7

MODELLING AUDIT SECURITY FOR SMART-CARD PAYMENT SCHEMES WITH UML-SEC

Jan Jürjens*
Computing Laboratory
University of Oxford
GB
http://www.jurjens.de/jan
jan@comlab.ox.ac.uk

Abstract To overcome the difficulties of correct secure systems design, we propose formal modelling using the object-oriented modelling language UML. Specifically, we consider the problem of accountability through auditing.

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, indicate possible vulnerabilities and present concrete security advice on that system.

1. INTRODUCTION

Designing secure systems correctly is difficult. Many flaws have been found in proposed security-critical systems and protocols, sometimes years after their publication (e.g. [Low96]). This motivates using formal concepts and tools developed for systems design to ensure fulfillment of security requirements.

In this work we concentrate on *accountability* and the enforcement of *audit policy,* which provides the requirements for record keeping.

The Unified Modeling Language (UML) [RJB99] is an industry standard language for specifying software systems. Following [Jür01c], we use a simplified formal core of UML (for which [Jü01c] gave an extension with security

*Supported by the Studienstiftung des deutschen Volkes and the Computing Laboratory.

primitives called UMLsec) extended to model and investigate a security-critical part of the Common Electronic Purse Specifications (CEPS) [CEP00]. CEPS is a candidate for a globally interoperable electronic purse standard and is supported by organisations (including Visa International) representing 90 percent of the world's electronic purse cards, making its security an important goal.

A more general aim of this line of research started in [Jür01c, Jür01d] is to use UML to encapsulate knowledge on prudent security engineering and thereby make it available to developers not specialized in security.

In the following subsection we present some background information and refer to related work. In Section 3, we give an overview over the Common Electronic Purse Specifications, specify the part under consideration, explain the security threat model and give results. We end with a conclusion and indicate further planned work.

1.1. SECURITY-ASSURANCE USING FORMAL MODELLING

There has been extensive research in using formal models to verify secure systems. A few examples are [BAN89, Low96, Pau98, Jür00, AJ01, Jür01a, WW01], for an overview wrt. security protocols cf. [GSG99, RSG+01]. However, auditing does not seem to have been considered extensively.

An overview on payment systems is given in [AJSW00]. Smart card protocols have been investigated using formal logic in [ABKL93]. [BCG+00] considers secure information flow between applets in a multi-application smart-card. A different part of the CEPS is investigated in [JW01] using the CASE-tool AUTO FOCUS.

While many case-studies consider security protocols from the academic literature (usually presented in a much more tractable form), a notable example of a verification of a smart-card payment system used in practice can be found in [And99]. Also, [SCW00] gives a detailed, formal proof of a Smartcard product for electronic commerce.

Object-oriented systems offer a very suitable framework for considering security due to their encapsulation and modularisation principles [Eck95, BdVFS98, Sam00]. In [OvS94] the authors formulate a taxonomy for security in object-oiented databases. An object-oriented data flow model for smart card security is given in [GHdJF96].

2. MODELLING OBJECT-ORIENTED SECURITY

We use a simplified fragment of the visual modeling language UML (the industry-standard in object-oriented modelling), following [Jür01c].

UML consists of several diagram types describing different views on a system. Here we concentrate on using the UML notation to specify security requirements on auditing mechanisms of a system.

We use the following two kinds of diagrams:

Class diagrams define the static structure of the system: classes with attributes and operations/signals and relationships between classes. We use them to specify how the objects may communicate.

Statechart diagrams give the dynamic behaviour of an individual object: input events may cause state in change or (output) actions.

Below we will define the (simplified) abstract syntax for these two kinds of diagrams (on which the formal reasoning relies). Later we will also use the usual diagrammatic notation for readability.

We define the data type **Exp** of cryptographic messages that can be exchanged between objects. We assume a set D of basic data values. The set **Exp** contains the expressions defined inductively by the grammar

E ::=	expression
d	data value ($d \in D$)
K	key ($K \in$ **Keys**)
x	variable ($x \in$ **Var**)
$(E_1, \ldots, E_n)$	concatenation
$\mathsf{Enc}(K, E)$	encryption ($K \in$ **Keys** $\cup$ **Var**)
$\mathsf{Dec}(K, E)$	decryption ($K \in$ **Keys** $\cup$ **Var**)
$\mathsf{Mac}(K, E)$	MAC ($K \in$ **Keys** $\cup$ **Var**)
$\mathsf{Ver}(K, E)$	verify MAC ($K \in$ **Keys** $\cup$ **Var**)

The part of the CEPS considered here uses symmetric encryption. As usual, we assume the equations $\mathsf{Dec}(K, \mathsf{Enc}(K, E)) = E$ and $\mathsf{Ver}(K, \mathsf{Mac}(K, E)) = E$ and assume that no equations except those following from these hold.

