Formally testing fail-safety of Electronic Purse Protocols [*] Extended Abstract

Jan Jürjens
Computing Laboratory, University of Oxford[†]

Guido Wimmel
Department of Computer Science, Munich University of Technology[‡]

Abstract

Designing and implementing security-critical systems correctly is very difficult. In practice, most vulnerabilities arise from bugs in implementations. We present work towards systematic specification-based testing of security-critical systems using the CASE tool AutoFocus.

Cryptographic systems are formally specified with state transition diagrams, a notation for state machines in the AutoFocus system. We show how to systematically generate test sequences for security properties based on the model that can be used to test the implementation for vulnerabilities. In particular, we focus on the principle of fail-safety.

We explain our method at the example of a part of the Common Electronic Purse Specifications (CEPS), a candidate for an international electronic purse standard.

Most commonly, attacks address vulnerabilities in the way security mechanisms are used, rather than the mechanisms themselves. Being able to treat security aspects with a general CASE tool within the context of system development enables detection of such vulnerabilities.

1 Introduction

Correct design and implementation of security-critical systems that are part of an open network is a difficult task. In practice, most vulnerabilities arise from bugs in implementations [And01]. It would be highly desirable to gain confidence in the protection of implemented security-critical systems against attacks.

Towards this goal we present work for systematically testing security-critical systems. The idea is to specify the system (at the abstract design level) using a formal specification language and to use this specification to generate test-sequences to find security weaknesses in an implementation in a systematic way. In the current work (which is part of a wider effort reported previously in [Jür01b, WW01, JW01c, JW01b]) we concentrate on one classical principle of computer security engineering, namely that of *fail-safety* of security-critical systems [SS75]. This principle postulates that, if a security-critical system fails, it should do so in a secure state. What this means exactly in the

[*] This work was partially supported by the Studienstiftung des deutschen Volkes, and by the German Ministry of Economics within the FairPay project

[†], tel. +44 1865 284104, fax +44 1865 273839 - Wolfson Building, Parks Road, Oxford OX1 3QD, Great Britain

[‡], tel. +49 89 289 28362, fax +49 89 289 25310 - TU München, 80290 München, Germany

system context depends on the system at hand and the security aspect under consideration. When considering access control, for example, fail-safety means to base access decisions on permission rather than exclusion [SS75]. When considering payment protocols, as we do here, it means that interruption of the protocol leaves the participants in a secure state (eg., no more funds are given out of the system than are taken in). More specifically, we use the CASE tool AUTOFOCUS [HMR$^+$98, HMS$^+$98] developed for design and formal verification of distributed systems to formally specify the unlinked load transaction of the Common Electronic Purse Specifications (CEPS) [CEP01]. We use this specification to generate test-sequences for implementations of the protocol. CEPS is a candidate for a globally interoperable electronic purse standard supported by organisations (including Visa International) representing 90 percent of the world's electronic purse cards and likely to become an accepted standard, making its security an important goal.

As well-known, testing cannot *prove* the absence of implementation errors. It is however currently the technique most widely used in industry to gain some confidence in the absence of major bugs, since mechanically assisted theorem proving or model-checking of code have thus far been perceived as being limited in the size of treatable systems and as being comparatively costly.

The effectiveness of testing depends crucially on the ability to identify adequate test strategies. This is very difficult when testing for security requirements, since it is not sufficient to establish that no failures will occur *most of the time*, as the remaining, non-tested situations that lead to failures must be assumed to be found by motivated attackers and then be systematically exploited. Rather, one needs to establish that certain security-critical parts of the system are indeed free from failures under all conceivable attack attempts from the system environment. The current work aims to provide some guidance on how to do this in a systematic way.

In this extended abstract, we can only give some short remarks about the work undertaken; for more details, as well as a discussion of related work, cf. [JW01a].

2 Specification of CEPS Load Transaction

For an overview over the Common Electronic Purse Specifications (CEPS) and an explanation of the load transaction cf. [Jür01a]. Here we only shortly present our formal specification, for a more detailed explanation cf. [JW01a].

We specified the CEPS load transaction (slightly simplified by leaving out security irrelevant details) and its participating components with help of the formal CASE tool AUTOFOCUS/Quest. AUTOFOCUS [HMR$^+$98, HMS$^+$98] is a tool for graphically specifying distributed systems. It is based on the formal method Focus, and the models have a simple formally defined semantics. AUTOFOCUS supports different views on the system model, describing structure, data types, behaviour and interactions. These views are related to UML-RT diagrams. In addition to modelling, AUTOFOCUS offers simulation, code generation, test sequence generation and formal verification of the modelled systems.

