Safety, Liveness, and Information Flow: Dependability Revisited*

Zinaida Benenson[1] Felix C. Freiling[2] Thorsten Holz[2]
Dogan Kesdogan[3] Lucia Draque Penso[3]

[1] Uppsala University, Department of Information Technology, 751 05 Uppsala, Sweden
[2] Universität Mannheim, Informatik 1, 68131 Mannheim, Germany
[3] RWTH Aachen, Informatik 4, 52056 Aachen, Germany

Abstract: We present a formal framework to talk and reason about dependable systems. The framework is based on three distinct classes of (system specification) properties we call *safety*, *liveness* and *information flow*. We discuss several examples of dependable systems within this framework and argue that these classes are sufficient to model the functional requirements of dependable systems satisfying to high degrees both fault-tolerance and security attributes. The framework is meant to be a minimal security-specific extension of the asynchronous system model from fault-tolerant distributed algorithms and aimed to support teaching the concepts of fault-tolerance and security within a uniform system model. To remain minimal, the framework does not cover probabilistic or complexity theoretic aspects of dependability (like reliability or computational security).

1 Introduction

Following the terminology of Laprie et al. [Lap92], a system is *dependable* if reliance can be justifiably placed on the service it delivers. Dependability includes important quality aspects of systems like fault-tolerance and security. The terminology of Laprie has been very influential in the areas of safety-critical and fault-tolerant systems. But despite several attempts to integrate the view of security better into the terminological framework of dependability [MAF, ALRL04], two distinct views and communities remain.

This paper presents a framework in which it is possible to precisely reason about important aspects of a dependable system specification. The framework comprises a simple system model which allows to formally define terms such as *system* and *failure*, and relate them to others linked to a dependable system requirements' description, such as *property* or *specification*. The framework is mainly based on three classes of system properties which we call *safety*, *liveness*, and *information flow*. These classes can be precisely defined and

*Work by Zinaida Benenson, Thorsten Holz, and Lucia Draque Penso was supported by Deutsche Forschungsgemeinschaft as part of the Graduiertenkolleg "Software für mobile Kommunikationssysteme" at RWTH Aachen University. Work by Felix C. Freiling was supported in part by Deutsche Forschungsgemeinschaft as part of an Emmy Noether scholarship at the Swiss Federal Institute of Technology Lausanne (EPFL), Switzerland.

distinguished within the formal system model. We give several examples of dependable systems in this framework and justify that these classes of system properties are sufficient to describe the functional requirements of dependable systems satisfying to high degrees *both* fault-tolerance and security attributes.

The framework can be understood as a "minimal extension" of the *asynchronous* (or *time-free*) model which dominates large parts of the literature on fault-tolerant distributed algorithms (see for example the book by Lynch [Lyn96]). There, safety and liveness properties alone are regarded as a basis to describe functional requirements of fault-tolerant systems. We extend this pair of properties with a single additional concept, that of information flow properties, which can be shown to fall outside the scope of safety and liveness but which, in our view, are vital to specify security requirements. One of the main goals of our framework is to provide a basis with which to *teach* the concepts related to dependability and to provide structured engineering support on how to specify a dependable system. The price for this minimality is paid in that the framework does not cover probabilistic or complexity theoretic aspects of dependability (like reliability or computational security).

The framework presented has been recently successfully used in a graduate-level lecture "Dependable Distributed Systems" at RWTH Aachen. Apart from that, it may offer some insights and unifying views and may contribute to the ongoing [Die04, Pfi04] discussions within the *GI Fachgruppe "Sicherheit"* on terminology and model issues.

We first present the basics of the model (Section 2), then present several examples of how to talk about dependable systems in the model (Section 3), and finally put the framework into the context of previous and ongoing discussions (Section 4).

2 System Model

We now present the definition of a formal system model, the context in which we talk about dependable systems. Due to a lack of space, we cannot explain the full formalism but stress that all concepts can be fully formalized using, e.g., trace-based concepts from linear temporal logic [MP91, Pnu81].

Systems, traces, and properties. A system is a black box with an interface (see Fig. 1). Everything outside of the system is called the environment. The interface consists of a set of actions which resemble interface operations which can be either invoked by the system (output operations) or by the environment (input operations).

