

Polynomial Fairness and Liveness

Michael Backes¹, Birgit Pfitzmann², Michael Steiner³, and Michael Waidner⁴
^{1,3}Saarland University, Saarbrücken, Germany
^{2,4}IBM Zurich Research Laboratory, Rüschlikon, Switzerland
{¹mbackes, ³steiner}@cs.uni-sb.de {²bpf, ⁴wmi}@zurich.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 obviously depend on the scheduler, which is usually considered to be fair in this case. Unfortunately, the standard definitions of fairness and liveness based on infinite sequences cannot be applied for most cryptographic protocols since one must restrict the adversary and the runs as a whole to polynomial length. We present the first general definition of polynomial fairness and liveness in asynchronous scenarios which is suited to cope with arbitrary cryptographic protocols. Furthermore, our definitions provide a link to the common approach of simulatability which is used throughout modern cryptography, and 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

If we consider properties of arbitrary protocols, we can distinguish between liveness and safety properties (according to the characterization of Alpern and Schneider [1]), besides confidentiality properties. Informally, a liveness property states that something good eventually happens. One usual 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., [6]) is to concentrate on so-called fair schedulers. Roughly, those guarantee that every message is delivered at some point of time in every infinite run of the system.

For most cryptographic protocols, these definitions cannot be used because one must restrict the adversary and the runs as a whole to polynomial length.

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. 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. 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 only aware of one paper that handles polynomial liveness properties for security protocols: Cachin et al. presented a nice approach for a specific protocol, asynchronous Byzantine agreement in [4]. It is shown 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 was limited to this specific protocol, and applying the approach more generally presumes a fixed number of subprotocols and the use of session ID’s (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, so the scheduler can 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 then happens in a polynomially bounded number of steps.

To the best of our knowledge, the approach of letting

In Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW), Cape Breton, Nova Scotia, Canada, pages 160 - 174, June 2002. IEEE Computer Society.

© 2001 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

the system run empty has not been used before to prove liveness for security protocols. However, in the verification of microprocessors we meet similar techniques. Roughly speaking, Burch and Dill showed in [3] 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 this paper, we only consider one run-empty phase. In principle, the user and adversary could restart outputting messages after the good event has happened, so that liveness would be extended to multiple good events. However, this task is tedious and considered as future work.

Moreover, we will show that our definition of liveness behaves well under the concept of simulatability which has asserted its position as a fundamental 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 [7]. 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 probabilistic reactive systems as introduced by Pfitzmann and Waidner in [7]. Due to lack of space, 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 \tilde{M} of machines $\{M_1, \dots, M_n\}$, one for each user u from a set $\mathcal{M} = \{1, \dots, n\}$, and ideal systems built by one machine $\{\text{TH}\}$.

Communication between different machines is done via ports. Inspired by the CSP-Notation [5], we write output and input ports as $p!$ and $p?$ respectively. The ports of a machine M are denoted by $\text{ports}(M)$. Connections are defined implicitly by naming convention, that is port $p!$ sends messages to $p?$. To achieve asynchronous timing, a message is not directly sent to its recipient, but it is first stored in a special machine \tilde{p} called a buffer and waits to be scheduled. If a machine wants to schedule the i -th message of buffer \tilde{p} (this machine must have the unique clock out port p^{cl}) it simply sends i at p^{cl} , see Figure 1. The buffer then schedules the i -th message and removes it from its internal list. In our case, most buffers are either scheduled by a specific master scheduler or the adversary, i.e., one of those has the clock-out port.

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 [6]. 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 at most produce 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

$$M = (\text{name}_M, \text{Ports}_M, \text{States}_M, \delta_M, l_M, \text{Ini}_M, \text{Fin}_M)$$

of a name $\text{name}_M \in \Sigma^+$, a finite sequence Ports_M of ports, a set $\text{States}_M \subseteq \Sigma^*$ of states, a probabilistic state-transition function δ_M , a length function $l_M : \text{States}_M \rightarrow (\mathbb{N} \cup \{\infty\})^{|\text{in}(\text{Ports}_M)|}$, and sets $\text{Ini}_M, \text{Fin}_M \subseteq \text{States}_M$ of initial and final states. Its input set is $\mathcal{I}_M := (\Sigma^*)^{|\text{in}(\text{Ports}_M)|}$; the i -th element of an input tuple denotes the input at the i -th in-port. Its output set is $\mathcal{O}_M := (\Sigma^*)^{|\text{out}(\text{Ports}_M)|}$. The empty word, ϵ , denotes no in- or output at a port. δ_M maps each pair $(s, I) \in \text{States}_M \times \mathcal{I}_M$ to a finite distribution over $\text{States}_M \times \mathcal{O}_M$. If $s \in \text{Fin}_M$ or $I = (\epsilon, \dots, \epsilon)$, then $\delta_M(s, I) = (s, (\epsilon, \dots, \epsilon))$ deterministically. Inputs are ignored beyond the length bounds,

i.e., $\delta_M(s, I) = \delta_M(s, I \upharpoonright_{l_M(s)})$ for all $I \in \mathcal{I}_M$, where $(I \upharpoonright_{l_M(s)})_i := I_i \upharpoonright_{l_M(s)_i}$ for all i . \diamond

In the text, we often write “M” also for $name_M$. We only briefly state here that these machines have a natural realization as a probabilistic turing machine.

A collection \mathcal{C} of machines is a finite set of machines with pairwise different machine names and disjoint sets of ports. The completion $[\mathcal{C}]$ of a collection \mathcal{C} is the union of all machines of \mathcal{C} and the buffers needed for every 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 $[\mathcal{C}]$ are denoted as $free([\mathcal{C}])$. A collection \mathcal{C} is called *closed* if its completion $[\mathcal{C}]$ has no free ports except a special master clock-in port $clk^{\leftarrow?}$, i.e., $free([\mathcal{C}]) = \{clk^{\leftarrow?}\}$.

