Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 220 (2008) 93–104
www.elsevier.com/locate/entcs

Model-based Security Testing Using UMLsec A Case Study

Jan Jürjens

Computing Department, The Open University, GB

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 based on UMLsec models. 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. 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.

Keywords: Model-based Testing, UML, Security, UMLsec

1 Introduction

Modern society and economy rely on infrastructures for communication, finance, energy distribution, and transportation. These infrastructures depend increasingly on networked information systems. This leads to vulnerabilities, for the exploitation of which there have recently been a number of widely publicised examples. 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. 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. More specifically, we use UMLsec [11,12] to specify the unlinked load transaction of the Common

⋆ . This work was partially funded by the Royal Society within the project Model-based Formal Security Analysis of Crypto Protocol Implementations.

doi:10.1016/j.entcs.2008.11.008

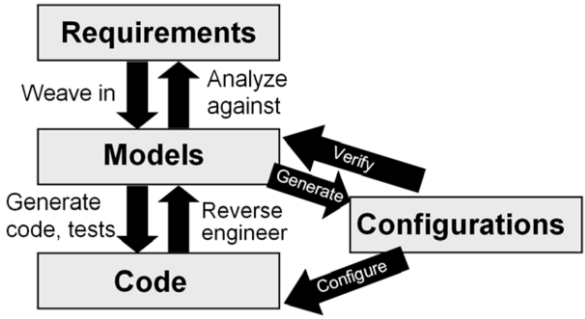

Fig. 1. Model-based Security Engineering

Electronic Purse Specifications (CEPS) [5]. 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, 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.

The work presented here is part of a more general approach towards model-based security engineering visualized in Fig. 1 (see [11]).

In Section 2, we give an overview over the Common Electronic Purse Specification, specify the part under consideration and explain the security threat model. In Sect. 3 we explain our use of the UMLsec tool to generate test-sequences that examine the security of the above specification. In Sect. 4 we refer to related work. We end with a conclusion and indicate further planned work.

2 CEPS

We give an overview over the Common Electronic Purse Specifications. Stored value smart cards ("electronic purses") have been proposed to allow cash-free point-of-sale (POS) transactions offering more fraud protection than credit cards: Their built-in chip can perform cryptographic operations which allows transaction-bound authentication (whereas credit card numbers are valid until the card is stopped,

which enables misuse). The card contains an account balance that is adjusted when loading the card or purchasing goods. The Common Electronic Purse Specifications (CEPS) define requirements for a globally interoperable electronic purse scheme providing accountability and auditability. The specifications outline overall system security, certification and migration. For more detail on the functionality of CEPS cf. [5].

Here we consider a central part of CEPS, the (unlinked, cash-based) load transaction, which allows the cardholder to load electronic value onto a card in exchange for cash at a load device belonging to the load acquirer. The participants involved in the transaction protocol are the customer's card, the load device and the card issuer. The load device contains a Load Security Application Module (LSAM) that is used to store and process data (and is assumed to be tamper-resistant). During the transaction, the account balance in the card is incremented, and the amount is logged in the LSAM and sent to the issuer for later financial settlement between the load acquirer and the card issuer. Load transactions in CEPS are on-line transactions using symmetric cryptography for authentication. We only consider unlinked load, where the cardholder pays cash into a, possibly unattended, loading machine and receives a corresponding credit on the card. Linked load, where funds are transferred for example from a bank account, the so-called funds issuer, is viewed as offering fewer possibilities for fraud, because funds are moved only within one financial institution [5, Funct. Req. p. 12].

To perform a cash-based load transaction, the cardholder inserts his card into the card reader and the money into the cash slot of the load device. To load the cash on the card, he enters the PIN. Note again that the cardholder is not able to communicate with the card directly, but only through the display of the load device. A Load Secure Application Module (LSAM) is used to provide the necessary cryptographic and control processing. The LSAM may reside within the load device or at the load acquirer host. The load acquirer keeps a log of all transactions processed. Through the load host application, the LSAM communicates with the card issuer. Below, we analyze the load protocol between the card, the LSAM, and the card issuer that is executed after the cardholder inserts the cash.

