# Developing provable secure m-commerce applications

**Holger Grandy, Dominik Haneberg, Wolfgang Reif, Kurt Stenzel**

**Angaben zur Veröffentlichung / Publication details:**

Grandy, Holger, Dominik Haneberg, Wolfgang Reif, and Kurt Stenzel. 2006. "Developing provable secure m-commerce applications." In *Emerging Trends in Information and Communication Security: International Conference, ETRICS 2006, Freiburg, Germany, June 6-9, 2006; proceedings*, edited by Günter Müller, 115–29. Berlin: Springer. https://doi.org/10.1007/11766155_9.



# Developing Provable Secure M-Commerce Applications

Holger Grandy, Dominik Haneberg, Wolfgang Reif, and Kurt Stenzel

Lehrstuhl für Softwaretechnik und Programmiersprachen
Institut für Informatik, Universität Augsburg
86135 Augsburg Germany
{grandy, haneberg, reif, stenzel}@informatik.uni-augsburg.de

**Abstract.** We present a modeling framework and a verification technique for m-commerce applications[1]. Our approach supports the development of secure communication protocols for such applications as well as the refinement of the abstract protocol descriptions into executable Java code without any gap. The technique is explained using an interesting m-commerce application, an electronic ticketing system for cinema tickets. The verification has been done with KIV [BRS+00].

## 1 Introduction

Many m-commerce applications (e.g. electronic ticketing) transmit and store confidential user data and digital data that represents business goods. Data that is transmitted or stored is subject to modification or duplication. This poses a threat to the applications because it may lead to fraud, therefore such problems must be ruled out in order to offer a secure application. One major problem are design errors in the security protocols, another are programming errors in the protocol implementation.

Many cryptographic protocols initially had weaknesses or serious errors (see, e.g. [AN95] [WS96] [BGW01]). Different approaches have been proposed to verify the protocols, e.g. model checking based approaches [Low96] [BMV03], specialized logics of belief [BAN89], interactive theorem proving [Pau98] [HRS02] and specialized protocol analyzers [Mea96]. To cope with the problem of erroneous implementations the generic techniques for program verification can be used, but must be adapted.

This paper presents an interesting m-commerce application called Cindy for buying cinema tickets using mobile phones. It is modeled and formally analyzed using Abstract State Machines (ASM). We deal with the formal verification of the security protocols and verify a refinement step to a Java implementation. While not highly security critical, the Cindy application is simple to understand and serves to illustrate the relevant issues.

The usual verification of cryptographic protocols is focused on proving standard properties like secrecy or authenticity. We, however, focus on application

[1] This work is supported by the Deutsche Forschungsgemeinschaft.

specific properties. We also do not use the common Dolev-Yao attacker model [DY83]. Instead the abilities of our attacker are tailored to realistically represent the application scenario. Additionally, the properties that must be proved are more complicated than the standard properties. Secrecy or authenticity of data is just the basis for proving the properties we are interested in, e.g. 'only tickets issued by the cinema permit entry to it'. An especially important difference to the usual protocol analysis is that we also deal with availability properties that are of interest to the customer and prove such properties without using a temporal logic.

This paper is structured as follows: Section 2 introduces the example application Cindy. In section 3 the formal application model is described, followed by the refinement to Java (section 4). Section 5 presents the security properties and their proofs, and section 6 contains a conclusion.

## 2 The Cindy Application

Cindy introduces electronic tickets for a cinema. A ticket is stored on the visitor's mobile phone and displayed for inspection. The service works as follows (see Fig. 1): The user orders a ticket in advance, either by Internet or with an application running on a mobile phone. Payment is handled either by credit card number or by the usual phone bill. Then the ticket is sent to the phone as a MMS (Multimedia Messaging Service) message. It contains the ticket data and