The system components involved in the CEPS Load transaction and their communication links are shown in Figure 1, as an AUTOFOCUS **system structure diagram**. Here, the components are depicted as rectangles — in our model, Card, LSAM, and Issuer.

Components can have input and output ports (drawn as empty and filled circles) used to send and receive messages. These ports are connected via communication channels.

To model security-critical systems, a communication channel can be marked with a "**public**" tag. This denotes that the communication over this channel can be manipulated by another entity. This way, faulty channels or channels that are subject to attacks can be modelled — which we will use later for test se-

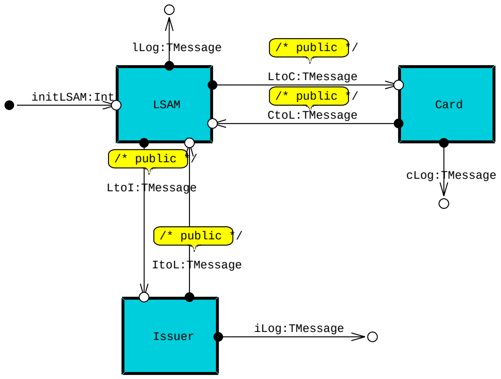

Figure 1. System structure diagram for Load transaction

quence generation. The "public" channels were introduced in earlier work extending AUTOFOCUS for security modelling (see [WW01]).

The channels between the LSAM and the Card and Issuer components in our case are public. However, the components also generate log information (channels cLog,lLog,iLog), which can not be manipulated. The values carried by the chanels are of type TMessage, which models the possible messages passed between the components. AUTOFOCUS supports hierarchical data types, i.e. a value of type TMessage can for instance be "Init(x)" with x being an integer, or "Enc(k, msg)", with k being a key and msg being a message of type TMessage itself.

The **behaviour** of the components is given as a state transition diagram (STDs). STDs are extended finite automata (meaning they can have a data state represented by local variables). The transitions are annotated with precondition, input statements (reading a value from the input port, if the pattern in the input statement matches that value), output statements, and assignments to local variables — separated by colons ":".

Representatively, the state transition diagrams in Figure 3 specifies the behaviour of the LSAM. The other components can be found in [JW01a]. Figure 2 shows a message sequence chart (MSC) describing the interactions for a successful protocol run. This MSC was generated from the model using the simulation feature of AUTOFOCUS.

2.1 Formally deriving test-sequences

With the help of the AUTOFOCUS model, we can now test the resistance of an implementation of the CEPS load transaction against threats. We use the approach of specification-based testing, as advocated in e.g. [WLPS00, PLP01]. For this purpose, test case specifications based on the system model have to be formulated. Test specifications would be, for example, that a certain log entry should be generated, certain data is sent on the channels, or a component should reach a success or failure state. The test specification and the model are translated into logic and their conjunction is solved. The solutions are all test sequences of a given maximum length satisfying the test case specification. These test sequences represent concrete system executions, and can be depicted as message sequence charts. To test the system, the inputs contained in the test sequence are fed into the system components and it is verified if the output is as expected. Test sequence generation can also be used to validate and correct the specification: if the test sequence itself contains an unexpected system run (e.g. there should be no execution fulfilling the test case specification, but the test sequence generation computed one), this indicates an er-