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

1. Switch current scheduler: Switch machine M_{CS} , i.e., set $(s', O) \leftarrow \delta_{M_{CS}}(s, I)$ for its current state s and in-port values I . Then assign ϵ to all in-ports of M_{CS} .
2. Termination: If X is in a final state, the run stops.
3. Buffer messages: For each simple out-port $p!$ of M_{CS} , in their given order, switch buffer \tilde{p} with input $p^{\leftarrow?} := p!$, cf. Figure 1. Then assign ϵ to all these ports $p!$ and $p^{\leftarrow?}$.
4. Clean up scheduling: If at least one clock out-port of M_{CS} has a value $\neq \epsilon$, let $q^{\leftarrow!}$ denote the first such port and assign ϵ to the others. Otherwise let $clk^{\leftarrow?} := 1$ and $M_{CS} := X$ and go back to Phase (1).
5. Scheduled message: Switch \tilde{q} with input $q^{\leftarrow?} := q^{\leftarrow!}$ (cf. Figure 1), set $q? := q^{\leftarrow!}$ and then assign ϵ to all

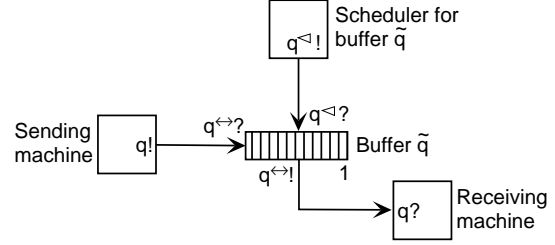


Figure 1. Ports and buffers.

ports of \tilde{q} and to $q^{\leftarrow!}$. Let $M_{CS} := M'$ for the unique machine M' with $q? \in ports(M')$. Go back to Phase (1).

Whenever a machine (this may be a buffer) with name $name_M$ is switched from (s, I) to (s', O) , we add a step $(name_M, s, I, s', O)$ to the run r for $I' := I \upharpoonright_{l_M(s)}$, except if s is final or $I' = (\epsilon, \dots, \epsilon)$. This gives a family of random variables

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

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

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

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

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

and similarly for l -step prefixes. For a singleton $\hat{M} = \{H\}$ we write $view_{\hat{\mathcal{C}}}(H)$ instead of $view_{\hat{\mathcal{C}}}(\{H\})$ 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 $S \subseteq \text{free}([\hat{M}])$ is called specified ports. If \hat{M} is clear from the context, let $\bar{S} := \text{free}([\hat{M}]) \setminus S$. We call $\text{forb}(\hat{M}, S) := \text{ports}(\hat{M}) \cup \bar{S}^c$ the forbidden ports.
- 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.

◇

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 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 liveness definition 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 $conf = (\hat{M}, S, H, A)$ where $(\hat{M}, S) \in Sys$ is a structure, H is a simple machine without forbidden ports, i.e., $\text{ports}(H) \cap \text{forb}(\hat{M}, S) = \emptyset$, and the completion $\hat{C} := [\hat{M} \cup \{H, A\}]$ is a closed collection. The set of configurations is written $\text{Conf}(Sys)$.
- 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 $\text{Ini}'_{\hat{C}} := \{(1^k)_{M \in \hat{C}} \mid k \in \mathbb{N}\}$ of $\text{Ini}_{\hat{C}}$. We write run_{conf} and $\text{view}_{conf}(\hat{M})$ for the families $\text{run}_{\hat{C}}$ and $\text{view}_{\hat{C}}(\hat{M})$ restricted to $\text{Ini}'_{\hat{C}}$, and similar for l -step prefixes. Furthermore, we identify $\text{Ini}'_{\hat{C}}$ with \mathbb{N} and thus write $\text{run}_{conf,k}$ etc. for the individual random variables.
- c) The set of configurations of Sys with polynomial-time user H and adversary A is called $\text{Conf}_{\text{poly}}(Sys)$. The index poly is omitted if it is clear from the context.

◇

We only briefly state here that several machines can be combined into one single machine (which has the original machines as submachines), cf. [7] 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 Simulatability

The definition of one system securely implementing another one is based on the common concept of *simulatability*. Simulatability essentially means that whatever might happen to an honest user in a concrete system Sys_{real} can also happen in an ideal system Sys_{id} . More precisely, for every configuration $conf_1 \in \text{Conf}(Sys_{\text{real}})$, there exists a configuration $conf_2 \in \text{Conf}(Sys_{\text{id}})$ yielding indistinguishable views of the same user in both configurations. We abbreviate this by $Sys_{\text{real}} \geq_{\text{sec}} Sys_{\text{id}}$ and we say that Sys_{real} is “at least as secure” as the system Sys_{id} . A typical situation is illustrated in Figure 2. The notion of simulatability was introduced in [8] and has asserted its position as a fundamental concept of modern cryptography.

However, we do not want to compare a structure $(\hat{M}_1, S_1) \in Sys_{\text{real}}$ 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 $(\text{var}_k)_{k \in \mathbb{N}}$ and $(\text{var}'_k)_{k \in \mathbb{N}}$ of random variables (or probability distributions) on common domains D_k are

- a) perfectly indistinguishable (“=”) if for each k , the two distributions var_k and var'_k are identical.
- b) statistically indistinguishable (“ \approx_{SMALL} ”) for a suitable class $SMALL$ of functions from \mathbb{N} to $\mathbb{R}_{\geq 0}$ if the distributions are discrete and their statistical distances

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

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

- c) computationally indistinguishable (“ \approx_{poly} ”) if for every algorithm Dis (the distinguisher) that is probabilis-

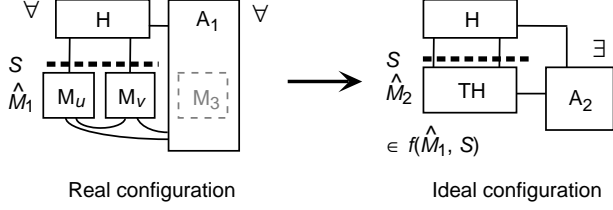


Figure 2. Example of simulatability. The view of H is compared.

tic polynomial-time in its first input,

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

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., $g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0} \in \text{NEGL}$ 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. \diamond

We now present the simulatability definition.

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

- a) We say $\text{Sys}_1 \geq_{\text{sec}}^{f, \text{perf}} \text{Sys}_2$ (perfectly at least as secure as) if for every configuration $\text{conf}_1 = (\hat{M}_1, S, H, A_1) \in \text{Conf}^f(\text{Sys}_1)$, there exists a configuration $\text{conf}_2 = (\hat{M}_2, S, H, A_2) \in \text{Conf}(\text{Sys}_2)$ with $(\hat{M}_2, S) \in f(\hat{M}_1, S)$ (and the same H) such that

$$\text{view}_{\text{conf}_1}(\text{H}) = \text{view}_{\text{conf}_2}(\text{H}).$$

- b) We say $\text{Sys}_1 \geq_{\text{sec}}^{f, \text{SMALL}} \text{Sys}_2$ (statistically at least as secure as) for a class SMALL if the same as in a) holds with $\text{view}_{\text{conf}_1, l}(\text{H}) \approx_{\text{SMALL}} \text{view}_{\text{conf}_2, l}(\text{H})$ for all polynomials l , i.e., statistical indistinguishability of all families of l -step prefixes of the views.
- c) We say $\text{Sys}_1 \geq_{\text{sec}}^{f, \text{poly}} \text{Sys}_2$ (computationally at least as secure as) if the same as in a) holds with configurations from $\text{Conf}_{\text{poly}}^f(\text{Sys}_1)$ and $\text{Conf}_{\text{poly}}(\text{Sys}_2)$ 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_{sec}^f , and we omit the indices f and sec if they are clear from the context. \diamond

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. As we already stated in the introduction, this definition is not suited for schedulers of most cryptographic protocols, since both the adversary and the honest user are polynomially bounded, and the runs as a whole are restricted to polynomial length. Hence, the terminology of infinity does not apply.