While the interface operations define the "syntax" of a system, its semantics are defined by its externally visible behavior (i.e., the behavior at its interface). The visible behavior is formalized using linear traces. A *trace* is a sequence of interface operations which may happen at the interface of the system interleaved with a sequence of system states. A *trace property* of a system is a set of traces. A system can be regarded as a generator of traces. Depending on internal concurrency and different input operations a system may generate many (possibly infinitely many) different traces. The *system trace property* is the set of all

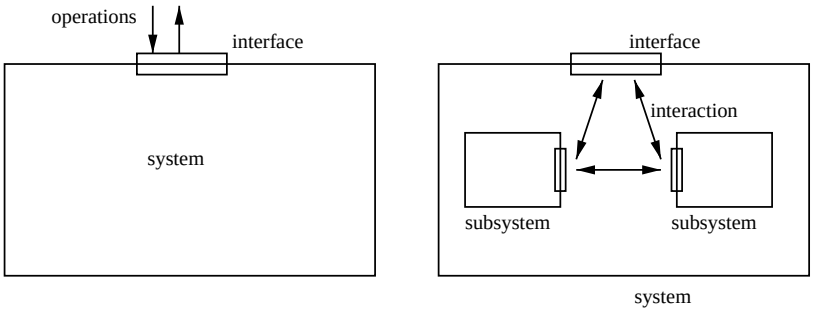

Abbildung 1: A system as a black box (left) and as the composition of subsystems (right).

traces which could possibly be observed at its interface.

As an example, consider a *message system*. For a given set of messages $\mathcal{M}$ such a system has one input operation *Send*(m) and one output operation *Receive*(m) for every $m \in \mathcal{M}$. Basically, *Send*(m) inserts a message into the message system and the operation *Receive*(m') may spontaneously happen at the interface of the system when a message is received. A possible trace of the system is the sequence

$$\{\}, \textit{Send}(m), \{m\}, \textit{Send}(m'), \{m, m'\}, \textit{Receive}(m'), \{m\}, \ldots$$

but also

$$\{\}, \textit{Send}(m'), \{m'\}, \textit{Receive}(m'), \{\}, \textit{Send}(m), \{m\}, \textit{Receive}(m), \{\}, \ldots$$

is a possible trace.

Subsystems. A system can itself be composed of subsystems, which essentially are again systems (see Fig. 1). The algorithms governing the interaction between the subsystems result in the behavior of the composed system at its interface. The recursion of decomposing systems into subsystems must stop at some level which contains *primitive systems*.

Continuing the example, the message system may be composed of a *multiset system*, i.e., a component that implements a multiset[1], and a *activator system*, i.e., a system which has no input operation but one output operation *activate* which it periodically invokes. Invocations of *Send*(m) result in an insertion operation of m into the multiset. If the multiset is not empty and whenever the activator invokes *activate*, we remove some message m' from the multiset and invoke *Receive*(m') at the interface of the message system.

Agents. Since we aim at modeling distributed systems, we postulate a set $\mathcal{A}$ of *agents*. An agent models a process or processor, i.e., an executor of (parts of) systems. Intuitively, systems offer a *vertical* way to structure an application while agents offer a *horizontal*

[1] A multiset is a set which can contain the same element multiple times.

decomposition (see Fig. 2). The set of input and output operations is then tagged with the identity of the agents which a concerned with the operation. For a system which spans several agents, we assume that there is a part of that system executing concurrently on every agent.

In our running example, we can augment the syntax of the *Send* operation to contain source and destination of a message. For example, ***Send***(A, m, B) could denote that agent A sends message m to agent B.

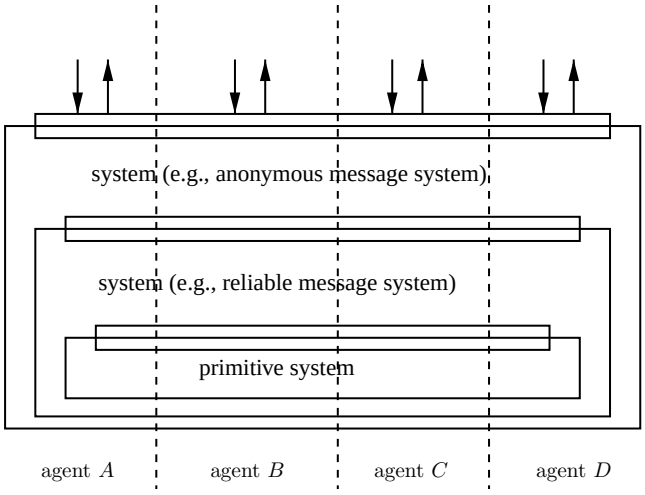

Abbildung 2: Vertical and horizontal structures within systems.