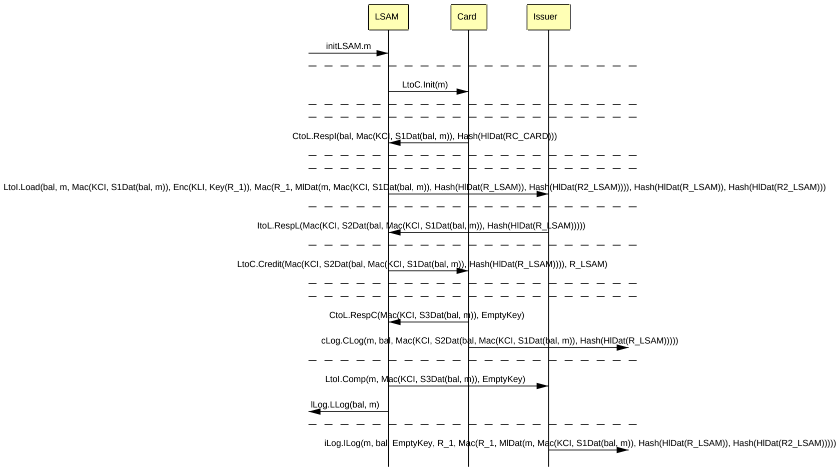
LSAM
Card
Issuer
initLSAM.m
LtoC.Init(m)
CtoL.RespI(bal, Mac(KCI, S1Dat(bal, m)), Hash(HlDat(RC_CARD)))
LtoI.Load(bal, m, Mac(KCI, S1Dat(bal, m)), Enc(KLI, Key(R_1)), Mac(R_1, MlDat(m, Mac(KCI, S1Dat(bal, m)), Hash(HlDat(R_LSAM)), Hash(HlDat(R2_LSAM)))), Hash(HlDat(R_LSAM)), Hash(HlDat(R2_LSAM)))
ItoL.RespL(Mac(KCI, S2Dat(bal, Mac(KCI, S1Dat(bal, m)), Hash(HlDat(R_LSAM)))))
LtoC.Credit(Mac(KCI, S2Dat(bal, Mac(KCI, S1Dat(bal, m)), Hash(HlDat(R_LSAM)))), R_LSAM)
CtoL.RespC(Mac(KCI, S3Dat(bal, m)), EmptyKey)
cLog.CLog(m, bal, Mac(KCI, S2Dat(bal, Mac(KCI, S1Dat(bal, m)), Hash(HlDat(R_LSAM)))))
LtoI.Comp(m, Mac(KCI, S3Dat(bal, m)), EmptyKey)
lLog.LLog(bal, m)
iLog.ILog(m, bal, EmptyKey, R_1, Mac(R_1, MlDat(m, Mac(KCI, S1Dat(bal, m)), Hash(HlDat(R_LSAM)), Hash(HlDat(R2_LSAM)))))

Figure 2. MSC for Successful Load Transaction

ror in the model.

For classical specification based testing, the main emphasis of testing is on normal system behaviour (e.g., for certain inputs, the correct result is computed). When security aspects come into consideration, this is turned around: the system has to behave in a secure way even in case it is under attack. Thus, in testing we have to assume that system components may act maliciously.

We did this by including a threat model into the system specification: channels marked "**public**" are unreliable, and can be accessed and manipulated by an intruder. The intruder is modelled by an additional component, through which all messages must pass. The corresponding state transition diagram specifies what the intruder can do. This threat scenario (see Fig. 4) can be generated automatically from the system model.

The Smart Card Protection Profile [Gro01] of the Common Criteria lists the following threats relevant to fail-safety of a smart-card scheme:

Forced Reset : An attacker may corrupt Target of Evaluation Security Function (TSF) data through inappropriate termination of selected operations.

Insertion of Faults : An attacker may determine user and TSF information though observation of the results of repetitive insertion of selected data.

Invalid Input : An attacker may compromise the TSF data through introduction of invalid inputs.

Environmental Stress : An attacker may introduce errors in the TSF data through exposure of the TOE to environmental stress.

Correspondingly, we consider the following two threat scenarios:

(1) the attacker can only pass on the messages or drop message parts (replace them by Empty)

(2) the attacker can pass on the messages, or replace them by own messages not containing secret keys he does not know in advance.

The first scenario corresponds to the situation where the adversary may interrupt the communication between the different protocol participants at some point (*Forced Reset*, e.g. by pulling out the card). The second scenario models the case that the adversary may force one of the involved cards to behave in an arbitrary way (by *Insertion of Faults*, *Invalid Input*,