What we would like to express is that an enabled process will always be scheduled by the master scheduler after a polynomially bounded number $J(k)$ of the master scheduler's steps unless the master scheduler reaches its runtime bound. We start with an intuitive description of how this can be formalized.

Starting from the t -th view-step of X in one particular trace tr (denoted by $O_{t, tr}$), we search for the first future time $m > t$ such that the first message of \tilde{p} is scheduled (denoted by $(O_{m, tr})_{p^{\dagger}} = 1$). 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 view-steps (i.e., $m - t$) by $\text{wait}(t, tr, p^{\dagger})$. 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 p^{\dagger} . If the master scheduler enters a final state after its m -th view-step without ever outputting 1 at p^{\dagger} we define this number to be $m - t$.

We denote the scheduler as J -fair for a function $J : \mathbb{N} \rightarrow \mathbb{N}$ 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 $\text{ports}(X) \cap \text{forb}(\hat{M}, S) = \emptyset$. 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.

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 $(\hat{M}, S) \in \text{Sys}$ if and only if the following holds:

- X does not connect to an unspecified port of the system, i.e., we have $\text{ports}(X) \cap \text{forb}(\hat{M}, S) = \emptyset$.
- Let $O_{t,tr}$ and $S_{t,tr}$ be X 's output and state after the t -th switching of X , respectively, in every considered trace tr of the configuration. Moreover, let Fin_X denote the set of final states of X . Now we define

$$\text{wait}(t, tr, p^{\triangleleft!}) := \infty$$

if for all $m \in \mathbb{N}$: $(O_{m+t,tr})_{p^{\triangleleft!}} \neq 1$ and $S_{m+t,tr} \notin \text{Fin}_X$, and otherwise

$$\text{wait}(t, tr, p^{\triangleleft!}) := \min_{m \in \mathbb{N}} \{ (O_{m+t,tr})_{p^{\triangleleft!}} = 1 \vee S_{m+t,tr} \in \text{Fin}_X \}.$$

Then we require that

$$\text{wait}(t, tr, p^{\triangleleft!}) \leq J(k)$$

holds for all traces tr of the configuration, all $t \in \mathbb{N}$ and $p^{\triangleleft!} \in \text{ports}(X)$.

If a master scheduler is J -fair for a polynomial J , we call it polynomially fair. In the following, we will augment a master scheduler X by the index fair if it is fair, i.e., we write X_{fair} . \diamond

Remark 3.1. Our definition of 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 \tilde{p} 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 clockout port $p^{\triangleleft!}$ for scheduling this buffer). Now according to our definition, this buffer will always be scheduled after at most $J(k)$ steps for a polynomial J and the security parameter k , 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. \circ

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 $Q_H(k)$ view-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 view-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 $t_{\text{stop}}(k)$, 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.

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 view-steps of the master scheduler and the honest user.

Definition 3.2 (Liveness Configuration) Let an arbitrary system Sys and four functions $t_{\text{stop}}, J, Q_X, Q_H: \mathbb{N} \rightarrow \mathbb{N}$ be given. Furthermore, let a configuration $\text{conf} = (\hat{M}, S, \{H\} \cup \{X_{\text{fair}}\}, A) \in \text{Conf}(\text{Sys})$ be given where X_{fair} is a J -fair scheduler for (\hat{M}, S) . We call this configuration a $(t_{\text{stop}}, J, Q_X, Q_H)$ -liveness configuration if the following holds:

- The honest user H has special ports $\text{stop}_H^!$, $\text{stop}_H^{\triangleleft!}$ which are connected to the adversary, i.e., $\{\text{stop}_H^!, \text{stop}_H^{\triangleleft!}\} \subseteq \text{ports}(H)$ and $\text{stop}_H^? \in \text{ports}(A)$.
- 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 $t_{\text{stop}}(k)$. In this case it outputs (stop) at $\text{stop}_H^!$, 1 at $\text{stop}_H^{\triangleleft!}$. From now on, the user only reads its inputs, but no longer produces

outputs. Similarly, if the adversary gets an input (stop) at stop_H , it only reads its inputs in the future without producing any outputs.

- The user H does not output anything at stop_H except in the above case.
- The number of view-steps of the master scheduler and the honest user is lower-bounded by $Q_X(k)$ and $Q_H(k)$, respectively, for every trace of the configuration.

Liveness configurations will be denoted by $\text{conf}^{\text{live}} = (\hat{M}, S, \{H\} \cup \{X_{\text{fair}}\}, A)^{\text{live}}$, the set of all liveness configurations of a system Sys by $\text{Conf}^{\text{live}}(Sys)$, and the set of all polynomial-time liveness configurations by $\text{Conf}_{\text{poly}}^{\text{live}}(Sys)$. \diamond

Thus, for a given trace tr of a $(t_{\text{stop}}, J, Q_X, Q_H)$ -liveness configuration, we will not have any further outputs of both the user and the adversary after the $t_{\text{stop}}(k)$ -th switching of the user. We use this point in time to split the trace into two parts, a prefix tr_i and a tail extr_i , so that we obtain $tr = tr_i \circ \text{extr}_i$. The tail extr_i is called the *extended trace* or the *run-empty phase*.