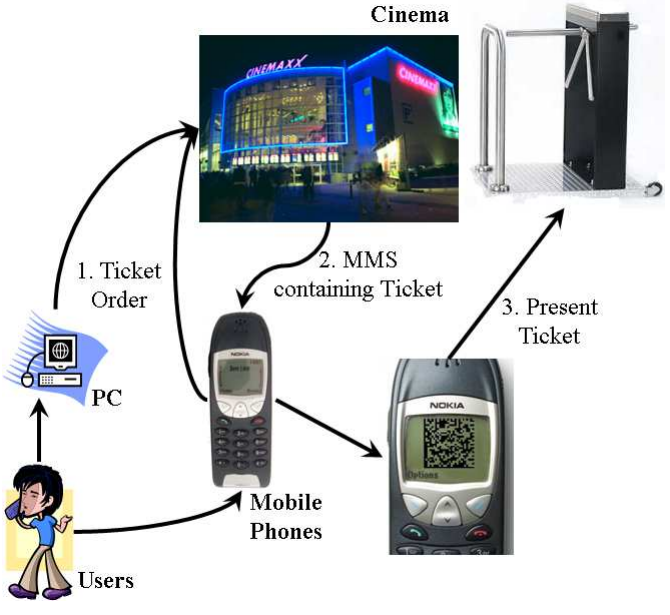


**Fig. 1.** The Cindy Application

an image that is unique for this ticket. On entry this image is scanned from the phone's display. This can be done automatically using a special entrance with a turnstile.[2] The application exists in the Netherlands for seven cinemas [Bee]; the authors are not aware of any other deployments in Europe.

Electronic tickets are attractive for customers because the typical moviegoer has a mobile phone, and can buy the ticket everywhere, any time without a PC and without waiting in a queue. The cinema, on the other hand, can reduce the ticket sales staff, and save on specialized paper and ink for printing tickets.

One important question for the cinema is, of course, how to avoid fraud. The idea is simple: Every ticket contains a nonce, a unique random number that is too long to guess. Therefore, it is virtually impossible to 'forge' a ticket. This nonce is displayed as a data matrix code, a two-dimensional matrix bar code that can be scanned from a handheld's display. The scanner must be connected to a server that keeps track of issued and presented tickets. It is possible to copy a ticket: A user can buy one ticket and send it to his friends. However, this is easy to detect. On entry, the server must check if this number was already presented. If this was the case the second visitor is not admitted.

The more interesting question in the context of this paper is: What can be guaranteed for the user of the service? The user must register in advance and provide payment information. Then tickets can be ordered either with a PC by Internet (with a password and using a standard SSL (Secure Sockets Layer) connection) or with a mobile phone. The ticket can be sent to an arbitrary phone number, for example as a gift. Furthermore, it is possible to pass on a ticket from one phone to another (e.g. one person buys the tickets for a group of people). We will assume that the cinema is honest, but the user should be secure from third-party attacks. We want to guarantee (and formally prove) the following properties:

1. If the user orders a ticket he will eventually receive it.
2. If the user is charged for a ticket he ordered it.
3. If the user has a ticket and presents it at the turnstile, then he will be admitted.

These properties are quite natural, and describe what one would expect. However, they do not hold in this general form. They all have preconditions, for example concerning the user's behavior: If a user sends the ticket to another phone, another person has access to the ticket and could pass the turnstile *before* the user, who then will be rejected.

(Maybe not) surprisingly, these properties are usually not considered in the world of formal (cryptographic) protocol verification. First, they do not deal with confidentiality, but rather with availability, or with things that can or will happen. This, however, is usually difficult to express formally unless temporal logic is used. Second, the usual attacker in formal protocol verification is a Dolev-Yao attacker that may analyze, modify, or suppress any protocol message between

[2] However, this is not unique to electronic tickets, but could be done with paper tickets as well.

any participants in real time. But if any message concerning our user is suppressed he will not be admitted to the cinema. Due to that we model a limited attacker, that cannot manipulate all communication channels.