Specification: We give a specification of the CEPS load transaction, slightly simplified by leaving out security-irrelevant details, but including exception processing. The specification is given in form of the UML subsystem $\mathcal{L}$. Here we show as fragments the class and statechart diagrams in Fig. 2 to 5. The values exchanged in the protocol are listed in Fig. 6.

We use the notation *var* ::= *exp* as a syntactic short-cut. Here *var* is a local variable not used for any other purpose and *exp* may not contain *var*. Before assigning a semantics to the diagram, the variable *var* should be replaced by the expression *exp* at each occurrence. Also, for increased readability, we use pattern matching: for example, $(\mathsf{lda}', \mathsf{m}') ::= \mathsf{Init}$ means that when deriving the formal semantics of the sequence diagram, one would have to replace lda' with Init_1 and m' with Init_2 in each case.

The link between the LSAM and the loading device, and the loading device itself,

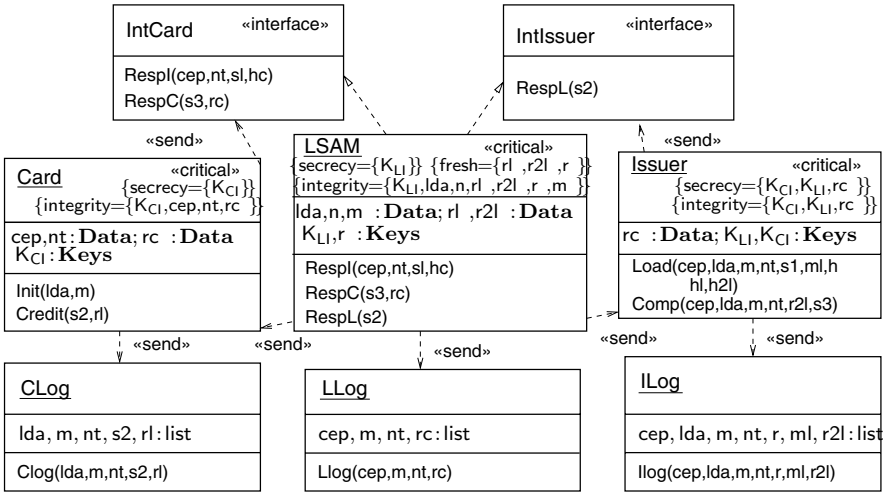

Fig. 2. Load transaction class diagram

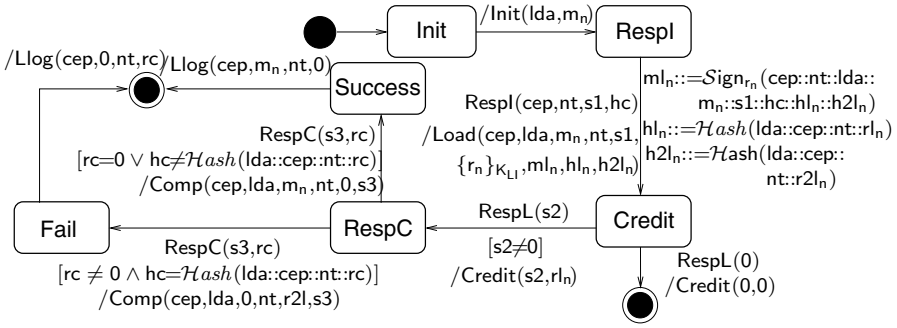

Fig. 3. Load transaction: load acquirer

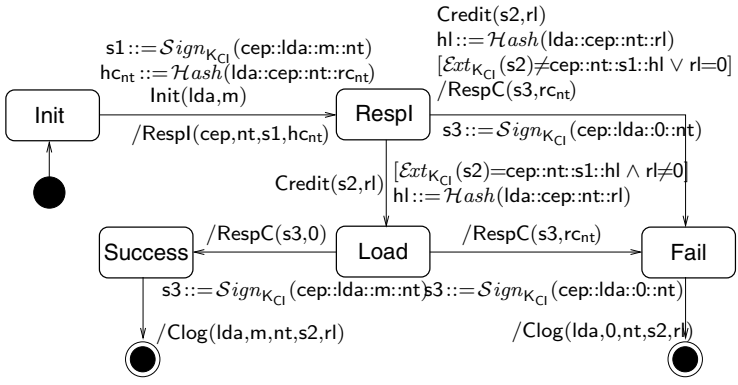

Fig. 4. Load transaction: card

need to be secured. Otherwise an attacker could initiate the protocol without having inserted cash into the machine. For simplicity, we leave out the communication between the LSAM and loading device to determine the amount to be loaded, but assume that the amount is communicated to the LSAM in a secure way. Here, a

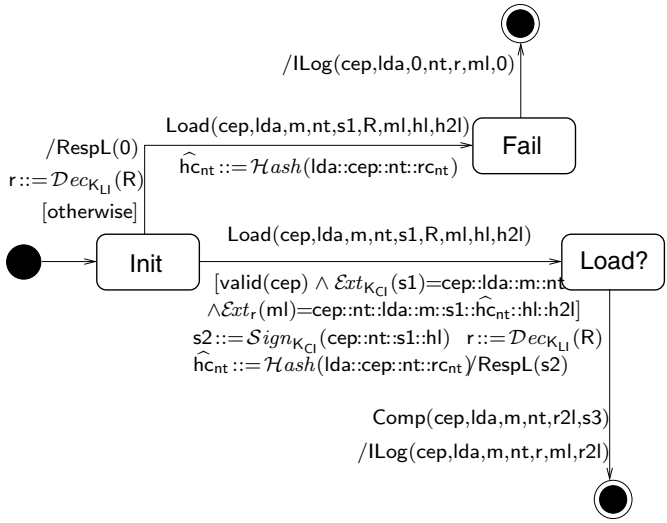

Fig. 5. Load transaction: card issuer

CEP card name cep is called valid if the name is registered at the card issuer and not on the list of revoked cards.

For the participants of the protocol, we have the classes Card, LSAM, and Issuer. Also, each of the three classes has an associated class used for logging transaction data named CLog, LLog, and ILog, respectivly. The logging objects simply take the arguments of their operations and update their attributes accordingly.