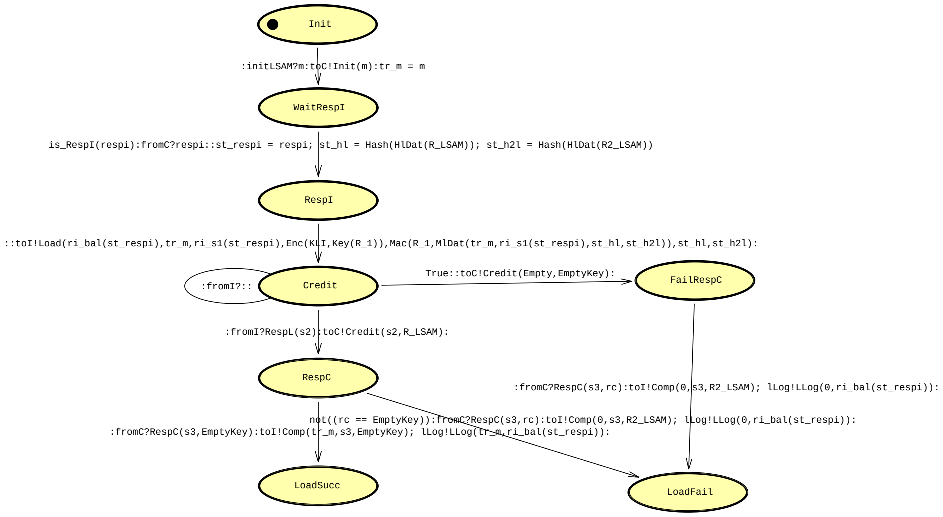
Init
:initLSAM?m:toC!Init(m):tr_m = m
WaitRespI
is_RespI(respi):fromC?respi::st_respi = respi; st_hl = Hash(HlDat(R_LSAM)); st_h2l = Hash(HlDat(R2_LSAM))
RespI
::toI!Load(ri_bal(st_respi),tr_m,ri_s1(st_respi),Enc(KLI,Key(R_1)),Mac(R_1,MlDat(tr_m,ri_s1(st_respi),st_hl,st_h2l)),st_hl,st_h2l):
:fromI?::
Credit
True::toC!Credit(Empty,EmptyKey):
FailRespC
:fromI?RespL(s2):toC!Credit(s2,R_LSAM):
RespC
:fromC?RespC(s3,rc):toI!Comp(0,s3,R2_LSAM); lLog!LLog(0,ri_bal(st_respi)):
not((rc == EmptyKey)):fromC?RespC(s3,rc):toI!Comp(0,s3,R2_LSAM); lLog!LLog(0,ri_bal(st_respi)):
:fromC?RespC(s3,EmptyKey):toI!Comp(tr_m,s3,EmptyKey); lLog!LLog(tr_m,ri_bal(st_respi)):
LoadSucc
LoadFail

Figure 3. STD for LSAM

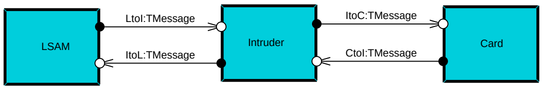

Figure 4. Threat Scenario (for LSAM and Card)

or *Environmental Stress* – such as heat). This may have the result that the card sends arbitrary messages instead of the intended ones, which may involve keys stored on the card, but it is unlikely that the misbehaving card "guesses" unknown keys.

The attacker is specified directly in constraint logic — taking advantage of the possibility to use unbound variables and predicates on these to model generation of (a restricted set of) arbitrary messages by the attacker. The Prolog interpreter then automatically finds those attacker messages corresponding to a given test scenario.

Now we can generate test sequences from the specification that correspond to executions when the system is under attack. The main remaining problem is that we now have a very large number of potential test sequences. As mentioned before, it is much more difficult to test systems for the absence of undesired than for the presence of desired behaviour. There are very many executions where the system fails — which should we choose to cover as many different attack situations as possible ?

We can take advantage of the fact that we know

- firstly, the security requirements (as given in the CEPS specifications) and
- secondly, we know which parts of the model (e.g. which states and transitions) relate to these requirements.

We indicate in the following how to employ this knowledge in the situation of our example.

Firstly, the CEPS specifications contain the following requirements on the behaviour of the protocol participants relevant to fail-safety:

(1) RC_{CARD} is sent by the card to the LSAM if the card experiences an error.

(2) In case the LSAM experiences an error, either $s3$ or $R2_{LSAM}$ are sent by the LSAM to the issuer.

(3) If there is no response to the $s1$ sent to issuer, the LSAM must send $R2_{LSAM}$.

(4) $R2_{LSAM}$ is not sent out if the card balance incremented.

(5) The LSAM performs only one of the following two events:

- $s2$ and R_{LSAM} are sent to the card or
- $R2_{LSAM}$ is sent to the issuer.

The implementation can be checked wrt. these requirements by generating test sequences. For example, for the first requirement we compute a test sequence from the model so that RC_{CARD} is sent by the card to the LSAM, which corresponds to an error at the card. This test sequence can then be used to verify if the implementation has the same behaviour.