2.1. CLASS DIAGRAMS

We first give the definition for class models.

An *attribute specification* $A = (\mathsf{att\_name}, \mathsf{att\_type})$ is given by a name $\mathsf{att\_name}$ and a type $\mathsf{att\_tags}$.

An *operation specification* $O = (\mathsf{op\_name}, \mathsf{Arguments}\ \mathsf{op\_type})$ is given by a name $\mathsf{op\_name}$, a set of $\mathsf{Arguments}$, and the type $\mathsf{op\_type}$ of the return value. Note that the set of arguments may be empty, and that the return type may be the empty type Ø denoting absence of a return value. An *argument* $A = (\mathsf{arg\_name}, \mathsf{arg\_type})$ is given by its name $\mathsf{arg\_name}$ and its type $\mathsf{arg\_type}$.

A *signal specification* is just like an operation specification, except that there is no return type.

An *interface* I = (int_name, Operations, Signals) is given by a name ink_name and sets of operation names Operations and signal names Signals specifying the operations and signals that can be called resp. sent through it.

A *class model* C = (class_name, Stereotypes, AttSpecs, OpSpecs, SigSpecs, Interfaces) is given by a name class_name, a set of Stereotypes (for our present purposes, this may be empty or contain the stereotype «*log*»), a set of attribute specifications AttSpecs, a set of operation specifications OpSpecs, a set of signal specifications SigSpecs and a set of class interfaces Interfaces.

A *class diagram* D = (Cls, Dependencies) is then given by a set Cls of class models and a set of Dependencies. A *dependency* is a tuple (client, supplier, interface, stereotype) consisting of class names client and supplier (signifying that client depends on supplier), an interface name interface (giving the interface of the class supplier through which client accesses supplier; if the access is direct this field contains the client name) and a stereotype which for our present purposes will be «send». We require that the names of the class models are mutually distinct.

In the diagrammatic notation (cf. Figure 1), a class model is represented by a rectangle with three compartments giving its name, its attributes and its operations (since all values are of type **Exp,** the type information is omitted in the diagrams given in this paper for readability).

The concurrenctly executed objects communicate asynchronously by exchanging signals, possibly with arguments. Dependency arrows marked with «*send*» from a class C to a class C' indicate that (an object instance of) C may send a signal to (an object of) C'. If the arrow points to an interface of C' (represented by a circle attached to the class rectangle), C may only use the signal listed in the corresponding interface specification (the respective rectangle marked «*inter face*»). For example, in Figure 1 Card may send the signal CLog with arguments *dt, lda, m, nt, bal, s*2 to CardLog, and Issuer may send RespL with arguments *ceps, iss, lda, s*2 to LSAM (but not the other signals offered by the LSAM, since they are reserved for Card).

2.2. STATECHART DIAGRAMS

We fix a set Var of (typed) variables *x, y, z, . . .* used in statechart diagrams.

We define the notion of a statechart diagram for a given class model C: A *statechart diagram* S = (States, init_state, Transitions) is given by a set of States (that includes the initial state init_state) and a set of Transitions.

A *statechart transition* t = (source, event, guard, Actions, target) is given by a source state, an operation term op_term, a guard,a list of Actions and a target state. Here an *event* is the name of an operation or signal with a list of distinct variables as arguments that is assumed to be well-typed (e.g. op(*x, y, z*)). Let the set Assignments consist of all partial functions that assign

to each variable and each attribute of the class C a value of its type (partiality arises from the fact that variables may be undefined). A *guard* is a function g : Assigments $\rightarrow$ Bool evaluating each assignment to a boolean value. An *action* can be either to assign a value v to an attribute a (written $a := v$), to call an operation op resp. to send a signal sig with values $v_1, \ldots, v_n$ (written $\mathsf{op}(v_1, \ldots, v_n)$ resp. $\mathsf{sig}(v_1, \ldots, v_n)$), or to return values $v_1, \ldots, v_n$ as a response to an earlier call of the operation op (written $\mathsf{return}_{\mathsf{op}}(v_1, \ldots, v_n)$). In each case, the values can be constants, variables or attributes (and need to be well-typed). In the case of *output actions* (calling an operation or sending a signal) we include the types of the arguments (and possibly of the return value).