Going to the movies requires three 'messages': 1. the ticket order (by PC or by SMS), 2. delivery of the ticket (by MMS), 3. ticket inspection at the turnstile (by visual scan). In principle, all three messages can be suppressed by an attacker. However, suppressing, manipulating, or faking the originator of a GSM (Global System for Mobile Communications) message requires either insider access or sophisticated equipment, and is out of proportion for this application. It is also very difficult to eavesdrop on a GSM connection. Suppression of the ticket presentation at the turnstile requires physical force, and can also be discounted for protocol verification purposes. The PC/Internet connection, on the other hand, can be suppressed or manipulated, but can be considered confidential if we assume that an underlying SSL protocol is secure enough. To summarize, even though the application does not actually use cryptography, it is an interesting m-commerce application with several features that are usually not considered in formal protocol verification.

## 3 The Abstract Model of Cindy

The formal model of the Cindy application uses a combination of Abstract State Machines (ASM) and algebraic specifications. The algebraic part contains the necessary information on the participants of the application (the so-called agents), the communication between the agents, the abilities of the attacker and so on. The dynamic aspects of the application, i.e. the possible actions of the agents, are described by the rules of the ASM. ASMs are an abstract specification formalism [BS03] [Gur95] that has a programming language-like syntax and an exact semantics. ASMs can be used for a variety of specification tasks, from programming language semantics to distributed systems. The protocol ASM describes the possible traces of the application. In this context the term 'trace' designates a possible run of the application. A trace is a list of events that may happen within the application, e.g. an activity by the attacker or a protocol step consisting of receiving some input and sending an output.

### 3.1 The Formal Application Model

The formal application model consists of two parts. The first part is an algebraic specification and the second part is the protocol ASM. The algebraic specification defines the used data types (e.g. the messages that are exchanged between the agents are represented by the freely generated data type **Document**, cf. [HGRS05]) and describes the communication structure of the application and how the attacker can influence the communication. The protocol ASM is a set of rules each describing a step possible for one agent type. These include, of course, the actual protocol steps by the different systems appearing in the application, but also steps that represent actions of the attacker or steps of the infrastructure

representing the environment in which the application is operating, e.g. changes to the established connections. The agents consist of the users, their mobile phones, the PCs, the cinema, and the attackers. In the Cindy application we must consider attacks that involve several people, e.g. one person buys a ticket and all his buddies are admitted to the cinema as well.

### 3.2 Communication in the Cindy Application

The Cindy application uses different communication techniques each with very specific features. The most important ones are:

- The usage of MMS to send tickets. The GSM network guarantees[3] that a transmitted MMS cannot be manipulated and the attackers cannot eavesdrop into the communication. Also important is that the receiver of a MMS is determined uniquely (by the phone number to which the MMS is sent) and that the sender of a MMS is known to the receiver (because the phone number of the sender is contained in the MMS).
- A SSL connection between the Internet PC and the cinema. The attackers cannot eavesdrop on or manipulate the data transmitted using the SSL connection. Additionally, the user can identify the cinema as his communication partner (by checking the SSL certificate) but the cinema cannot directly identify the Internet PC.
- The visual scan of the ticket at the turnstile. We assume that the attacker can eavesdrop on the presentation of the ticket (e.g. taking a photo of the data matrix code on the customer's mobile phone display) but he cannot manipulate the presentation of the ticket. (Something that is shown on a display should never be considered secret.)

All these communication techniques are different from the Internet-like communication assumed in the formal analysis of cryptographic protocols which uses an attacker model based on the Dolev-Yao attacker [DY83]. The Dolev-Yao attacker has access to all communication, and the infrastructure does not guarantee the identity of sender and receiver. This is inadequate for the Cindy application.

All the specific features of the communication must be specified in the formal model, because they are essential for the security of the application (otherwise the security goals would not be provable). For example, the algebraic specification of the infrastructure ensures that the sender of a MMS cannot be forged by the attackers.