Remark 3.2. If the user has exceeded the time $t_{\text{stop}}(k)$ he will never be able to produce any outputs again. In real life, the user will usually resume outputs after the good event has happened, e.g., send reply-messages. Our definition could be modified such that both the honest user and the adversary are “switched on” again after the good event has happened. However, this task is tedious because it significantly complicates our upcoming definitions and the preservation theorem of Section 4, so we will consider it as future work only. \circ

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 a set of traces at the ports in S to each set S with $(\hat{M}, S) \in Sys$. Informally speaking, $Req(S)$ states which are the “good” traces for the given structure (\hat{M}, S) . More precisely, such a trace is a sequence $(v_i)_{i \in \mathbb{N}}$ of values over port names and Σ^* , i.e., sets of port-value pairs so that v_i is of the form $v_i := \bigcup_{p \in \mathcal{I}} \{p : v_{p,i}\}$ for a subset $\mathcal{I} \subseteq S$ 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 $\{p_1^{\triangleleft!}, \dots, p_n^{\triangleleft!}\}$ is a subset of the complement of the specified ports of the structure, i.e., $\{p_1^{\triangleleft!}, \dots, p_n^{\triangleleft!}\} \subseteq S^c$. 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 $t_s : \mathbb{N} \rightarrow \mathbb{N}$. 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 trace of the configuration before the event Req occurs. Therefore, we have to bound the point in time at which the extended trace 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

$$\begin{aligned} \varphi_{Sys} : Sys &\rightarrow \text{LiveProp} \\ (\hat{M}, S) &\mapsto (Req, \{p_1^{\triangleleft!}, \dots, p_n^{\triangleleft!}\}, t_s)_{(\hat{M}, S)} \end{aligned}$$

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

After introducing what polynomial liveness properties are, we have to define what it means that a system fulfills them. Essentially, our complete definition states that a structure fulfills the liveness property $(Req, \{p_1^{\triangleleft!}, \dots, p_n^{\triangleleft!}\}, t_s)$ if the following holds:

If the ports $\{p_1^{\triangleleft!}, \dots, p_n^{\triangleleft!}\}$ 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 $t_{\text{stop}}(k) < t_s(k)$) then there are polynomials Q_X, Q_H such that the event Req will happen within a polynomial number of view-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 $\text{run}_{\text{conf}, k, l(k)} \upharpoonright_{S^H}$ with $S^H := \{p \in S \mid p^c \in \text{ports}(H)\}$ and a polynomial l (cf. Definition 2.3). In slight abuse of notation, we write $Req(S^H)$ instead of $Req(S) \upharpoonright_{S^H}$, i.e., we restrict the trace 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.

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 $LiveReq := (Req, \{p_1^{\triangleleft!}, \dots, p_n^{\triangleleft!}\}, t_s) := \varphi_{Sys}(\hat{M}, S)$

- **perfectly** $((\hat{M}, S) \models^{\text{perf}} LiveReq)$ iff \forall polynomials J, t_{stop} with $t_{\text{stop}} < t_s \exists$ polynomials Q_X, Q_H such that for all $(t_{\text{stop}}, J, Q_X, Q_H)$ -liveness configurations $conf_{\text{live}} = (\hat{M}, S, \{H\} \cup \{X_{\text{fair}}\}, A)^{\text{live}} \in \text{Conf}^{\text{live}}(Sys)$ with $\{p_1^{\triangleleft!}, \dots, p_n^{\triangleleft!}\} \subseteq \text{ports}(X_{\text{fair}})$ the following holds: all $Q_H(k)$ -prefixes of the restriction of the run to the ports in S^H lie in $Req(S^H)$. In formulas,

$$[(run_{conf_{\text{live}}, k, Q_H(k)} \upharpoonright_{S^H})] \subseteq Req(S^H)$$

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

- **statistically** $((\hat{M}, S) \models^{\text{SMALL}} LiveReq)$ iff \forall polynomials J, t_{stop} with $t_{\text{stop}} < t_s \exists$ polynomials Q_X, Q_H such that for all $(t_{\text{stop}}, J, Q_X, Q_H)$ -liveness configurations $conf_{\text{live}} = (\hat{M}, S, \{H\} \cup \{X_{\text{fair}}\}, A)^{\text{live}} \in \text{Conf}^{\text{live}}(Sys)$ with $\{p_1^{\triangleleft!}, \dots, p_n^{\triangleleft!}\} \subseteq \text{ports}(X_{\text{fair}})$ the following holds: the probability that $Req(S^H)$ is not fulfilled after $Q_H(k)$ steps is small, i.e.,

$$P(run_{conf_{\text{live}}, k, Q_H(k)} \upharpoonright_{S^H} \notin Req(S^H)) \in \text{SMALL}.$$

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

- **computationally** $((\hat{M}, S) \models^{\text{poly}} LiveReq)$ iff \forall polynomials J, t_{stop} with $t_{\text{stop}}(k) < t_s(k) \exists$ polynomials Q_X, Q_H such that for all polynomial-time $(t_{\text{stop}}, J, Q_X, Q_H)$ -liveness configurations with $\{p_1^{\triangleleft!}, \dots, p_n^{\triangleleft!}\} \subseteq \text{ports}(X_{\text{fair}})$ the following holds: the probability, that $Req(S^H)$ is not fulfilled after $Q_H(k)$ steps is negligible, i.e.,

$$P(run_{conf_{\text{live}}, k, Q_H(k)} \upharpoonright_{S^H} \notin Req(S^H)) \in \text{NEGL}.$$

We write $(\hat{M}, S) \models (Req, \{p_1^{\triangleleft!}, \dots, p_n^{\triangleleft!}\}, t_s)$ 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 $Sys \models^{\text{perf}} \varphi_{Sys}$, etc. \diamond

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 it 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} . However, we cannot expect the liveness property to automatically carry over to Sys_1 if both systems are completely unrestricted for the following reason: assume that we have given a valid $(t_{\text{stop}}, \cdot, \cdot)$ -liveness configuration $conf_1^{\text{live}} \in \text{Conf}^{\text{live}}(Sys_1)$, so there has to be an indistinguishable configuration $conf_2 \in \text{Conf}(Sys_2)$ because of $Sys_1 \geq^f Sys_2$. However, it is clear that $conf_2$ is not necessarily a $(t_{\text{stop}}, \cdot, \cdot)$ -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. The remaining three parameters are omitted because they must remain unchanged under simulatability since they are part of the honest user.