We assume a sequence of random values $\mathsf{rc_{nt}}$ to be given that is shared between the card C and its card issuer I. These random values are required to be fresh within the Load subsystem as indicated by the tag {fresh} attached to Load. Note that when viewing the Load subsystem in isolation, the associated condition is vacuous: It just requires that any appearance of an expression rc_x in Load must be in Load. Using the {fresh} tag at a top-level subsystem is still meaningful, because one may want to include the subsystem in another subsystem also stereotyped «data security», which would extend the scope of the freshness constraint to the larger subsystem. In this example, it would not make sense to attach the {fresh} tag with value rc to any of the objects in Load, because the random values are supposed to be shared among Card and Issuer. As usual, we write rc : **Data** to denote an array with fields in **Data**. Also given are the random numbers $\mathsf{rl_n}$, $\mathsf{r2l_n}$ and the symmetric keys $\mathsf{r_n}$ of the LSAM. These values are also supposed to be generated freshly by the LSAM. In fact, one can see that expressions of the form rl_x, $\mathsf{r2l}_x$, r_x, for any subexpression x, only appear in the object and the statechart associated with LSAM. Again, the keys and random values are independent of each other and of the other expressions in the diagram. Also, again, constant attributes have their initial values as attribute names and the corresponding attribute types are underlined. Finally, we are given the transaction amounts $\mathsf{m_n}$. Before the first protocol run, the card and LSAM initialize the card transaction number nt and the acquirer-generated identification number n, respectively. Also, before each protocol run, the card and LSAM increment the card transaction number nt and