Secondly, one can consider test case specifications based on the structure of the model:

(i) Compute a concrete execution where one of the components reaches the LoadSucc state. In particular, the test sequence reflects the fact that no other component reaches the LoadFail state (validating the model), and the implementation can be tested with respect to this.

(ii) Analogous to the above, one can compute test sequences where one of the component reaches the LoadFail state and verify that no other components then reach the LoadSucc state, even in presence of an attacker.

(iii) More specifically, for any of the security-critical transitions to LoadFail or LoadSucc one may compute test sequences so that this transition is executed.

(iv) One can compute test sequences with respect to attacker activity. E.g. messages are manipulated at certain points in time or a certain number of times.

As an example, Figure 5 shows a test sequence derived from the model corresponding to the class of specifications (5) given above: the test case is that an R2_LSAM is sent to the issuer log because of a failure of the card. In this case, $s2$ and R_LSAM are not sent to the card, and all three components stop together in their LoadFail states. The above test sequence consists of 24 steps (executions of transitions) and is computed in approximately 10 seconds by the test sequence generator. Briefly, the test sequence proceeds as follows: $R2_{LSAM}$ is sent to the issuer log because of a failure. In the computed test sequence, the failure occurs after the LSAM sent the Load message to the issuer. The LSAM sends the message $\mathsf{Credit}(\mathsf{cEmpty}, \mathsf{cEmptyKey})$ to the card to cancel the transaction, and the response RespL from the issuer is dropped by the intruder (intruder reports dropI0!cPresent). The messages RespC and Comp with cancellation information are sent from the card via the LSAM back to the issuer, and all three components report the failure to their logs.

3 Conclusion and Future Work

We used the distributed systems CASE tool AUTOFOCUS to generate test-sequences for security aspects of the currently developed Common Electronic Purse Specifications (CEPS) from formal specifications. This gives a systematic way of doing security testing. Here we concentrated on the principle of *fail-safety* from the classical security literature [SS75]. Since security vulnerabilities often arise from bugs in the implementation, having a systematic way to eliminate security-critical bugs is a worth-while goal.

Future work includes the development of a test case specification language which can be compiled “intelligently” into test cases by ap-

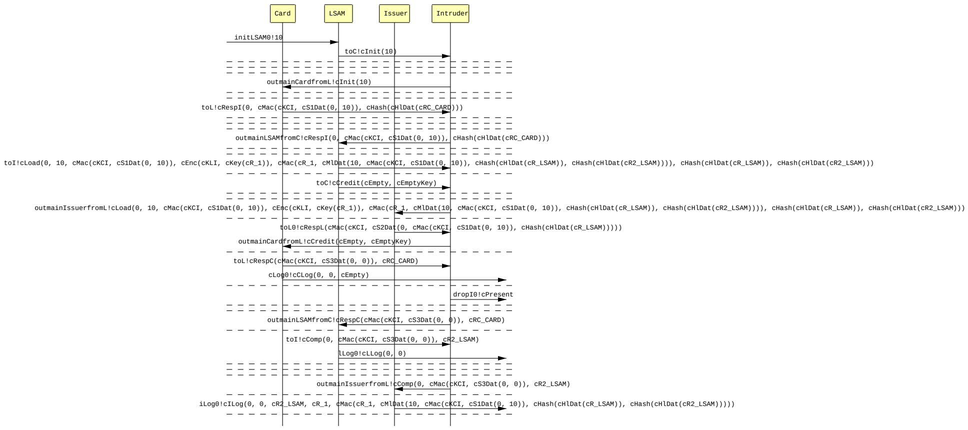
Card
LSAM
Issuer
Intruder
initLSAM0!10
toC!cInit(10)
outmainCardfromL!cInit(10)
toL!cRespI(0, cMac(cKCI, cS1Dat(0, 10)), cHash(cHlDat(cRC_CARD)))
outmainLSAMfromC!cRespI(0, cMac(cKCI, cS1Dat(0, 10)), cHash(cHlDat(cRC_CARD)))
toI!cLoad(0, 10, cMac(cKCI, cS1Dat(0, 10)), cEnc(cKLI, cKey(cR_1)), cMac(cR_1, cMlDat(10, cMac(cKCI, cS1Dat(0, 10)), cHash(cHlDat(cR_LSAM)), cHash(cHlDat(cR2_LSAM)))), cHash(cHlDat(cR_LSAM)), cHash(cHlDat(cR2_LSAM)))
toC!cCredit(cEmpty, cEmptyKey)
outmainIssuerfromL!cLoad(0, 10, cMac(cKCI, cS1Dat(0, 10)), cEnc(cKLI, cKey(cR_1)), cMac(cR_1, cMlDat(10, cMac(cKCI, cS1Dat(0, 10)), cHash(cHlDat(cR_LSAM)), cHash(cHlDat(cR2_LSAM)))), cHash(cHlDat(cR_LSAM)), cHash(cHlDat(cR2_LSAM)))
toL0!cRespL(cMac(cKCI, cS2Dat(0, cMac(cKCI, cS1Dat(0, 10)), cHash(cHlDat(cR_LSAM)))))
outmainCardfromL!cCredit(cEmpty, cEmptyKey)
toL!cRespC(cMac(cKCI, cS3Dat(0, 0)), cRC_CARD)
cLog0!cCLog(0, 0, cEmpty)
dropI0!cPresent
outmainLSAMfromC!cRespC(cMac(cKCI, cS3Dat(0, 0)), cRC_CARD)
toI!cComp(0, cMac(cKCI, cS3Dat(0, 0)), cR2_LSAM)
lLog0!cLLog(0, 0)
outmainIssuerfromL!cComp(0, cMac(cKCI, cS3Dat(0, 0)), cR2_LSAM)
iLog0!cILog(0, 0, cR2_LSAM, cR_1, cMac(cR_1, cMlDat(10, cMac(cKCI, cS1Dat(0, 10)), cHash(cHlDat(cR_LSAM)), cHash(cHlDat(cR2_LSAM)))))