Hitting the spot, simulatability is not forced to map liveness configurations of the first system to liveness configurations of the second. Thus, given an arbitrary $(t_{\text{stop}}, \cdot, \cdot)$ -liveness configuration $conf_1^{\text{live}} \in \text{Conf}^{\text{live}}(Sys_1)$, we cannot exploit our assumption $Sys_2 \models \varphi_{Sys_2}$ by means of simulatability in order to decide whether or not the considered liveness property holds for the first system. We therefore have to restrict our attention to those systems in which simulatability respects $(t_{\text{stop}}, \cdot, \cdot)$ -liveness configurations, i.e., simulatability yields indistinguishable $(t_{\text{stop}}, \cdot, \cdot)$ -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 ad-

versary 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, fair exchange protocols, and secure group key exchange, 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 $\text{Sys}_1 \geq^f \text{Sys}_2$ holds for a valid mapping f . We then call Sys_1 “at least as secure as” Sys_2 “with respect to liveness” (written $\text{Sys}_1 \geq^{f,\text{live}} \text{Sys}_2$) if the following holds: for a given $(t_{\text{stop}}, J, Q_X, Q_H)$ -liveness configuration $\text{conf}_1^{\text{live}} = (\hat{M}_1, S_1, \{\text{H}\} \cup \{\text{X}_{\text{fair}}\}, \text{A}_1)^{\text{live}} \in \text{Conf}^{\text{live}}(\text{Sys}_1)$ there exists a $(t_{\text{stop}}, J, Q_X, Q_H)$ -liveness configuration $\text{conf}_2^{\text{live}} = (\hat{M}_2, S_2, \{\text{H}\} \cup \{\text{X}_{\text{fair}}\}, \text{A}_2)^{\text{live}} \in \text{Conf}^{\text{live}}(\text{Sys}_2)$ 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, if the important parts of the master scheduler for achieving liveness are identical in both the considered ideal and real system. 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 $\Delta(\phi(\text{var}_k), \phi(\text{var}'_k))$ for a function ϕ of random variables is at most $\Delta(\text{var}_k, \text{var}'_k)$. \square*

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 $\text{Sys}_2 \models \varphi_{\text{Sys}_2}$ holds. Furthermore, let a system Sys_1 be given with $\text{Sys}_1 \geq^{f,\text{live}} \text{Sys}_2$ for a mapping f with $S_1 = S_2$ whenever $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$. Then $\text{Sys}_1 \models \varphi_{\text{Sys}_1}$ for all φ_{Sys_1} with $\varphi_{\text{Sys}_1}(\hat{M}_1, S_1) := \varphi_{\text{Sys}_2}(\hat{M}_2, S_2)$ for an arbitrary structure $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$.*

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

Proof. At first, we show that φ_{Sys_1} is a well-defined liveness property for Sys_1 . Let an arbitrary structure $(\hat{M}_1, S_1) \in \text{Sys}_1$ be given. Simulatability implies that for every structure $(\hat{M}_1, S_1) \in \text{Sys}_1$ 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 $\varphi_{\text{Sys}_2}(\hat{M}_2, S_2) = (\text{Req}, \{\text{p}_1^{\triangleleft!}, \dots, \text{p}_n^{\triangleleft!}\}, t_s)$ is a liveness property for (\hat{M}_2, S_2) . By precondition, we have $S_1 = S_2$, so the integrity requirement Req is well-defined on (\hat{M}_1, S_1) . Moreover, we have $\{\text{p}_1^{\triangleleft!}, \dots, \text{p}_n^{\triangleleft!}\} \subseteq S_1^c$ if and only if $\{\text{p}_1^{\triangleleft!}, \dots, \text{p}_n^{\triangleleft!}\} \subseteq S_2^c$. Therefore, the liveness property $\varphi_{\text{Sys}_1}(\hat{M}_1, S_1)$ 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 would 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 $t_{\text{stop}} < t_s$ such that for every polynomials Q_X, Q_H there exists a $(t_{\text{stop}}, J, Q_X, Q_H)$ -liveness configuration $\text{conf}_1^{\text{live}}$ so that the good event does not occur within $Q_H(k)$ view-steps of the honest user. Because of $\text{Sys}_1 \geq^{f,\text{live}} \text{Sys}_2$ there is a $(t_{\text{stop}}, J, Q_X, Q_H)$ -liveness configuration $\text{conf}_2^{\text{live}} = (\hat{M}_2, S_2, \{\text{H}\} \cup \{\text{X}_{\text{fair}}\}, \text{A}_2)^{\text{live}} \in \text{Conf}^{\text{live}}(\text{Sys}_2)$ for $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$ and for every polynomial Q_X and Q_H such that

$$\text{view}_{\text{conf}_1^{\text{live}}}(\{\text{H}\} \cup \{\text{X}_{\text{fair}}\}) \approx \text{view}_{\text{conf}_2^{\text{live}}}(\{\text{H}\} \cup \{\text{X}_{\text{fair}}\})$$

holds. For the sake of readability we abbreviate $\{\text{H}\} \cup \{\text{X}_{\text{fair}}\}$ by H' and set $S := S_1 := S_2$. Since H is a submachine of H' , we can apply Lemma 4.1 which yields $\text{view}_{\text{conf}_1^{\text{live}}}(\text{H}) \approx \text{view}_{\text{conf}_2^{\text{live}}}(\text{H})$. Moreover, the view of H in both configurations contains the trace at S^{H} , i.e., the trace is a function of the view, so we finally obtain

$$\text{run}_{\text{conf}_1^{\text{live}}}[S^{\text{H}}] \approx \text{run}_{\text{conf}_2^{\text{live}}}[S^{\text{H}}].$$

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 $\text{view}_{\text{conf}_1^{\text{live}}}(\text{H}') = \text{view}_{\text{conf}_2^{\text{live}}}(\text{H}')$ because of $\text{Sys}_1 \geq^{f,\text{live,perf}} \text{Sys}_2$, i.e., the distributions of the views are identical which yields $\text{run}_{\text{conf}_1^{\text{live}}}[S^{\text{H}}] = \text{run}_{\text{conf}_2^{\text{live}}}[S^{\text{H}}]$. Because of $(\hat{M}_2, S_2) \models^{\text{perf}} (\text{Req}, \{\text{p}_1^{\triangleleft!}, \dots, \text{p}_n^{\triangleleft!}\}, t_s)$, there exist two polynomials Q'_X, Q'_H such that

$$[(\text{run}_{\text{conf}_2^{\text{live}},k,Q'_H(k)}[S^{\text{H}}])] \subseteq \text{Req}(S^{\text{H}})$$

holds for every $(t_{\text{stop}}, J, Q'_X, Q'_H)$ -liveness configuration $\text{conf}_2^{\text{live}}$. Thus, for every given $(t_{\text{stop}}, J, Q'_X, Q'_H)$ -liveness configuration $\text{conf}_1^{\text{live}}$, we have an indistinguishable $(t_{\text{stop}}, J, Q'_X, Q'_H)$ -liveness configuration $\text{conf}_2^{\text{live}}$ of (\hat{M}_2, S_2) with the above property. Assume now that (\hat{M}_1, S_1) does not fulfill $(\text{Req}, \{\text{p}_1^{\triangleleft!}, \dots, \text{p}_n^{\triangleleft!}\}, t_s)$. This immediately contradicts the assumption that $[(\text{run}_{\text{conf}_1^{\text{live}},k,Q'_H(k)}[S^{\text{H}}])] \not\subseteq \text{Req}(S^{\text{H}})$ while $[(\text{run}_{\text{conf}_2^{\text{live}},k,Q'_H(k)}[S^{\text{H}}])] \subseteq \text{Req}(S^{\text{H}})$, since $\text{run}_{\text{conf}_1^{\text{live}}}[S^{\text{H}}] = \text{run}_{\text{conf}_2^{\text{live}}}[S^{\text{H}}]$ holds.