Variable	Explanation
C	card
L	LSAM
I	card issuer
$\mathsf{rc_{nt}}$	secret random values shared between card and issuer
$\mathsf{rl_n}$, $\mathsf{r2l_n}$	random numbers of LSAM
$\mathsf{r_n}$	symmetric keys of LSAM
$\mathsf{m_n}$	transaction amounts
m, rl, hl	$\mathsf{m_n}$, $\mathsf{rl_n}$, $\mathsf{hl_n}$ as received at card issuer
nt	card transaction number
n	acquirer-generated identification number
lda	load device identifier
cep	card identifier
$\mathsf{s1}$	card signature: $\mathcal{S}ign_{\mathsf{K_{CI}}}(\mathsf{cep}::\mathsf{lda}::\mathsf{m}::\mathsf{nt})$
$\mathsf{hc_{nt}}$	card hash value: $\mathcal{H}ash(\mathsf{lda}::\mathsf{cep}::\mathsf{nt}::\mathsf{rc_{nt}})$
$\widehat{\mathsf{hc_{nt}}}$	$\mathsf{hc_{nt}}$ as created at issuer
rc, hc	$\mathsf{rc_{nt}}$, $\mathsf{hc_{nt}}$ as received at load acquirer
$\mathsf{K_{CI}}$	key shared between card and issuer
$\mathsf{K_{LI}}$	key shared between LSAM and issuer
$\mathsf{ml_n}$	$\mathcal{S}ign_{\mathsf{r_n}}(\mathsf{cep}::\mathsf{nt}::\mathsf{lda}::\mathsf{m_n}::\mathsf{s1}::\mathsf{hc}::\mathsf{hl_n}::\mathsf{h2l_n})$ (signed by LSAM)
$\mathsf{hl_n}$	hash of transaction data: $\mathcal{H}ash(\mathsf{lda}::\mathsf{cep}::\mathsf{nt}::\mathsf{rl})$
$\mathsf{h2l_n}$	hash of transaction data: $\mathcal{H}ash(\mathsf{lda}::\mathsf{cep}::\mathsf{nt}::\mathsf{r2l})$
$\mathsf{s2}$	issuer signature: $\mathcal{S}ign_{\mathsf{K_{CI}}}(\mathsf{cep}::\mathsf{nt}::\mathsf{s1}::\mathsf{hl})$
$\mathsf{s3}$	card signature of the form $\mathcal{S}ign_{\mathsf{K_{CI}}}(\mathsf{cep}::\mathsf{lda}::\mathsf{m}::\mathsf{nt})$

Fig. 6. Values exchanged in the load specification

the acquirer-generated identification number n, repectively, as long as a given limit is not reached (to avoid the rolling over of the numbers).

Security Threat Model: We derive the following security conditions:

Cardholder security: If the card appears to have been loaded with a certain amount according to its logs, the cardholder can prove to the card issuer that there is a load acquirer who owes the amount to the card issuer.

Load acquirer security: A load acquirer has to pay an amount to the card issuer only if the load acquirer has received the amount in cash from the cardholder.

Card issuer security: The sum of the balances of the cardholder and the load acquirer remains unchanged by the transaction.

Note that the correct functioning of the settlement scheme relies on the fact that the cardholder should only be led to believe that a certain amount has been correctly loaded (for example, when checking the card with a portable cardreader) if the cardholder is later able to *prove* this using the card. Otherwise the load acquirer could first credit the card with the correct amount, but later in the settlement process claim that the cardholder tried to fake the transaction.