Figure 5. Test Sequence for Load transaction

plying optimisation depending on the test case specification in question.

Acknowledgement Part of this work has been presented in a course "Principles of Secure Systems Design" at the Oxford University Computing Laboratory, Trinity term 2001; feedback from the participants is gratefully acknowledged.

References

[And01] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 2001.

[CEP01] CEPSCO. Common Electronic Purse Specifications, 2001. Business Requirements vers. 7.0, Functional Requirements vers. 6.3, Technical Specification vers. 2.3, available from http://www.cepsco.com.

[Gro01] Smart Card Security User Group. Smart card protection profile. Common Criteria for Information Technology Security Evaluation, 21 March 2001. Draft Version 2.1d.

[HMR+98] F. Huber, S. Molterer, A. Rausch, B. Schätz, M. Sihling, and O. Slotosch. Tool supported Specification and Simulation of Distributed Systems. In *International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 155–164, 1998.

[HMS+98] F. Huber, S. Molterer, B. Schätz, O. Slotosch, and A. Vilbig. Traffic Lights – An AutoFocus Case Study. In *1998 International Conference on Application of Concurrency to System Design*, pages 282–294. IEEE Computer Society, 1998.

[Jür01a] Jan Jürjens. Object-oriented modelling of audit security for smart-card payment schemes. In P. Paradinas, editor, *IFIP/SEC 2001 – 16th International Conference on Information Security*. Kluwer, 2001.

[Jür01b] Jan Jürjens. Secrecy-preserving refinement. In *Formal Methods Europe*, LNCS. Springer, 2001.

[JW01a] J. Jürjens and G. Wimmel. Formally testing fail-safety of electronic purse protocols (long version), 2001.

[JW01b] Jan Jürjens and Guido Wimmel. Security modelling for electronic commerce: The Common Electronic Purse Specifications. In *First IFIP conference on e-commerce, e-business, and e-government (I3E)*. Kluwer, 2001.

[JW01c] Jan Jürjens and Guido Wimmel. Specification-based testing of firewalls. In *Andrei Ershov 4th International Conference "Perspectives of System Informatics" (PSI'01)*, LNCS. Springer, 2001. To be published.

[PLP01] A. Pretschner, H. Lötzbeyer, and J. Philipps. Model Based Testing in Evolutionary Software Development. In *Proc. 11th IEEE Intl. Workshop on Rapid System Prototyping (RSP'01)*, Monterey, June 2001.

[SS75] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[WLPS00] G. Wimmel, H. Lötzbeyer, A. Pretschner, and O. Slotosch. Specification Based Test Sequence Generation with Propositional Logic. *Journal on Software Testing Verification and Reliability*, 10, 2000.

[WW01] G. Wimmel and A. Wißpeitner. Extended description techniques for security engineering. In *IFIP SEC*, 2001.