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

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

for every polynomial l . If we use the above inequality with $l := Q'_H$ we obtain

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

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

In the computational case, we define a distinguisher $Dist$ as follows: Given a view of machine H , it extracts the $Q'_H(k)$ prefix of the user's view restricted to S^H and verifies if the result lies in $Req(S^H)$. 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 $Req(S^H)$) was required to be polynomial-time decidable. Its advantage in distinguishing is

$$\begin{aligned} & |P(Dist(1^k, view_{conf_1^{live},k}) = 1) \\ & - P(Dist(1^k, view_{conf_2^{live},k}) = 1)| \\ & = |P(run_{conf_1^{live},k,Q'_H(k)} \upharpoonright_{S^H} \notin Req(S^H)) \\ & - P(run_{conf_2^{live},k,Q'_H(k)} \upharpoonright_{S^H} \notin Req(S^H))|. \end{aligned}$$

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. [7] for more details). In real

life, every user u usually has exactly one machine M_u and its machine is correct if and only if the user is honest. The machine M_u of user u has special ports $in_u?$ and $out_u!$ for connecting to the user u . A standard cryptographic system Sys can now be derived by a *trust model*. The trust model consists of an access structure ACC and a channel model χ . If n denotes the number of all participants, then ACC is a set of subsets $\mathcal{H} \subseteq \{1, \dots, n\}$ denoting the possible sets of correct machines. For each set \mathcal{H} there will be exactly one structure built by the machines belonging to the set \mathcal{H} . The channel model classifies every connection as either secure (private and authentic), authenticated or insecure. In the considered model these changes can easily be done via port renaming (cf. [7]).

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 $M_{u,\mathcal{H}}$. We denote the set of them by $\hat{M}_{\mathcal{H}}$ (i.e., $\hat{M}_{\mathcal{H}} := \{M_{u,\mathcal{H}} \mid u \in \mathcal{H}\}$), so real systems are given by $Sys_{real} = \{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}}) \mid \mathcal{H} \in ACC\}$. Ideal systems typically are of the form $Sys_{id} = \{(\{TH_{\mathcal{H}}\}, S_{\mathcal{H}}) \mid \mathcal{H} \in ACC\}$ with the same sets $S_{\mathcal{H}}$ as in the corresponding real system Sys_{real} . The machine $TH_{\mathcal{H}}$ is called *trusted host*.

After this brief review we can turn our attention to the actual reliable system. Both the ideal and real system are based on the systems for secure message transmission introduced in [7] 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 $Sys_{id} = \{(\{TH_{\mathcal{H}}\}, S_{\mathcal{H}}) \mid \mathcal{H} \in ACC\}$, ACC is the powerset of $\{1, \dots, n\}$. The ideal machine $TH_{\mathcal{H}}$ models initialization, sending and receiving of messages. A user u can initialize communications with other users by inputting a command of the form (snd_init) to the port $in_u?$ of $TH_{\mathcal{H}}$. 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 $TH_{\mathcal{H}}$ and a command $(send_blindly, i, l, v)$ 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, $TH_{\mathcal{H}}$ has to wait for a special term $(receive_blindly, v, i)$ or (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 connection between u and v should be initialized, respectively. The user v will receive inputs of the form $(receive, u, m)$ and (rec_init, u) , respectively. If

v is dishonest, $\text{TH}_{\mathcal{H}}$ will simply output (send, m, v) to the adversary. Finally, the adversary can send a message m to a user u by sending a command $(\text{receive}, v, m)$ to the port $\text{from_adv}_u?$ of $\text{TH}_{\mathcal{H}}$ for a corrupted user v , and he can also stop the machine of any user by sending a command (stop) to a corresponding port of $\text{TH}_{\mathcal{H}}$ 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 $\text{rel}_{u,v}$, modeling the “reliable net” in the real world.

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 $\{\text{rel}_{u,v}!, \text{rel}_{u,v}?\} \subseteq \text{ports}(\text{TH}_{\mathcal{H}})$. The corresponding clock port $\text{rel}_{u,v}^{\triangleleft}!$ 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 , $\text{TH}_{\mathcal{H}}$ outputs this message at $\text{rel}_{u,v}!$. 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.

After presenting the main ideas, we can now turn our attention to the actual modifications of the system. Let $n \in \mathbb{N}$ and $\mathcal{M} := \{1, \dots, n\}$ 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 $\text{TH}_{\mathcal{H}}$ mainly has to be modified in the “send message” and “send initialization” transitions. Moreover, we have to define what $\text{TH}_{\mathcal{H}}$ does if it receives an input at one of the special new ports $\text{rel}_{u,v}?$ yielding the following modifications:

- If $\text{TH}_{\mathcal{H}}$ receives an input (snd_init) at port $\text{in}_u?$, it additionally outputs (snd_init) at $\text{rel}_{u,v}!$ for all v with $(u, v) \in \mathcal{I}_{\mathcal{H}}$. However, it still schedules the output intended for the adversary.
- If $\text{TH}_{\mathcal{H}}$ receives an input (send, m, v) at port $\text{in}_u?$, it checks whether $(u, v) \in \mathcal{I}_{\mathcal{H}}$. In this case it addition-

ally outputs $(\text{send_blindly}, i, l, v)$ at $\text{rel}_{u,v}!$, but it still schedules the output intended for the adversary.

- If $\text{TH}_{\mathcal{H}}$ receives an input (snd_init) at $\text{rel}_{u,v}?$ 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 $(\text{rec_init}, u)$ at $\text{out}_v!$.
- If $\text{TH}_{\mathcal{H}}$ receives an input $(\text{send_blindly}, i, l, v)$ at $\text{rel}_{u,v}?$, 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 $(\text{receive}, u, m)$ at $\text{out}_v!$.