Specifications. A *trace specification* is an intersection of properties, i.e., an intersection of a set of traces, which is itself a set of traces. Intuitively, a specification defines acceptable behavior, namely all traces which are allowed to be generated by a system. For example, we would wish to disallow the following trace (it does not correspond to a sensible trace) of a message system:

$$\{\}, \textbf{\textit{Send}}(A, m, B), \{m\}, \textbf{\textit{Receive}}(B, m', A), \{m\}, \ldots$$

where B receives a message which was never sent.

We will use a natural language based on concepts from temporal logic to express specifications. The two central concepts of that language are the terms “always” and “eventually”. The statement “always φ” refers to all traces where φ holds in every suffix (φ holds to some arbitrary statement). The statement “eventually φ” denotes all traces where at some future point in the trace φ holds. For example, the expression

$$\forall A, B \in \mathcal{A} : \forall m \in \mathcal{M} : \text{ always } \textbf{\textit{Send}}(A, m, B) \text{ implies eventually } \textbf{\textit{Receive}}(B, m, A)$$

refers to all traces where a message m sent by A is eventually received by B.

A system *satisfies* a given is a subset of the set of traces specified by the specification. Of course, in order to correctly implement a specification, a system usually has to execute internal (unobservable) actions within the subsystem which are different from the interface actions. Therefore, depending on the level of abstraction, there can be different views onto a trace. For example, a trace of the message system in our running example may be viewed at the level of the multiset subsystem, yielding traces like

$$\{\}, \textit{Send}(A, m, B), \{m\}, \textit{insert}(A, m, B), \{m\}, \textit{activate}, \{m\}\textit{Receive}(B, m, A), \{m\}, \ldots$$

which at the level of the message system looks like this:

$$\{\}, \textit{Send}(A, m, B), \{m\}, \textit{Receive}(B, m, A), \{\}, \ldots$$

These changing views on traces are known in the literature on *refinement* [Les83].

Faults and adversaries. When talking about dependable systems, we need a way to express faulty or adversarial behavior. We assume that faulty behavior is tied to the notion of an agent. At any point in time, an agent that executes correctly an algorithm in which it is participating is called *correct* (regarding that algorithm). Otherwise, we call the agent *faulty*.

A *fault model* describes what type of faulty behavior we allow from agents. We distinguish two basic types of faulty behavior: *crash* and *arbitrary*. The crash model allows an agent to simply stop executing steps whereas the arbitrary model (sometimes also called *Byzantine* [LSP82]) allows an agent to act in arbitrary ways.

In a security context we treat all faulty agents to be under the control of an intelligent adversary. We define a hierarchy of three increasingly powerful adversary classes which specify in addition to the types of faulty behavior also the amount of information which is visible to the adversary. For the exposition of this paper it suffices to mention attacker class A1:

A1 The adversary sees only events which occur at the interfaces of faulty agents (this corresponds to an adversary who sees only the behavior of faulty agents).

Safety and liveness. In 1977, Lamport [Lam77] observed that there are two fundamental classes of trace properties, namely *safety* and *liveness* properties. Briefly spoken, safety properties describe what is not allowed to happen, e.g., "no message is received unless it was sent". Liveness properties demand what eventually must happen, e.g., "every message which is sent is eventually received". Later, Alpern and Schneider [AS85] came up with formal definitions of safety and liveness. They also were able to prove a *decomposition theorem*, i.e., that every dependable property could be written as the intersection of a safety and a liveness property. This theorem is a useful guideline for dependable system designers to cleanly structure system specifications.

In the area of specifying and verifying security properties, trace-based formalisms have been adopted, too (see for example Gray III. and McLean [GM95] or Mantel and Sabelfeld [MS01]). *Possibilistic security properties* [McL94, McL90] use a particular type of

trace-based formalization to specify the absence of information flow in multi-level security environments. The idea of this approach is that information cannot be deduced by observing the system because the set of traces which might have generated this observation is too large. Interestingly, possibilistic security properties fall outside of the safety/liveness classification because they describe *properties* of trace sets, and hence must be formalized as a *set of sets of traces* [McL96].