### 3.3 State of an Agent

The state of an agent is defined by the values that are currently contained in the fields of the agent (this is an object-oriented view of the agents). Therefore

[3] Although this is not entirely true, for the scope of this application an attack against the infrastructure seems unlikely.

the content of the fields of all agents must be stored. This is done using dynamic functions, as usual in ASM.

The state of the attackers and the users only consists of sets of documents (containing data) that they may use to generate new documents. Each user knows his personal login secret for Internet orders. The attackers initially have an empty knowledge. Since in the worst case all attackers cooperate they share a common knowledge. The state of a mobile phone consists of three lists of documents, one for the tickets stored on the cell-phone (*tickets*), one for the tickets that were passed on to another mobile phone (*passedOn*) and one for the bookings that were done by the phone (*booked*). The cinema state contains one list of documents containing the issued tickets (*issued*), all the nonces that were presented at the turnstile (*presented*), the ones that were rejected (*rejected*) and those that were accepted (*accepted*). In order to express a specific security property the information which visitor was admitted for a given ticket is stored, too (*accepted-with-presenter*). A PC stores all the ticket orders it sends to the cinema in the list *booked*.

### 3.4 The Protocol ASM

The protocol ASM is a nondeterministic machine built in a modular way. The ASM on top-level only chooses nondeterministically the agent that should perform its next protocol step. The nondeterministic selections ensure that the ASM can construct all possible traces of the application. On the agent level of the ASM there is a rule for each type of agent that exists in the application. After the agent was chosen the protocol ASM branches into the ASM rule that describes its behavior. All these rules consist of a case statement that tests the applicability conditions of all protocol steps specified for this type of agent until the first condition is found that holds in the current state. Then the protocol step that belongs to this condition is executed.

```
   ...
1) if   is-comdoc(indoc) ∧ indoc.inst = loadTicket
2)    ∧ inport = 2 ∧ is-doclist(indoc.data)
3)    ∧ # indoc.data.list = 2 ∧ is-intdoc(get-part(indoc.data, 1))
4)    ∧ is-noncedoc(get-part(indoc.data, 2))
5)    ∧ #(tickets)(agent) < MAX-NO-TICKETS
6) then tickets(agent) := tickets(agent) + inmsg '
   ...
```

**Fig. 2.** ASM rule for receiving tickets

For example, the protocol step that is performed by a mobile phone after it received a MMS containing a new ticket is on described in figure 2. The condition part (lines 1 to 5) states that this step will be performed only if the message that is processed is a command to load a new ticket (line 1 to 4), is-comdoc(*doc*) is a predicate that is true if *doc* has a certain form. Basically, the condition

means that the data part of the MMS is well-formed, i.e. it contains an encoded ticket in a format that is accepted by the mobile phone. Line 5 demands that the list of tickets stored in the mobile phone has not yet reached its maximal accepted length. The change of the internal state for this protocol step is limited to extracting the new ticket from the MMS and appending it to the list of already stored tickets. This is done in line 6. The new ticket is represented by the variable *inmsg* which contains the message that is currently processed by the agent.

## 4 Refinement

Refinement is a well-established method for proving concrete implementations correct with respect to an abstract specification [HHS86] [BDW99] [WD96] [dRE98] [DB01]. When the concrete implementation adheres to certain rules, all properties of the abstract specification (especially security properties in our case) are automatically satisfied by the concrete implementation.

Refinement is difficult because when writing the specification on the abstract level, one usually does not consider how things should be implemented later. For example, when specifying a list of tickets for the mobile phone, the first thought on the abstract level would be to use an algebraically specified list of arbitrary length. To permit a later refinement and an implementation, the writer of the abstract specification has to keep such things in mind. Additionally, the encoding of certain types is different on the abstract and on the concrete level.

### 4.1 Refining a Protocol ASM

After specifying the protocol on an abstract level and proving security we now verify that the real implementation running on a mobile phone is correct with respect to the abstract specification. For this purpose we developed a refinement method for ASM protocol specifications. In our approach the concrete implementation contains the Java source code for the real application. In the Cindy scenario this implementation is based on the Java Micro Edition [Sun].