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 $\mathcal{M} := \{1, \dots, n\}$ 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

$$\text{Sys}_{\text{id}}^{\text{rel}} = \{(\{\text{TH}_{\mathcal{H}}\}, \mathcal{S}_{\mathcal{H}}) \mid \mathcal{H} \in \mathcal{ACC}\}.$$

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

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

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

Internally, $\text{TH}_{\mathcal{H}}$ maintains arrays $(\text{init}_{u,v}^*)_{u,v \in \mathcal{M}}$ and $(\text{stopped}_u^*)_{u \in \mathcal{M}}$ over $\{0, 1\}$, both initialized with 0 everywhere, and an array $(\text{deliver}_{u,v}^*)_{u,v \in \mathcal{M}}$ of lists, all initially empty. The state-transition function of $\text{TH}_{\mathcal{H}}$ is defined by the following rules, written in a pseudo-code language. For the sake of readability, we exemplarily annotate the first part of the definition, the **Send initialization** part, 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 (snd_init) to $\text{TH}_{\mathcal{H}}$. 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 $\text{stopped}_u^* = 0$, and $\text{init}_{u,u}^* = 0$, respectively. If both checks hold, the keys are distributed over authenticated channels, modeled by an output (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 $\text{rel}_{u,v}$ for $(u,v) \in \mathcal{I}_{\mathcal{H}}$. In our pseudo-code language this is expressed as follows:

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

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

Sending and receiving messages.

- **Send.** On input (send, m, v) at $\text{in}_u?$ with $m \in \Sigma^+$, $l := \text{len}(m) \leq L(k)$, and $v \in \mathcal{M} \setminus \{u\}$: If $\text{stopped}_u^* = 0$, $\text{init}_{u,u}^* = 1$, and $\text{init}_{v,u}^* = 1$: If $v \in \mathcal{A}$ then { output (send, m, v) at $\text{to_adv}_u!$ } else { $i := \text{size}(\text{deliver}_{u,v}^*) + 1$; $\text{deliver}_{u,v}^*[i] := m$. If $(u,v) \in \mathcal{I}_{\mathcal{H}}$ it outputs (send_blindly, i, l, v) at $\text{rel}_{u,v}!$, (send_blindly, i, l, v) at $\text{to_adv}_u!$ and 1 at $\text{to_adv}_u^{\triangleleft}!$. Otherwise it outputs (send_blindly, i, l, v) at $\text{to_adv}_u!$, 1 at $\text{to_adv}_u^{\triangleleft}!$. }.
- **Receive from honest party u .** On input (receive_blindly, u, i) at $\text{from_adv}_v?$ with $u, v \in \mathcal{H}$, $i \in \mathbb{N}$ or (send_blindly, i, l, v) at $\text{rel}_{u,v}?$: If $\text{stopped}_v^* = 0$, $\text{init}_{v,v}^* = 1$, $\text{init}_{u,v}^* = 1$, and $m := \text{deliver}_{u,v}^*[i] \neq \downarrow$, then output (receive, u, m) at $\text{out}_v!$.
- **Receive from dishonest party u .** On input (receive, u, m) at $\text{from_adv}_v?$ with $u \in \mathcal{A}$, $m \in \Sigma^+$, $\text{len}(m) \leq L(k)$, and $v \in \mathcal{H}$: If $\text{stopped}_v^* = 0$,

$\text{init}_{v,v}^* = 1$ and $\text{init}_{u,v}^* = 1$, output (receive, u, m) at $\text{out}_v!$.

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

◇

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

For understanding it is sufficient to give a brief review of Sys_{real} . The system is a standard cryptographic system of the form $\text{Sys}_{\text{real}} = \{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}}) \mid \mathcal{H} \in \mathcal{ACC}\}$. \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 $\text{aut}_{u,v}$. 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 $\text{net}_{u,v}$, 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 (M_u, M_v) of machines with $(u,v) \in \mathcal{I}_{\mathcal{H}}$, and let the master scheduler schedule it. Now the machine M_u behaves just as $\text{TH}_{\mathcal{H}}$ does, i.e., it additionally sends messages over these reliable channels. Obviously, these channels precisely correspond to the channels $\text{rel}_{u,v}$ of the ideal specification, so both system still provide the same functional behaviour for their environment, i.e., for the honest user and the adversary. It would even be nicer and 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 due to lack of space. We will denote the modified real system by $\text{Sys}_{\text{real}}^{\text{rel}}$.

Moreover, if we take a look at the security proof of [7] we can see that the concrete implementation is derived using blackbox simulatability, so our preservation theorem can be applied. Looking at the proof, it is immediately obvious that this still holds for our modified systems.

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

$$\begin{aligned} \{\text{in}_u!^c : (\text{snd_init}), \text{in}_u^{\triangleleft!c} : 1\} &\subseteq \text{tr}_{l_1} && \text{(Key generation of } u \text{ in } \text{tr}_{l_1}) \\ \{\text{in}_v!^c : (\text{snd_init}), \text{in}_v^{\triangleleft!c} : 1\} &\subseteq \text{tr}_{l_2} && \text{(Key generation of } v \text{ in } \text{tr}_{l_2}) \end{aligned}$$

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

$$\begin{aligned} (\text{out}_v?^c : (\text{rec_init}, u)) &\in \text{tr}_{l_3} && \text{(Connection established from } u \text{ to } v \text{ in } \text{tr}_{l_3}) \\ (\text{out}_u?^c : (\text{rec_init}, v)) &\in \text{tr}_{l_4} && \text{(Connection established from } v \text{ to } u \text{ in } \text{tr}_{l_4}) \end{aligned}$$

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

$$\begin{aligned} \{\text{in}_u!^c : (\text{send}, m, v), \text{in}_u^{\triangleleft!c} : 1\} &\subseteq \text{tr}_t && \text{(User } u \text{ sends a message } m \text{ to } v \text{ and schedules it)} \\ \wedge t > l &&& \text{(after the connection has been established)} \\ \wedge m \in \Sigma^+ \wedge \text{len}(m) < L(k) &&& \text{(and the message is valid)} \\ \wedge \forall t_1 < t: (\text{out}_u?^c : (\text{stop})) \notin \text{tr}_{t_1} &&& \text{(and the machine of } u \text{ has not been stopped)} \\ \Rightarrow (\exists t_2 \in \mathbb{N}: (\text{out}_v?^c : (\text{stop})) \in \text{tr}_{t_2}) &&& \text{(then } v\text{'s machine is either stopped in the run)} \\ \vee \exists t_3 > t: &&& \text{(or there is a future time } t_3) \\ (\text{out}_v?^c : (\text{receive}, u, m)) &\in \text{tr}_{t_3} && \text{(such that the message } m \text{ will be delivered in } \text{tr}_{t_3}) \end{aligned}$$

Figure 3. The formula for $\text{Req}(\mathcal{S}_{\mathcal{H}})$.