To formally reason about statecharts, [Jür01c] gives a formal behavioural semantics (which has to be omitted here).

In the diagrammatic notation (cf. e.g. Figure 2), the states in a statechart are represented by rectangles, where the initial state has an ingoing transition from the start marker (a full circle). As specified in the abstract syntax, the transitions between states can carry three kinds of information as labels:

Events are names of operations provided by the class together with argument variables (e.g. *RespI*(*ic, cep, ex, nt, s*1) in Figure 2). If another object sends a signal, the corresponding transition is triggered, and the variables are bound to the arguments given. If a variable has already been assigned a value at an earlier point in the execution of the state machine, the transition is only executed if the two values match (i. e. an implicit equality conditional is enforced).

Guards are conditionals written in square brackets (e.g. [Ver(K_I, *s*2) = (*bal, cep, , iss, nt, s*1) Λ . . .] in Figure 3). A transition can only be triggered if all labeling guards are fulfilled. Sometimes a guard involves a variable that has not been assigned a value before (e.g. as an argument of an input event). Since in our behavioural formal semantics we implicitly quantify over free variables, this means that the equation *assigns* the corresponding value to the free variable and to make this clear we write the equation then as ":=" (but formally there is no difference to the usual "="). An example is [*ml* := Mac(*r,* (*ic, cep, nt, lda, m, s*1))] in Figure 2. Note that this is different from an action that assigns a value to an attribute; the variables here are local to the statechart diagram and are merely syntactic means for describing the object behaviour.

Actions are names of operations provided by other classes, written with a preceding backslash and including arguments (e.g. *Init*(*dt, lda, m*) in Figure 2). If a transition is fired, all labeling actions are executed, which means that the objects supplying the operations are called with the respective arguments.

E.g., in Figure 4, the transition from Init to Load is fired when the signal Load is sent and certain validity conditions are fulfilled. Then in turn the signal RespL is sent.

2.3. MODELLING SYSTEMS

We model a system S by a class diagram D and a set of statechart diagrams S, one for each object. In general, we also use *deployment diagrams* e.g. to distinguish secure from insecure communication links [Jür01c]. We omit these here because all links between the participants in the CEPS load transaction considered below are insecure.

We briefly sketch how to formally interpret such system models (for more details cf. [Jür01c]). When interpreting a system model S, each operation, say op, communicating along an insecure dependency is replaced by an operation op_out (for actions) resp. op_in (for events). An adversary A is a state machine with actions op_in and events op_out (for each operation op in S communicating insecurely). We only consider adversaries that are computationally bounded in the sense that they can encrypt or decrypt messages only when in possession of the relevant key (for a formalisation of this concept cf. [Jür01b]).

Output values are buffered without preserving the order of messages (i. e. buffers are multi-sets). Values without specified transition in an object are ignored. In both these assumptions we follow the usual UML point of view.

Histories are sequences of states of all state machines corresponding to the objects, and buffer contents (where the state machines for the specified objects are derived from the statechart diagrams as defined in [Jür01c]).

Given a system model S and an adversary A, the *execution of S in presence of A* is given as the set of *possible histories.*

A history $\vec{h}$ is a possible history if

- in its first component $\vec{h}_0$, all states are initial states and the buffer is empty, and if
- for each $n \geq 0$ and each class model $C \in \mathsf{Cls} \cup \{A\}$ that changes state at time n, there is a transition $t_{C,n}$ from its state at n to its state at $n+1$ such that for given n the multiset of (input) events ε_n corresponding to the transitions $(t_{C,n} : C \in \mathsf{Cls}\}$ is contained in the buffer content B_n at n and $B_{n+1} = (B_n \setminus \varepsilon_n) \cup A_n$ (for the multiset A_n of (output) actions fired by the transitions $\{t_{C,n} : C \in \mathsf{Cls}\}$).

2.4. AUDITING

We incorporate auditing in our framework by specifying a subset $\mathsf{Audit} \subseteq \mathsf{Cls}$ of class models used to store the audit data.

For completeness we give the following general definition of secure auditing. Note that the definition only applies to the situation where all the objects in the system model are honest. Thus in the considerations on CEPS below we need more specific notions of secure auditing.