The KIV system supports the verification of Java source code. It includes a calculus and semantics for sequential Java [Ste04] [Ste05]. The calculus has been proven correct regarding the semantics. Verification support has been tested and improved in many case studies.

Additionally, we use a verification kernel approach [GSR05]. Verification kernels allow to extract the security relevant part of the Java source code running on the mobile phone, thereby separating e.g. the GUI (Graphical User Interface) or the Communication subsystem without losing security properties.

The general idea of the refinement approach is to combine the Java calculus with the ASM methodology in KIV. The protocol ASM is refined to another ASM in which the Java source code running in the real application is embedded. Due to the modular specification on the abstract level it is possible to do a stepwise refinement and substitute the abstract protocol part of one agent after another by a Java implementation.

The Java calculus in KIV uses a store *st*, which contains all the information relevant for the behavior of a certain piece of Java source code. E.g. all the Java objects of the program with their actual field values are inside the store. Basically, the store can be seen as the heap together with the internal state of the Java Virtual Machine. This store *st* is now part of the state of the ASM on the concrete level.

The refinement is based on Downward Simulation, which has been adapted to ASM [Sch01] [Bör03] [Sch05]. We use the following notations:

- *st* is the Java store
- *as* is the state of the abstract ASM, *cs* is the state of the concrete ASM, both given by the state functions (the fields of the agents)
- $step_a$ is the relation of type $as \times as$ describing one step of the abstract ASM, and $step_c$ of type $(cs \times st) \times (cs \times st)$ the one for the concrete ASM
- $init_a$ is the initialization condition on *as*, $init_c$ the one for $cs \times st$
- $fin_a$ is the finalization condition (condition for termination of the ASM) on *as*, $fin_c$ the one for $cs \times st$
- *R* with type $as \times (cs \times st)$ is the retrieve relation between the states

Figure 3 shows the relation between the abstract protocol ASM and the concrete one in the Cindy scenario.

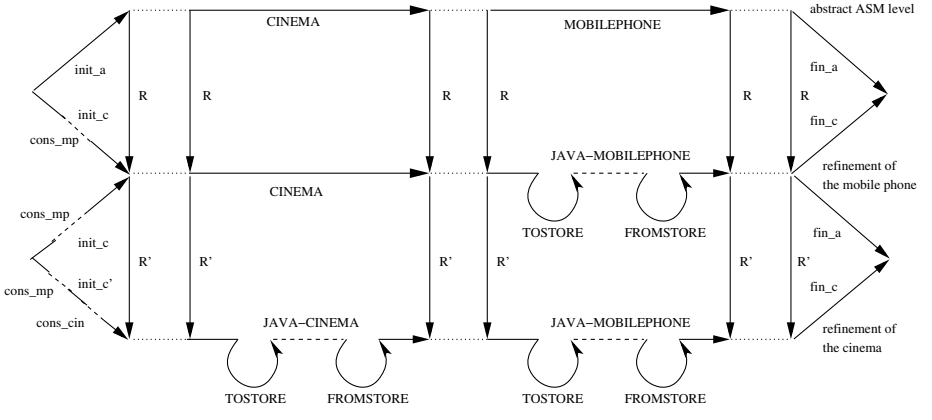


**Fig. 3.** Refinement Diagram

Figure 3 shows two subsequent refinements. All Java parts are shown as dashed lines. *CINEMA* and *MOBILEPHONE* are two of the possible abstract steps ($step_a$). *JAVA-MOBILEPHONE* and *JAVA-CINEMA* are possible concrete steps ($step_c$). The upper layer describes the abstract protocol specification. The middle layer represents the first refinement, in which the abstract mobile phone specification is replaced by a Java implementation. The third layer additionally contains a refinement of the cinema.

For this paper we refine the mobile phone which means that the following explains the upper two layers of figure 3.