Information flow. Mantel [Man00] proposed a modular formalism to specify information flow properties using closure conditions on trace sets. Interface events are separated into two classes H and L (usually referred to as *high* and *low*) with the idea that observing events from L leaks no information about events from H.

In our example, we may wish to specify that the fact that agent A sent a message to agent B remains confidential from other agents (this is sometimes called *sender anonymity*). For this, we separate all interface events on agents A and B into the set H and all other operations into set L (i.e., all interface operations of the other agents including the *activate* operation of the multiset system). To specify anonymity of the message system, the set of traces P generated by the system must satisfy the following closure property: For any trace σ in P which includes *Send*(A, m, B) and *Receive*(B, m, A) there must exist another trace σ' in P which is the same as σ but with all occurrences of *Send*(A, m, B) and *Receive*(B, m, A) removed.

Why does this closure condition specify anonymity? The idea is that other agents do not see the interface operations of A and B (those operations in H), i.e., they can only guess from what they themselves see (operations in L) as to what A and B are doing. If the message system had just one single behavior, namely the trace

$$\textit{Send}(A, m, B), \textit{insert}(A, m, B), \textit{activate}, \textit{Receive}(B, m, A), \ldots$$

then the other agents would simply know from observing the *activate* operation (which is in L) that a message has been sent between A and B. If for any such trace there is another trace which only contains *activate*, then by observing operations from L it is now impossible to deduce whether *Send*(A, m, B) or *Receive*(B, m, A) happened. Note that it still may be possible to deduce whether these operations did *not* happen.

Information flow properties fall outside of the safety and liveness domain, and hence, they are not subject to the Abadi-Lamport Composition Principle [AL93]. These properties can therefore not be composed in general. The framework proposed by Mantel offers the possibility to compose systems if certain conditions are met [Man03].

3 Talking About Dependable Systems

In this section, the core of the paper, we present several examples of dependable systems and discuss them within our framework. The aim of this section is to show that it is possible to talk and reason about classical fault-tolerance (i.e., safety and liveness) properties as well as security (i.e., information flow) properties *at the same time* and *in a natural way*.

3.1 A Reliable Message System

A reliable message system, which was used as running example in the previous section, allows any two agents to exchange messages in a reliable manner. It is a component that can be specified without any information flow properties.

The reliable message system abstraction has been used extensively in the literature on fault-tolerant distributed algorithms (see for example the book by Lynch [Lyn96]). Two agents, the sender S and the receiver R, want to exchange messages from a given set $\mathcal{M}$. There are two types of interface operations: *Send*(S, m, R) is invoked by S whenever it wants to send a message $m \in \mathcal{M}$ to R. The operation *Receive*(S, m, R) happens at R when the message finally arrives at its destination.

Formally, we specify a *reliable message system* as a system satisfying the following properties:

- (Liveness) If *Send*(S, m, R) happens and if S and R are both correct, then eventually *Receive*(R, m, S) will happen.
- (Safety) If S and R are both correct, *Receive*(R, m, S) does not happen unless previously a corresponding *Send*(S, m, R) happened.
- (Safety) *Receive*(R, m, S) happens at most once.

In this context, *corresponding* means that if the sender S sends a message $m \in \mathcal{M}$ to the receiver R invoking *Send*(S, m, R), then the operation *Receive*(R, m, S) can happen later on in the trace. Note that Safety and Liveness specify two different aspects of the channel: Liveness describes the *progress* of the channel, i.e., the promises it makes towards the *future*. Safety expresses that no messages may be received which were not previously sent, i.e., it restricts what may have happened in the *past*. Note that the channel is asynchronous, there are no time bounds on the delivery delay. Such time bounds, however, can also be expressed (in this case as an additional safety property).

3.2 An Anonymous Reliable Message System

On top of a reliable message system we now construct a message system that satisfies a special security property, namely *sender anonymity*.