Definition 1 A system model *S* provides *secure auditing* if, in presence of any adversary, the corresponding attribute values of all audit objects coincide when all objects have reached a final state.

Note that here we do not consider the question whether an object may be kept from reaching its final state.

3. 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 (while credit card numbers are valid until the card is stopped, enabling 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. [CEP00].

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.

3.1. SPECIFICATION OF CEPS LOAD TRANSACTION

We give a specification of the CEPS load transaction (slightly simplified by leaving out security-irrelevant details, and also leaving out details needed for exception processing and declined loads). 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 unat-

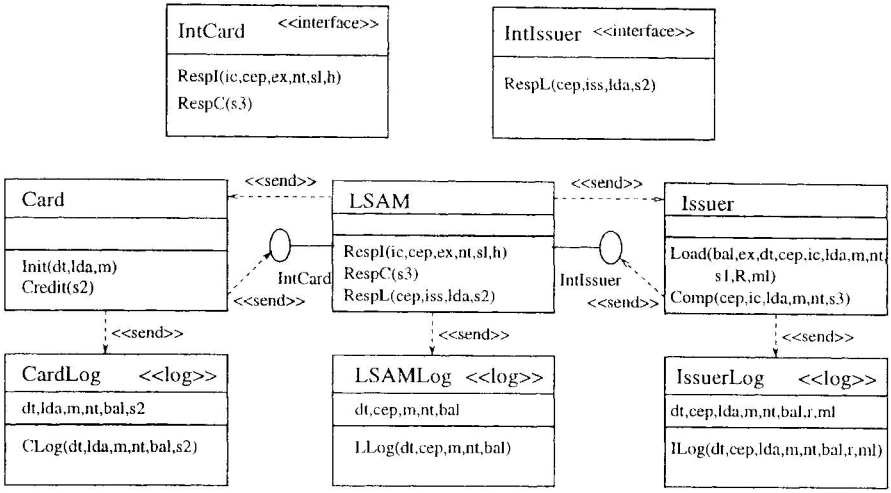

Figure 1 Class diagram for Load transaction

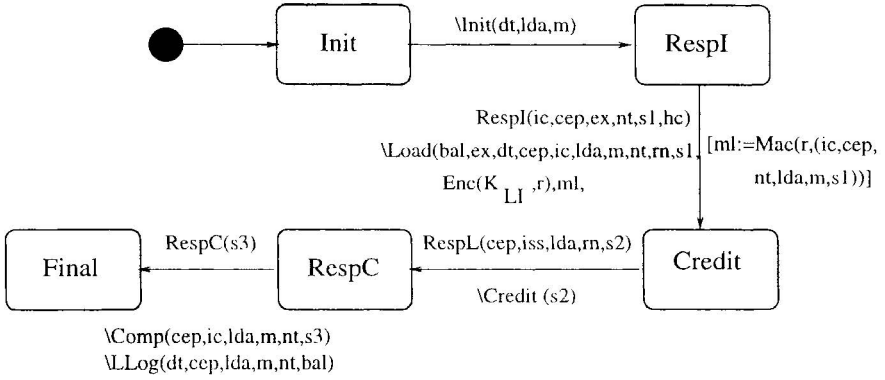

Figure 2 Statechart for LSAM

tended) loading machine and receives a corresponding credit on the card) since linked load (where funds are transfered e.g. from a bank account) offer fewer possibilities for fraud [CEP00, Funct. Req. p. 12]. We use class diagram and statechart diagrams introduced above.

First, we give the involved classes and their dependencies in the class diagram in Figure 1. 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 (marked with the stereotype $\langle\langle$ *log* $\rangle\rangle$).

We specify the behaviour of the classes Card, LSAM, and Issuer using UML statecharts in the remaining figures.