### 4.2 The Cindy Implementation

The concrete ASM specification is an extension of the abstract specification, since all the agents that are not refined work as on the abstract level. What is added is the Java implementation for the phone.

As an example, the mobile phone implementation of Cindy is partially listed below:

```
public class Cindy {
    private CommInterface comm;
    private static Protocol theinstance;
    private Doclist tickets;

    public Cindy(CommInterface comm){  ... }

    public void step(){
        if(comm.available()){
            Document inmsg = comm.receive();
            phoneStep(inmsg);
        }  else {  ... } }

    private void phoneStep(Document inmsg) {
        Document originator = inmsg.getPart(1);
        inmsg = inmsg.getPart(2);
        Doclist ticket = getTicket(inmsg, originator);
        if(ticket != null && tickets.len() < MAXTICKETLEN){
            tickets = tickets.attach(ticket);} }
    ...}
```

The class Cindy is responsible for executing the protocol steps. Communication with other participants is handled by the `CommInterface comm`, which is a field of class `Protocol`. The `CommInterface` implements a mechanism for sending and receiving objects of the class `Document`, which is the Java counterpart for the abstract **Document** type used in the protocol ASM. For every abstract document there exists one Java class which represents this document. The source code above is the main skeleton for receiving a ticket from the cinema. The method `step()` first tests whether a MMS message is available in the phone (`comm.available()`). Afterwards the incoming MMS is received and converted to a Java Document (`inmsg = comm.receive()`). The method `phonestep(inmsg)` extracts the ticket from this message and stores it in the local ticket store (`tickets.attach(ticket)`). This implementation is closely related to the abstract ASM rule for receiving a ticket, but has to deal with Java specific details like Nullpointer Exceptions.

During the initialization step of the concrete ASM the Java constructor of the Cindy class is called. In figure 3 this is shown by the Java step $cons_{mp}$ for the mobile phone and $cons_{cin}$ for a further refinement of the cinema's server application. The object resulting from the constructor call is assigned to the static field `theinstance` of the Cindy class. All further method calls are done on this object.

# 5 Proving Properties

For Cindy we consider two different kinds of security properties. Two security properties for the cinema are formalized and proved within the model, but we chose the customer's point of view as a main focus of the analysis of the Cindy application. Therefore we formalized and proved three properties that a user of the service will expect (cf. section 2). All formal specifications, theorems and proofs can be inspected on our project web page [KIV].

## 5.1 Security Properties of the Cinema

The following two important security properties are proven for the cinema:

1. Only tickets (i.e. nonces) issued by the cinema are accepted at the turnstile.
2. Each ticket is accepted at most once.

Given the representation of the state of the cinema (cf. section 3.3) the properties can be expressed as follows:

1. $\forall$ *ticket*. *ticket* $\in$ accepted(cinema) $\rightarrow$ *ticket* $\in$ issued(cinema)
2. $\neg$ duplicates(accepted(cinema))

If these two properties hold in every state that can be reached by the agent representing the cinema the corresponding security property holds at any time. Proving that these properties hold in every state is quite simple. It is just proved that the properties are invariant with respect to the protocol ASM. This is done by symbolic execution. The proof contains one branch for each possible step of the ASM and in each branch (i.e. after each protocol step) it must be proved that the property holds in the modified state given it was true in the initial state. The KIV verification systems achieves a high degree of automation in the proofs (approximately 80 percent). E.g. the proof obligation for the second property is:

**tickets-accepted-only-once:**
$\neg$ duplicates(accepted(cinema))
$\rightarrow$ [CINDY-STEP(*as*)] $\neg$ duplicates(accepted(cinema))

This theorem uses *as* as abbreviation for the complete state of the ASM. It states that if the accepted tickets initially had no duplicates then it holds that after all possible steps[4] of the protocol ASM the list still contains no duplicates. The proof of this property is done by symbolic execution of the ASM and almost automatic (317 proof steps and 2 interactions).