Properties to be tested: We turn to the formalizations of the above security conditions, which should be tested using the model-based testing approach. We focus on the condition providing security for the load acquirer. According to the CEPS, the value $\mathsf{ml_n}$, together with the value $\mathsf{rl_n}$ sent in the CreditforLoad message to the card, is taken as a guarantee that the amount m specified in $\mathsf{ml_n}$ has to be paid by the specified load acquirer to the issuer of the specified card, unless it is negated with the value $\mathsf{rc_{nt}}$ [5, Tech. Spec. 6.6.1.6]. The security condition is thus formalized as follows:

Load acquirer security: Suppose that the card issuer I possesses the value $\mathsf{ml_n} = \mathcal{S}ign_{\mathsf{r_n}}(\mathsf{cep} :: \mathsf{nt} :: \mathsf{lda} :: \mathsf{m_n} :: \mathsf{s1} :: \mathsf{hc_{nt}} :: \mathsf{hl_n} :: \mathsf{h2l_n})$ and that the card C possesses $\mathsf{rl_n}$, where $\mathsf{h_n} = \mathcal{H}ash(\mathsf{lda} :: \mathsf{cep} :: \mathsf{nt} :: \mathsf{rl_n})$. Then after execution of the protocol either of the following two conditions hold:

- a message $\mathsf{Llog}(\mathsf{cep}, \mathsf{lda}, \mathsf{m_n}, \mathsf{nt})$ has been sent to $\mathsf{I} : \mathsf{LLog}$ (which implies that L has received and retains $\mathsf{m_n}$ in cash) or
- a message $\mathsf{Llog}(\mathsf{cep}, \mathsf{lda}, 0, \mathsf{nt})$ has been sent to $\mathsf{I} : \mathsf{LLog}$ (that is, the load acquirer assumes that the load failed and returns the amount $\mathsf{m_n}$ to the cardholder) and the load acquirer L has received $\mathsf{rc_{nt}}$ with $\mathsf{hc_{nt}} = \mathcal{H}ash(\mathsf{lda} :: \mathsf{cep} :: \mathsf{nt} :: \mathsf{rc_{nt}})$ (thus negating $\mathsf{ml_n}$).

3 Generating test-sequences

With the help of the UMLsec model, we can now test the resistance of an implementation of the CEPS load transaction against threats using a specification-based testing approach. 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

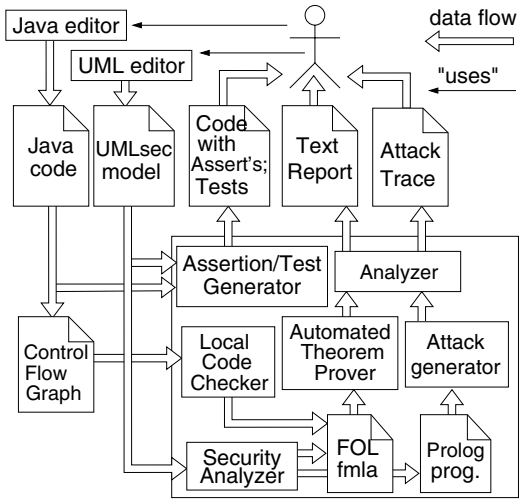

Fig. 7. Model-based Security Tool Suite

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 error 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: public channels are vulnerable, and can be accessed and manipulated by an intruder. The intruder is modelled by a Prolog program which is generated from the UMLsec specification using the UMLsec tool [18,13,10] (Fig. 7).

The Smart Card Protection Profile [7] 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 Target of Evaluation (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*, 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 generated as part of the generated Prolog program which 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 ?

A direction to do this is to use the model (states in the automata and transitions) as a basis for test case specification. However, unlike in the general case, we can take advantage of the fact that we know which parts of the model relate to the security requirement to provide fail-safety, so emphasis of testing can be focused on these.

The CEPS specifications contain the following requirements on the behaviour of the protocol participants relevant to fail-safety (cf. Fig. 6 for an explanation of the variables):

(1) rc_{nt} 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 $r2l_n$ are sent by the LSAM to the issuer.