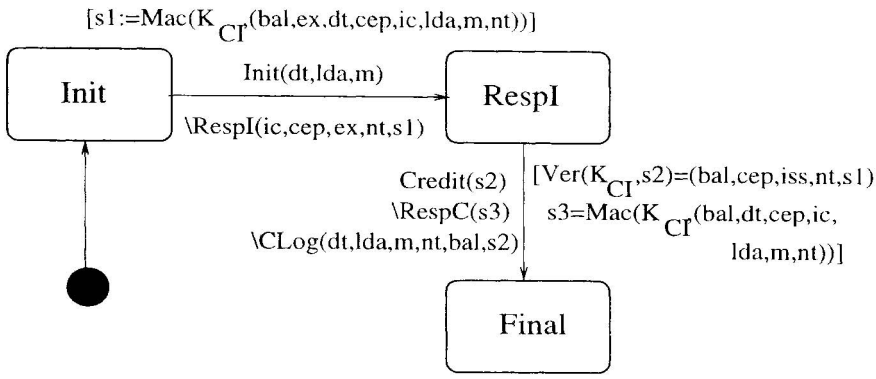

Figure 3 Statechart for card

The LSAM (Figure 2) initiates the transaction after the CEP card is inserted into the load device, by sending the "Init for load" message Init with arguments the transaction date and time *dt,* the load device identifier *lda* and the transaction amount *m* (which is the amount of cash paid into the load device by the card holder that is supposed to be loaded onto the card). Whenever the card (Figure 3) receives this message after being inserted into the load device, it sends back the "Init for load response" message RespI to the LSAM, with arguments the card issuer identifier *ic* (as stored on the card), the card identifier *cep,* the balance (prior to load) *bal,* the card expiration date *ex,* the card's transaction number *nt* unique to the transaction, and the card MAC $s1$. $s1$ consists of the values *ex, bal, dt, cep, ic, lda, m* and *nt,* all of which are signed with the key K_C^{-1} shared between a particular card and the corresponding card issuer. The LSAM then sends to the issuer the "load request" message Load with arguments *bal, ex, dt, cep, ic, lda, m, nt, rn,* $s1$, $\mathsf{Enc}(K_{LI}, r)$, and *ml. rn* is the reference number assigned by the LSAM to the transaction. $\mathsf{Enc}(K_{LI}, r)$ is the encryption of a random number *r* generated by LSAM under a key K_{LI} shared between the LSAM and the issuer. *ml* is the MAC of the following data using the fresh key *r* generated by the LSAM: *ic, cep, nt, lda, m,* and $s1$. The issuer (Figure 4) checks if *ic* is a valid issuer identifier, *cep* a valid card identifier and the expiration date *ex* has not been exceeded. The issuer verifies if $s1$ is a valid MAC generated from the values *ex, bal, dt, cep, ic, lda, m* and *nt* with the key K_{CI} (i. e. if $\mathsf{Ver}(K_{CI}, s1) = (bal, ex, dt, cep, ic, lda, m, nt)$). The issuer retrieves *r* from $\mathsf{Enc}(K_{LI}, r)$ (using the key K_{LI} shared between the LSAM and the issuer, i.e. $r := \mathsf{Dec}(K_{LI}, R)$) and checks if *ml* is a valid MAC of the values *ic, cep, nt, lda, m,* and $s1$ using the key *r,* i. e. if $\mathsf{Ver}(ml, r) = (ic, cep, nt, lda, m, s1, hc)$. Lastly, the issuer checks that the key K_{LI} is actually shared with the LSAM named *lda* (we write this as $\mathsf{Shared}(K_{LI}) = lda$ assuming a function Shared

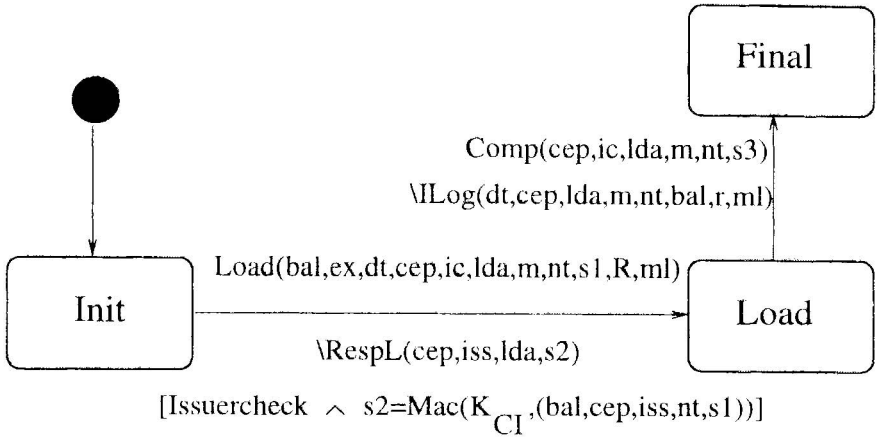

Figure 4 Statechart for Issuer

which assigns LSAMs to keys). If all these checks succeed (which in Figure 4 are abbreviated by the conditional Issuercheck), the issuer sends the "respond to load" message RespL with arguments *cep, ic, lda, rn,* and $s2$ to the LSAM. $s2$ consists of the following values, signed with the key K_{CI} : *bal, cep, iss, nt,* and $s1$.