3.2.1 Specification

Similar to a reliable message system, an anonymous reliable message system has two interface operations *AnonSend* and *AnonReceive*. The properties are also similar apart from one additional information flow property which has to be satisfied. Formally, a *anonymous reliable message system* satisfies the following properties:

- (Liveness) If *AnonSend*(S, m, R) happens and if S and R are correct, then eventually *AnonReceive*(S, m, R) will happen.
- (Safety) If S and R are correct, *AnonReceive*(S, m, R) does not happen unless previously a corresponding *AnonSend*(S, m, R) happened.
- (Safety) *Receive*(R, m, S) happens at most once.
- (Information Flow) If S and R are correct, the system leaks no information about the fact that S sent a message to R to any agent who is not S or R.

Note that in a security setting, the safety property guarantees some weak form of authenticity: upon arrival of a message m, it guarantees that m was in fact sent by S and m is unchanged (m was sent and no other message m'). However, it provides no means to identify the *identity* of the sender S.

3.2.2 An Implementation

An anonymous reliable message system can be implemented using a reliable message system and standard privacy enhancing techniques, namely DC networks [Cha88]. We briefly sketch an implementation here.

A DC network is time slotted and in each time slot all participating agents send a message, although for successful transmission only one of them has to be the real message and the others are dummy messages. Now the task is to hide the real message in the cover of the dummy messages. For this task the agents exchange secret keys along a given key graph, each agent then locally exclusive-ors (XORs) all of the keys it has with the dummy or real message which it is about to send. After sending, every agents receives a message from every other agents. The sent message is recovered by XORing all received messages together.

Note that the implementation of a DC network requires a time-slotted execution of send operations by the agents. This can be achieved in asynchronous systems by employing a network synchronizer [Awe85, GP02]. Note also that in contrast to the safety and liveness properties of the reliable message system, the validation of the information flow property is much harder to achieve. In fact, the framework has not adapted yet the rigorous proof techniques developed for proving the absence of information flow [Man03]. Developing intuitive guidelines for validating these properties is still future work.

4 Dependability Revisited

The three classes of safety, liveness and information flow have interesting parallels to the classification of security requirements known as the “CIA taxonomy” which seems to have its origin in a paper by Voydock and Kent [VK83] and later appeared very prominently in the harmonized IT security criteria of France, Germany, the Netherlands, and the United

Kingdom which are known as “ITSEC” [ITS91]. “CIA” sees security as the combination of three attributes: confidentiality, integrity and availability. There have been many discussions on whether or not the CIA spectrum really covers all relevant security properties. For example, *accountability* is often treated as a property aside from CIA, e.g., by Stelzer [Ste90], whereas others (e.g., Rannenberg *et al.* [RPM99] or Amoroso [Amo94]) argue that it falls into the domain of integrity. The framework presented in this paper offers a new perspective onto the CIA taxonomy. Availability has some resemblance to liveness properties (at least if non-real-time availability, i.e., termination in finite time, is concerned). Integrity properties have parallels to the class of safety properties (they describe the non-occurrence of “bad” things). Finally, confidentiality properties are concerned with secrecy, i.e., they have a large overlap with information flow properties. However note that our framework does not allow to express aspects of safety and security which refer to complexity or probabilities. For example, the concept of *reliability* (e.g., mean time to failure) or measures of attacker effort [Cac02] cannot be formalized in our model. Also, the existing work on possibilistic information-flow has not yet been extended to model confidentiality achieved through encryption.

We have used the framework in the course “Dependable Distributed Systems” at RWTH Aachen University. From our experience, the triangle of safety, liveness and information flow offers a model which is simple enough to explain many fundamental concepts in the design of dependable systems. For this domain it is a good compromise of expressiveness and simplicity.

Acknowledgements

We wish to thank Rachid Guerraoui for his inspirational lecture notes on fault-tolerant distributed algorithms and Heiko Mantel for valuable comments on a previous version of this paper.

Literatur

[AL93] Martín Abadi und Leslie Lamport. Composing Specifications. *ACM Trans. Progr. Lang. and Sys.*, 15(1):73–132, Jan. 1993.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell und Carl E. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dep. Sec. Comput.*, 1(1):11–33, 2004.

[Amo94] Edward Amoroso. *Fundamentals of Computer Security Technology*. Prentice Hall, 1994.

[AS85] Bowen Alpern und Fred B. Schneider. Defining liveness. *Inf. Proc. Lett.*, 21:181–185, 1985.