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 $\text{TH}_{\mathcal{H}}$ 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 $\varphi_{\text{Sys}_{\text{id}}^{\text{rel}}}$. Let an arbitrary structure $(\{\text{TH}_{\mathcal{H}}\}, \mathcal{S}_{\mathcal{H}}) \in \text{Sys}_{\text{id}}^{\text{rel}}$ be given. Then the three components of $\varphi_{\text{Sys}_{\text{id}}^{\text{rel}}}(\{\text{TH}_{\mathcal{H}}\}, \mathcal{S}_{\mathcal{H}})$ are defined as follows.

- 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 trace 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 l th step of the trace tr is denoted by tr_l . For the considered set $\mathcal{S}_{\mathcal{H}}$ of specified ports, we define that a trace tr of an arbitrary configuration is contained in $\text{Req}(\mathcal{S}_{\mathcal{H}})$ if the formula of Figure 3 holds.

- Secondly, we have to specify the set of ports that should be scheduled by the fair master scheduler. For a given set $\mathcal{S}_{\mathcal{H}}$, we define this set to be $\{\text{rel}_{u,v}^{\triangleleft!} \mid (u, v) \in \mathcal{I}_{\mathcal{H}}\}$.
- At last, the function t_s can be chosen arbitrarily as long as it is bounded by a polynomial, i.e., $t_s(k) \in O(k^x)$ for a natural number x .

We can now state our main theorem.

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

Proof. Let an arbitrary structure $(\{\text{TH}_{\mathcal{H}}\}, \mathcal{S}_{\mathcal{H}}) \in \text{Sys}_{\text{id}}^{\text{rel}}$, arbitrary polynomials J, t_{stop} with $t_{\text{stop}} < t_s$ be given. We then define $Q_X(k) := Q_H(k) := t_{\text{stop}}(k) + J(k) \cdot k^x$. Now, let $\text{conf}^{\text{live}} = (\{\text{TH}_{\mathcal{H}}\}, \mathcal{S}_{\mathcal{H}}, \{\text{H}\} \cup \{\text{X}_{\text{fair}}\}, \text{A})^{\text{live}} \in \text{Conf}^{\text{live}}(\text{Sys}_{\text{id}}^{\text{rel}})$ denote an arbitrary $(t_{\text{stop}}, J, Q_X, Q_H)$ -liveness configuration. We have to show that all $Q_H(k)$ prefixes of the restriction of the run to the ports $\mathcal{S}_{\mathcal{H}}^{\text{H}}$ lie in $\text{Req}(\mathcal{S}_{\mathcal{H}})$.

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

We now distinguish between two cases: at first, we 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, $\text{TH}_{\mathcal{H}}$ will output a command (stop) at $\text{out}_v!$ and schedule it by construction, so we have $(\text{out}_v?^c : (\text{stop})) \in \text{tr}_{t_2}$ for one particular $t_2 \in \mathbb{N}$. Hence, the requirement is fulfilled in this case, even before the run-empty phase begins.

At second, we 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 $\text{TH}_{\mathcal{H}}$ 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 $(\text{send_blindly}, i, l, v)$) has been stored in the buffer $\widetilde{\text{rel}}_{u,v}$. Moreover, this buffer can contain at most a polynomial number of messages (bounded by k^x). By precondition, the master scheduler X_{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 $\text{rel}_{u,v}^{\text{!}} \in \text{ports}(X_{\text{fair}})$ holds by assumption, the buffer $\widetilde{\text{rel}}_{u,v}$ has to be scheduled after at most $J(k)$ view-steps of X_{fair} , so the term $(\text{send_blindly}, i, l, v)$ will be scheduled after at most $J(k) \cdot k^x$ view-steps of X_{fair} . If we now take a closer look at the behaviour of $\text{TH}_{\mathcal{H}}$ 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 $\text{deliver}_{u,v}^*[i]$ before). Thus, $\text{TH}_{\mathcal{H}}$ outputs $(\text{receive}, u, m)$ at $\text{out}_v!$, so we have $(\text{out}_v?^c : (\text{receive}, u, m)) \in \text{tr}_{t_3}$ for one particular time $t_3 > t$. Note that our choice of Q_X and Q_H additionally ensures that both X_{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 view-steps of the *user*; by now we only showed that it will happen after a polynomial number of view-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 view-steps the honest user can perform in the run-empty phase is bounded by the number of view-steps of X_{fair} . Therefore, the message will be received after at most $J(k) \cdot k^x$ view-steps of the honest users, counted from the beginning of the run-empty phase, i.e., after at most $t_{\text{stop}}(k) + J(k) \cdot k^x = Q_H$ view-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 $Sys_{\text{real}}^{\text{rel}}$ fulfills the polynomial liveness property $\varphi_{Sys_{\text{real}}^{\text{rel}}}$ computationally, with $\varphi_{Sys_{\text{real}}^{\text{rel}}}$ given as in Theorem 4.1. In formulas, $Sys_{\text{real}}^{\text{rel}} \models^{\text{poly}} \varphi_{Sys_{\text{real}}^{\text{rel}}}$.* □

Proof. Obviously, perfect fulfillment of polynomial liveness implies fulfillment in the computational case. Thus, using Theorem 5.1, we know that $Sys_{\text{real}}^{\text{id}} \models^{\text{poly}} \varphi_{Sys_{\text{real}}^{\text{id}}}$. 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., $Sys_{\text{real}}^{\text{rel}} \geq^{f,\text{live},\text{poly}} Sys_{\text{id}}^{\text{rel}}$. 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.

Acknowledgments.

We thank *Christian Jacobi* for helpful discussions.

References

- [1] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters* 21 (1985) 181-185.
- [2] M. Backes. Cryptographically sound analysis of security protocols. Ph.D thesis, Computer Science Department, Saarland University, 2002, <http://www-krypt.cs.uni-sb.de/~mbackes>.
- [3] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. *Computer Aided Verification (CAV'94)*, Springer-Verlag, June 1994, 68-80.
- [4] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. *Cryptology ePrint Archive*, Report 2001/006, Mar. 2001. <http://eprint.iacr.org>. Full length version of the extended abstract in *Proc. Crypto 2001*.
- [5] C. A. R. Hoare. *Communicating sequential processes*. International Series in Computer Science, Prentice Hall, Hemel Hempstead 1985.
- [6] N. Lynch. *Distributed algorithms*. Morgan Kaufmann Publishers, San Francisco 1996.
- [7] B. Pfitzmann and M. Waidner. A model for asynchronous reactive systems and its application to secure message transmission. *IEEE Symposium on Security and Privacy*, Oakland, May 2001, 184-202.
- [8] A. C. Yao. Protocols for secure computations. *23rd Symposium on Foundations of Computer Science (FOCS) 1982*, IEEE Computer Society, 1982, 160-164.