Next, the LSAM sends the "credit for load" message Credit with argument $s2$ to the card. Finally, the card (on successful verification of $s2$) answers by sending the "response to credit for load" message RespC with argument $s3$ back to the LSAM. $s3$ consists of the following values, signed with the key K_{CI} : *bal, dt, cep, ic, nt, lda, m,* and *nt.* The card also sends the logging message CLog to the object CardLog, with arguments *dt, lda, m, nt, bal,* and $s2$. Finally, the LSAM sends to the issuer the "transaction completion message" Comp with arguments *cep, ic, lda, m,* and *nt.* Also, the LSAM sends the logging message LLog to the object LSAMLog, with arguments *dt, cep, iss, m, nt,* and *bal.* On receipt of the messsage Comp from the LSAM (and provided the contained values match the corresponding values communicated earlier), the issuer sends the logging message ILog to the object IssuerLog, with arguments *dt, cep, lda, m, nt,* and *bal.*

The logging objects simply take the arguments of their operations and update their attributes accordingly.

3.2. SECURITY THREAT MODEL

We consider the threat scenario for the load transaction and derive audit security conditions. The general assumption is that the card, the LSAM and the security module of the card issuer are tamper-resistant (in particular that the

contained secret keys cannot be retrieved). The protocol can be attacked e.g. by inserting adapters or relays between the LSAM and the card loading device or by intercepting the communication with the card issuer.

We concentrate on the load acquirer as a possible attacker of the transaction. The cardholder could try to attack the protocol by interrupting it e.g. by pulling out the card (thus one needs to make sure that money is not returned to the cardholder after the card has been loaded) or could try to duplicate the loaded money by loading it on two cards simultaneously using an adapter (at an unattended load device). We do not consider these kinds of attacks here. Also, the card issuer is not so interesting as an attacker since she controls the settlement scheme that is performed after the transactions, so the cardholder and the load acquirer have to trust her to some degree anyway (and my disputes would have to be settled in court).

Given the participants of the protocol, the load acquirer can attack either the cardholder, or another load acquirer, or the card issuer, with the goal either to keep the amount paid by the cardholder (and not have to pass it on to the card issuer), or to credit a card owned by the load acquirer himself without having to pay any money to the card issuer.

We consider attacks against the cardholder. Smart cards can not communicate directly with the cardholder. Thus there is the usual threat that a load device (possibly belonging to a corrupt load acquirer) is manipulated so that the transaction is performed as if the cardholder had only paid part of the amount that was actually paid, or so that the transaction is not performed at all. Then the load acquirer would not have to pay the amount to the card issuer. However, we assume that the cardholder can verify after the transaction if the correct amount has been loaded (possibly using a portable card reader), and that a complaint settlement scheme settles any disputes arising from such attacks. The correct functioning of the settlement scheme relies on the fact that the cardholder should only be lead to believe (e.g. when checking the card with a portable card reader) that a certain amount has been correctly loaded if he 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. Thus we have to check the following audit security condition on the attributes of CardLog after Card has reached its final state:

Correct amount: $s2$ and $s1$ verify correctly (say $\mathsf{Ver}(K_{CI}, \mathsf{CardLog}.\ s2) = (bal', cep', iss', nt', s1')$ and $\mathsf{Ver}(K_{CI}, s1') = (bal'', ex'', dt'', cep'', ic'', lda'', m'', nt'')$ for some values $bal', bal'', \ldots$), and additionally we have $\mathsf{CardLog}.m = m''$ (i. e. the correct amount is logged).

A load acquirer could also try to attack the protocol in order to masquerade as another load acquirer for the purpose of the settlement process, in order not

to pay the amount paid in by the cardholder to the card issuer. To prevent this, we need to ensure the following audit security condition:

No masquerade: We have Shared(K_{LI} , IssuerLog.*lda*).

ml is supposed to provide a guarantee that the load acquirer owes the transaction amount to the card issuer [CEP00, Funct.spec. ,6.6.1.6]. To be able to make use of this guarantee, the card issuer needs to be able to show that her possession of the guarantee implies that the load acquirer owes her the amount (and that the card issuer could not just produce *ml* himself). Thus we have the audit *functionality* condition

Acquirer guarantee functionality: If

$$\text{IssuerLog}.ml = \text{Mac}(\text{IssuerLog}.r,(ic', cep', nt', lda', m', s1', hl'))$$

then the LSAM *lda'* has received *m'*.

