mathsf{ports}(\mathsf{X}_{\mathsf{fair}})$ holds by assumption, the buffer $\widetilde{\mathsf{rel}_{u,v}}$ has to be scheduled after at most $J(k)$ switching steps of $\mathsf{X}_{\mathsf{fair}}$, so the term $(\mathsf{send_blindly}, i, l, v)$ will be scheduled after at most $J(k) \cdot k^x$ switching steps of $\mathsf{X}_{\mathsf{fair}}$. If we now take a closer look at the behavior of $\mathsf{TH}_{\mathcal{H}}$ in this case, we will see that all internal checks will succeed by assumption (the user v is initialized, a connection is established, its machine has not been stopped, and the message m has been stored in $deliver^*_{u,v}[i]$ before). Thus, $\mathsf{TH}_{\mathcal{H}}$ outputs $(\mathsf{receive}, u, m)$ at $\mathsf{out}_v!$, so we have $(\mathsf{out}_v?^c : (\mathsf{receive}, u, m)) \in r_{t_3}$ for one

particular time $t_3 > t$. Note that our choice of Q_{X} and Q_{H} additionally ensures that both $\mathsf{X}_{\mathsf{fair}}$ and H run sufficiently long for this event to happen.

Now, we are almost finished. The only thing left to show is that this input occurs after a polynomial number of switching steps of the *user*; by now we only showed that it will happen after a polynomial number of switching steps of the *master scheduler*. However, since the user H does not produce any outputs any further, the master scheduler will always be scheduled immediately after the honest user. Thus, the number of switching steps the honest user can perform in the run-empty phase is bounded by the number of switching steps of $\mathsf{X}_{\mathsf{fair}}$. Therefore, the message will be received after at most $J(k) \cdot k^x$ switching steps of the honest users, counted from the beginning of the run-empty phase, i.e., after at most $t_{\mathsf{stop}}(k) + J(k) \cdot k^x \leq Q_{\mathsf{H}}$ switching steps in total, which finishes the proof. ∎

After proving the liveness property for the ideal specification, we now concentrate on the concrete implementation.

Theorem 5.2 *The real system $Sys^{\mathsf{rel}}_{\mathsf{real}}$ fulfills the polynomial liveness property $\varphi_{Sys^{\mathsf{rel}}_{\mathsf{real}}}$ computationally, with $\varphi_{Sys^{\mathsf{rel}}_{\mathsf{real}}}$ given as in Theorem 4.1. In formulas, $Sys^{\mathsf{rel}}_{\mathsf{real}} \models^{\mathsf{poly}} \varphi_{Sys^{\mathsf{rel}}_{\mathsf{real}}}$.* □

Proof. Naturally, perfect fulfillment of polynomial liveness implies fulfillment in the computational case. Thus, using Theorem 5.1, we know that $Sys^{\mathsf{id}}_{\mathsf{rel}} \models^{\mathsf{poly}} \varphi_{Sys^{\mathsf{id}}_{\mathsf{rel}}}$. As we already stated above, the concrete implementation is at least as secure as the abstract specification with respect to liveness in the computational case, i.e., $Sys^{\mathsf{rel}}_{\mathsf{real}} \geq^{f,\mathsf{live},\mathsf{poly}} Sys^{\mathsf{rel}}_{\mathsf{id}}$. Now the claim follows with Theorem 4.1. ∎

6 Summary

We have presented the first general definition of polynomial fairness and polynomial liveness in asynchronous reactive systems. We considered three grades of fulfilling a given polynomial liveness property: perfect (denoting usual fulfillment), statistical (denoting fulfillment up to a statistically small error probability) and computational (denoting fulfillment up to a negligible error probability, if all machines have polynomial runtime). Especially the computational case is essential to cope with real cryptography, since usually we can only ensure that good things happen if the underlying cryptographic primitives have not been broken, which might happen with negligible probability. Our approach might help to make the important concept of liveness better accessible for systems involving real cryptographic primitives. We have shown that polynomial liveness properties behave well under simulatability under certain conditions which enables step-wise refinement and modular proofs. Moreover, properties of abstract specifications can be validated by formal proof tools more easily than concrete implementations, although the polynomial-time limits t_s and J might make that more complicated for polynomial liveness than it is for safety properties. As an example fitting our definition, we have presented a specification of secure message transmission with reliable channels. Here, reliability is considered as the desired liveness property, and we have shown that the abstract specification in

fact fulfills this property. Moreover, we have presented a concrete implementation, and, using our preservation theorem of the previous section, we have concluded that the implementation also fulfills this liveness property.

The concrete protocol considered in this paper was rather simple. On the long run, we might aim at proving polynomial liveness for more complex protocols, mainly focusing on those protocols which are in fact used in practice. Furthermore, so far our definitions are based on *strict* polynomial-time algorithms as used in the underlying system model [11]. As a number of problems, e.g., the reliable asynchronous broadcast considered in [6], admit only *expected* polynomial-time solutions, it is an interesting avenue of future research to extend the underlying system model to expected polynomial-time algorithms and verify whether our liveness definition applies also there.

Acknowledgments.

We thank *Christian Jacobi* for helpful discussions.

References

[1] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[2] M. Backes. *Cryptographically Sound Analysis of Security Protocols.* Dissertation, Naturwissenschaftlich-Technische Fakultät der Universität des Saarlandes, Saarbrücken, 2002.

[3] M. Backes and C. Jacobi. Cryptographically sound and machine-assisted verification of security protocols. In *20th Symposium on Theoretical Aspects of Computer Science (STACS'03)*, Lecture Notes in Computer Science. Springer-Verlag, Berlin Germany, 2003.

[4] M. Backes, C. Jacobi, and B. Pfitzmann. Deriving cryptographically sound implementations using composition and formally verified bisimulation. In L.-H. Eriksson and P. A. Lindsay, editors, *Proc. Formal Methods Europe '02 (FME)*, volume 2391 of *Lecture Notes in Computer Science*, pages 310–329. Springer-Verlag, Berlin Germany, 2002.

[5] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In *Proc. Computer Aided Verification'94 (CAV)*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80. Springer-Verlag, Berlin Germany, 1994.

[6] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology: CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer-Verlag, Berlin Germany, 2001.

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

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

[9] B. Pfitzmann, M. Schunter, and M. Waidner. Provably secure certified mail. Research Report RZ 3207 (#93253), IBM Research, 2000.

[10] B. Pfitzmann, M. Steiner, and M. Waidner. A formal model for multi-party group key agreement. Technical Report RZ 3383 (# 93419), IBM Research, 2002.

[11] B. Pfitzmann and M. Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *Proc. 22th IEEE Symposium on Security & Privacy*, pages 184–202, 2001.

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