(3) If there is no response to the $s1$ sent to issuer, the LSAM must send $r2l_n$.

(4) $r2l_n$ is not sent out if the card balance incremented.

(5) The LSAM performs only one of the following two events:
- $s2$ and rl_n are sent to the card or
- $r2l_n$ 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_{nt} 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 in this case.

Additionally 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

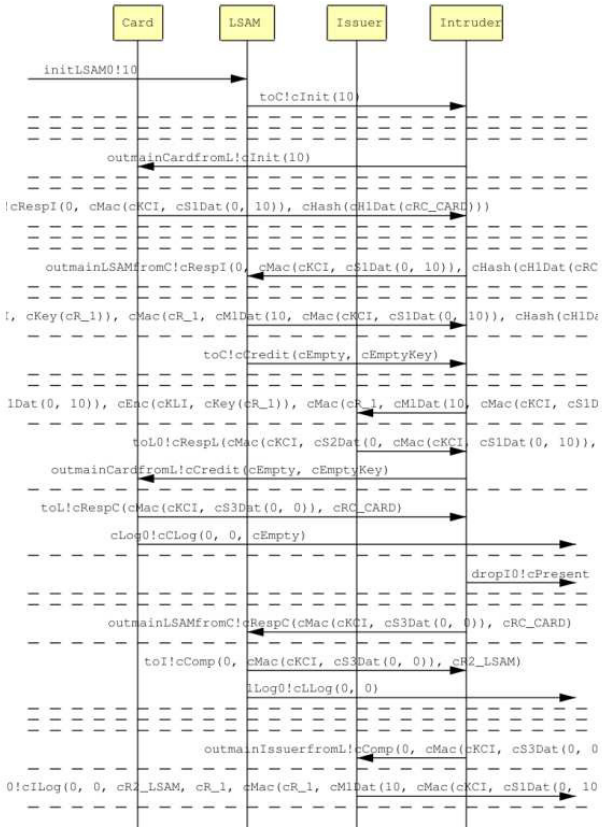

Fig. 8. Test Sequence for Load transaction

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.

The implementation tested in this case-study is a prototypical implementation of part of the CEPS specifications in Javacard (consisting of 600 KB of source code altogether) available from the UMLsec tool website [18]. Due to space restrictions, a more detailed discussion of the results from this case-study has to be deferred to a longer version of this paper.

As an example, Fig. 8 shows a test sequence derived from the model corresponding to the class of specifications (5) given above: the test case is that an $\mathsf{r2l_n}$ is sent to the issuer log because of a failure of the card. In this case, $s2$ and $\mathsf{rl_n}$ 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: $r2l_n$ 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 Comp to the card to cancel the transaction, and the response RespL from the issuer is dropped by the intruder. The messages RespC 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.

4 Related Work

There has been an increasing amount of work on the interaction between formal methods and testing, see [3,16,8,2,14,6] for examples and overviews.

The work presented here is an extension of earlier work [19] (which uses the CASE tool AutoFocus) towards using the UMLsec tool suite. Despite the title, the paper [17] is not really about model-based testing of cryptographic protocols in our usage of the term "model-based testing" (i.e., generate test-sequences from models) but rather about soundness and completeness of symbolic models of cryptographic protocols with respect to computational complexity models, and about using the SpecExplorer for model-checking Spec-sharp models of cryptographic protocols. The approach proposed in [15] deals with the problem of establishing whether or not a security property expressed using an observer formalised as an input/output labelled transition system (IOLTS) holds in an IOLTS providing a black-box specification of the system. The central idea of using a Dolev-Yao based model-verification approach to generate the traces is similar to that followed in [19] (but other techniques are added, such as "learning by testing"). Otherwise, work on model-based testing of cryptographic protocol implementations against security requirements is limited. Since the usage of cryptography poses particular challenges, work on testing other kinds of security-critical software (such as database management systems [4]) is not directly applicable here. Also, since the goal of our work is on testing high-level security requirements (such as secrecy and authenticity), work towards finding buffer overflow weaknesses in implementations (such as [1]) is not directly comparable to our work.