Also, we would like to ensure that this guarantee is always given, i. e. that the following audit security condition (the converse of the above functionality condition) is fulfilled:

Acquirer guarantee security: If the state machines of card and card issuer have reached the final state and CardLog.$m = m'$ then

$$\text{IssuerLog}.ml = \mathbf{Mac}(\text{IssurerLog}.r,(ic', cep', nt', lda', m' s1', hl'))$$

Note that the precondition that card and card issuer have reached their final states is necessary. In particular, if the load device simply takes the inserted cash without taking any further action, the cardholder has no proof of this (but this is the usual risk taken at automatic purchase machines), and if the LDA does not complete its last action, exception processing on the side of the card issuer would have to be followed (not considered here).

3.3. RESULTS

Theorem 1 Acquirer guarantee functionality *is not provided in the proposed scheme.*

The reason for this is that the security of the data elements in *ml* are only protected by the random value *r,* which in turn is communicated encryted under the secret key K_{LI} shared between load acquirer and card issuer. This means that the card issuer would in principle be capable of manufacturing *ml* and *r* herself. Therefore possession of *ml* does not suffice for the issuer to be able to prove that the load acquirer manufactured *ml.*

This is not a serious threat since one would expect that in practical situations any dispute arising from this could be resolved in a settlement process. However,

the CEPS explicitely postulate this requirement. This should either be clarified, or the data element *ml* be changed to involve a signature with a private key of the load acquirer.

Theorem 2 *The audit security conditions* **Correct amount, No Masquerade,** *and* **Acquirer guarantee security** *are fulfilled.*

The formal proof of this theorem has to be omitted for space limitations and will be included in the long version of the paper. The proof proceeds inductively along the lines of ideas in [Pau98] and uses results in [Jür01b, Jür01a]. Here we can only give some informal remarks:

Correct amount: Essentially, one has to show that the key K_{CI} shared between the card and the card issuer established end-to-end security between card and issuer.

No masquerade: This amounts to showing that the load device identifier, as stored in the issuer log, corrsponds to the load device with which the issuer shares the key K_{LI} .

Acquirer guarantee security: Here one has to show that the integrity of the information passed between card and card issuer is preserved.

4. CONCLUSION AND FUTURE WORK

We investigated the security of the currently developed Common Electronic Purse Specifications (CEPS) using the object-oriented modelling language UML. Benefits of our approach include the possibility to investigate security in the context of general system development. Since security violations often occur at the boundaries between security mechanisms (such as protocols) and the general system [And94], this is very helpful. We choose UML among the various object-oriented modelling languages since it is the current de-facto industry standard and thus many developers will be able to take advantage of an extension of UML by security primitives.

Apart from these methodological benefits, this work delivers concrete results on the security of the payment systems that are to be developed and fielded according to the CEPS. Our investigation exhibited a weakness arising from the fact that the card issuer does not obtain a sound proof of transaction from the load acquirer. As usual, the positive results given here should not be interpreted as proving the CEPS secure (as well-known, such a proof is impossible).

Due to space constraints we could only consider one part of the CEP specifications, the other parts are left for further work. Since UML offers a variety of modelling mechanisms with varying degrees of abstraction, considering a large part of a system seems relatively feasible. It may also be interesting to

consider reevaluation of security after system changes. Also, we will extend this approach beyond reasoning about accountability.

Acknowledgments

This idea to use UML to specify security properties arose when doing security consulting for a project during a research visit with M. Abadi at Bell Labs (Lucent Tech.), Palo Alto, whose hospitality is gratefully acknowledged. Comments or advice from participants of the summer school "Foundations of Security Analysis and Design 2000" and the Dagstuhl seminar "Security through Analysis and Verification" (especially D. Gollmann) are gratefully acknowledged, as well as useful comments from G. Wimmel and the anonymous referees on a draft.

References

[ABKL93] M. Abadi, M. Burrows, C. Kaufman, and B. Lampson. Authentication and delegation with smart-cards. *Science of Computer Programming,* 21(2):93 – 113, 1993.