[Awe85] Baruch Awerbuch. Complexity of Network Synchronization. *J. ACM*, 32(4):804–823, Oct. 1985.

[Cac02] Christian Cachin. Modeling complexity in secure distributed computing. In *Proc. Int. Workshop on Future Directions in Distr. Comp. (FuDiCo)*, Bertinoro, Italy, 2002.

[Cha88] David Chaum. The Dining Cryptographers Problem: Unconditional sender and recipient untraceability. *J. Cryptology*, 1(1):65–75, 1988.

[Die04] Rüdiger Dierstein. Sicherheit in der Informationstechnik—der Begriff IT-Sicherheit. *Informatik Spektrum*, 27(4):343–353, Aug. 2004.

[GM95] James W. Gray, III. und John McLean. Using Temporal Logic to Specify and Verify Cryptographic Protocols. In *Proc. Eighth Comp. Sec. Found. Workshop (CSFW '95)*, pages 108–117, Jun. 1995.

[GP02] Felix C. Gärtner und Stefan Pleisch. Failure detection sequencers: Necessary and sufficient information about failures to solve predicate detection. In *Proc. 16th Int. Symp. on Distr. Comp. (DISC 2002)*, number 2508 in LNCS, pages 280–294, Toulouse, France, Oct. 2002.

[ITS91] Information Technology Security Evaluation Criteria (ITSEC). Version 1.2, Juni 1991.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3(2):125–143, Mar. 1977.

[Lap92] Jean-Claude Laprie, Hrsg. *Dependability: Basic concepts and Terminology*, Number 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, 1992.

[Les83] Leslie Lamport. What good is Temporal Logic? In *Proc. IFIP Congress on Inf. Proc.*, pages 657–667, Amsterdam, 1983.

[LSP82] L. Lamport, R. Shostak und M. Pease. The Byzantine generals problem. *ACM Trans. Progr. Lang. and Sys.*, 4(3):382–401, Jul. 1982.

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Mateo, CA, 1996.

[MAF] MAFTIA Home – Malicious- and Accidental-Fault Tolerance for Internet Applications. Internet: `http://www.newcastle.research.ec.org/maftia/`.

[Man00] Heiko Mantel. Possibilistic Definitions of Security - An Assembly Kit. In *Proc. 13th Comp. Sec. Found. Workshop (CSFW 2000)*, Cambridge, England, Juli 2000.

[Man03] Heiko Mantel. *A Uniform Framework for the Formal Specification and Verification of Information Flow Security*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 2003.

[McL90] John McLean. Security models and information flow. In *Proc. Symp. Research in Security and Privacy*, pages 180–187, Oakland,CA, 1990.

[McL94] John McLean. Security Models. In *Encyclopedia of Software Engineering*. John Wiley & Sons, 1994.

[McL96] John McLean. A General Theory of Composition for a Class of "Possibilistic" Properties. *IEEE Trans. Softw. Eng.*, 22(1):53–67, Jan. 1996.

[MP91] Zohar Manna und Amir Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer-Verlag, 1991.

[MS01] Heiko Mantel und Andrei Sabelfeld. A Generic Approach to the Security of Multi-threaded Programs. In *Proc. 14th Comp. Sec. Found. Workshop (CSFW 2001)*, pages 126–142, Cape Breton, Nova Scotia, Canada, Jun. 2001.

[Pfi04] Andreas Pfitzmann. Why safety and security should and will merge. In *Proc. 23rd Int. Conf. SAFECOMP 2004*, number 3219 in LNCS, pages 1–2, Potsdam, Germany, Sep. 2004. Invited talk.

[Pnu81] Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[RPM99] Kai Rannenberg, Andreas Pfitzmann und Günter Müller. IT Security and Multilateral Security. In Günter Müller und Kai Rannenberg, Hrsg., *Multilateral Security in Communications — Technology, Infrastructure, Economy*, pages 21–29. Addison-Wesley, 1999.

[Ste90] Dirk Stelzer. Kritik des Sicherheitsbegriffs im IT-Sicherheitsrahmenkonzept. *Datenschutz und Datensicherheit*, 14(10):501–506, Oct. 1990.

[VK83] Victor L. Voydock und Stephen T. Kent. Security Mechanisms in High-Level Network Protocols. *ACM Comp. Surveys*, 15(2):135–171, Jun. 1983.