5 Conclusion

We used the UMLsec tool support to generate test-sequences for security aspects of the Common Electronic Purse Specifications (CEPS) from the UMLsec models. This gives a systematic way of doing security testing. Since security vulnerabilities often arise from bugs in the implementation, having a systematic way to eliminate security-critical bugs is a worth-while goal.

References

[1] W.H. Allen, Chin Dou, and G.A. Marin. A model-based approach to the security testing of network protocol implementations. In *31st IEEE Conference on Local Computer Networks*, pages 1008 – 1015. IEEE, 2006.

[2] J.P. Bowen, K. Bogdanov, J.A. Clark, M. Harman, R.M. Hierons, and P. Krause. Fortest: Formal methods and testing. In *COMPSAC*, pages 91–104. IEEE, 2002.

[3] G. Bernot, M.-C. Gaudel, and B. Marre. A formal approach to software testing. In *AMAST*, pages 243–253. Springer, 1991.

[4] R. Chandramouli and M.R. Blackburn. Automated testing of security functions using a combined model and interface-driven approach. In *HICSS*, 2004.

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

[6] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Testing concurrent object-oriented systems with spec explorer. In *FM*, volume 3582 of *LNCS*, pages 542–547. Springer, 2005.

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

[8] R.M. Hierons and J. Derrick. Editorial: special issue on specification-based testing. *Softw. Test., Verif. Reliab.*, 10(4):201–202, 2000.

[9] J. Jürjens and P. Shabalin. Tools for secure systems development with UML. *Intern. Journal on Software Tools for Technology Transfer*, 9(5–6):527–544, October 2007. Invited submission to the special issue for FASE 2004/05.

[10] J. Jürjens, J. Schreck, and Yijun Yu. Automated analysis of permission-based security using UMLsec. In *FASE*, LNCS. Springer, 2008.

[11] J. Jürjens. *Secure Systems Development with UML*. Springer, 2004.

[12] J. Jürjens. Sound methods and effective tools for model-based security engineering with UML. In *ICSE*. IEEE, 2005.

[13] J. Jürjens and Yijun Yu. Tools for model-based security engineering: Models vs. code. In *22nd IEEE/ACM Int. Conf. Autom. Softw. Eng.* ACM, 2007.

[14] S. Khurshid and D. Marinov. Testera: Specification-based testing of Java programs using SAT. *Autom. Softw. Eng.*, 11(4):403–434, 2004.

[15] M. Oostdijk, V. Rusu, J. Tretmans, R.G. de Vries, and T.A.C. Willemse. Integrating verification, testing, and learning for cryptographic protocols. In Jim Davies and Jeremy Gibbons, editors, *IFM*, volume 4591 of *Lecture Notes in Computer Science*, pages 538–557. Springer, 2007.

[16] A. Petrenko and N. Yevtushenko. Test suite generation from a fsm with a given type of implementation errors. In *IFIP TC6/WG6.1 Twelth International Symposium on Protocol Specification, Testing and Verification (PSTV)*, IFIP Transactions, pages 229–243. North-Holland, 1992.

[17] D. Rosenzweig, D. Runje, and W. Schulte. Model-based testing of cryptographic protocols. In *TGC*, volume 3705 of *LNCS*, pages 33–60. Springer, 2005.

[18] Security verification tool, 2001-08. http://computing-research.open.ac.uk/jj/umlsectool.

[19] G. Wimmel and J. Jürjens. Specification-based test generation for security-critical systems using mutations. In *International Conference on Formal Engineering Methods (ICFEM)*, volume 2495 of *LNCS*, pages 471–482. Springer, 2002.