[AJ01] M. Abadi and Jan Jürjens. Formal eavesdropping and its computational interpretation, 2001. Submitted.

[AJSW00] N. Asokan, P. Janson, M. Steiner, and M. Waidner. The state of the art in electronic payment systems. *Advances in Computers,* 53, 2000.

[And94] R. Anderson. Why cryptosystems fail. *Communications of the ACM,* 37(11):32–40, November 1994.

[And99] R. Anderson. The formal verification of a payment system. In Mike Hinchey and Jonathan Bowen, editors, *Industrial-Strength Formal Methods in Practice,* pages 43–52. Springer, 1999.

[BAN89] M. Burrows, M. Abadi, and R. Needham. A logic of authentication . *Proc. Royal Society of London A,* 426:233–271, 1989.

[BCG + 00] P. Bieber, J. Cazin, P. Girard, J.-L. Lanet, V Wiels, and G. Zanon. Checking secure interactions of smart card applets. In *ESORICS,* 2000.

[BdVFS98] E. Bertino, S. De Capitani di Vimercati, E. Ferrari, and P. Samarati. Exception-based information flow control in object-oriented systems. *ACM Transactions on Information and System Security,* 1 (1): 26–65, 1998.

[CEP00] CEPSCO. Common Electronic Purse Specifications, 2000. Business Requirements vers. 7.0, Functional Requirements vers. 6.3, Technical Specification vers. 2.2, available from http://www.cepsco.com.

[CFMS94] S. Castano, M. Fugini, G. Martella, and P. Samarati. *Database Security.* Addison Wesley, 1994.

[Eck95] C. Eckert. Matching security policies to application needs. In J. H.P. Eloff and S.H. von Solms, editors, *IFIP TC11 11th International Conference on Information Security,* pages 237–254. Chapman & Hall, 1995.

[GHdJF96] H. Glaser, P. Hartel, and E. de Jong Frz. Structuring and visualising an IC - card security standard. In *in [HPQ96],* pages 89–110, 1996.

[GSG99] Stefanos Gritzalis, Diomidis Spinellis, and Panagiotis Georgiadis. Security protocols over open networks and distributed systems: Formal methods for their analysis, design, and verification, *Computer Communications Journal,* 22(8):695–707, 1999.

[HPQ96] P. H. Hartel, P. Paradinas, and J. - J. Quisquater, editors. *2nd Smart card research and advanced application conference (CARDIS).* Stichting Mathematisch Centrum, Amsterdam, 1996.

[Jür00] Jan Jürjens. Secure information flow for concurrent processes. In *CONCUR 2000 (11th International Conference on Concurrency Theory),* volume 1847 of *LNCS,* pages 395–409, Pennsylvania, 2000. Springer.

[Jür01a] Jan Jürjens. Composability of secrecy. In *International Workshop on Mathematical Methods, Models and Architectures for Computer Network Security (MMM-ACNS 2001),* LNCS, St. Petersburg, 21-23 May 2001. Springer.

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

[Jür01c] Jan Jürjens. Towards development of Secure systems using UMLsec. In *Fundamental Approaches to Software Engineering (FASE/ETAPS, International Conference),* LNCS. Springer, 2001.

[Jür01d] Jan Jürjens. Transformations for introducing patterns – a secure systems case study. In *WTUML: Workshop on Transformations in UML (ETAPS 2001 Satellite Event),* Genova, 7 April 2001.

[JW01] Jan Jürjens and Guido Wimmel. Security modelling for electronic commerce: The Common Electronic Purse Specifications. Submitted, 2001.

[Low96] G. Lowe. Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR. *Software Concepts and Tools,* 17:93–102, 1996.

[OvS94] M. Olivier and S. von Solms. A taxonomy for secure object-oriented databases. *ACM Transactions on Database Systems,* 19(1):3–46, 1994.

[Pau98] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security,* 6(1–2):85–128, 1998.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[RSG+01] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *Modelling and Analysis of Security Protocols.* Addison Wesley, 2001. (to be published).

[Sam00] P. Samarati. Access control: Policies, models, architectures, and mechanisms. Lecture Notes, 2000.

[SCW00] S. Stepney, D. Cooper, and J. Woodcock. *An Electronic Purse: Specification, Refinement, and Proof.* Oxford University Computing Laboratory, 2000. Technical Monograph PRG- 126.

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