## 5.2 Security Properties of the Customer

In section 2 three properties that the customers of the service would expect were introduced. Formulating these security properties is not straightforward

[4] This is expressed using the box-operator of Dynamic Logic [HKT00]. $[\alpha]\ \varphi$ states that after all terminating runs of program $\alpha$ the property $\varphi$ holds.

because they deal with availability and we do not use a temporal logic that offers operators like *eventually*. Instead, the look ahead contained in formulations like 'he will eventually receive it' is replaced by a backward analysis of all traces of the application that end in a well-formed final state. The well-formedness condition wfc(*as*) of the final state is used as termination criteria for the ASM, i.e. when the ASM terminates, the state is well-formed. In the Cindy scenario the termination condition demands that all the tickets that were issued by the cinema (and that were not lost because they were sent to a phone number that does not exist or to a phone that has no space left) were presented at the turnstile and that there are no more unprocessed messages. This can be seen as the closing of the cinema at the end of the day, when all shows are finished and all tickets were presented[5].

The properties from section 2 have the following formal representations:

1. $\forall$ *agent.* mobile-phone?(*agent*)
   $\rightarrow \forall$ *ticket.* ordered-for(*ticket*, *agent*, booked)
   $\rightarrow$ *ticket* $\in$ tickets(*agent*)
   *(If the user orders a ticket he will eventually receive it)*
2. $\forall$ *agent.* user?(*agent*)
   $\rightarrow$ # bill(*agent*, issued) $\leq$ # booked(*agent*, booked)
   *(If the user is charged for a ticket he ordered it.)*
3. $\forall$ *ticket, agent.* owner-accepted(*ticket*, *agent*, tickets, passedOn, presented, accepted-with-presenter, inputs)
   *(If the user has a ticket and presents it at the turnstile, then he will be admitted.)*

The definitions given above are, of course, just a snippet of the real properties. E.g. the predicate *owner-accepted* used in property 3 has the following definition which states that each ticket that is stored in a mobile phone, and that was not passed on to another phone, and that was received from the cinema, and that was presented at the turnstile was accepted. The definition of *owner-accepted* is:

owner-accepted(*ticket*, *agent*, tickets, passedOn, presented, accepted-with-presenter, inputs)
$\leftrightarrow$ mobile-phone?(*agent*)
$\wedge$ cinema-ticket-tickets(*ticket*, tickets(*agent*))
$\wedge \neg$ ticket-forwarded(*ticket*, *agent*, passedOn)
$\wedge$ *ticket* $\in$ presented(cinema)
$\rightarrow$ doclist(intdoc(*agent*.no) + noncedoc(*ticket*))
$\in$ accepted-with-presenter(cinema)

However, to actually prove the invariance of one of the security properties a lot of additional information is necessary. To prove property three more than 10 additional preconditions are necessary, e.g. stating that certain parts of the agent's states are well-formed. Additionally, the history of each ticket must be represented in the invariant. Basically this means that for all tickets stored in

[5] We assume that anybody who has a cinema ticket actually comes to see the movie.

a mobile phone their complete life-cycle must be expressed. In total, the state invariant contains almost 30 predicates, each describing a different aspect of the state of the application.

### 5.3 Correctness of the Refinement

In general we show that for every concrete step $\mathbf{step_c}$ starting in a state $cs$ that corresponds to an abstract state $as$ via the retrieve relation $\mathbf{R}$ there exists an abstract step $\mathbf{step_a}$ whose result state also corresponds to the concrete result state via the retrieve relation. The ASM refinement methodology leads to the following proof obligation:

$$\begin{array}{l}\forall as, cs, cs', st, st' \ . \\ \quad as \ \mathbf{R} \ (cs \times st) \wedge \\ \quad (cs \times st) \ \mathbf{step_c} \ (cs' \times st') \rightarrow \\ \qquad \exists \ as' . \ as \ \mathbf{step_a} \ as' \wedge as' \ \mathbf{R} \ (cs' \times st')\end{array}$$

Additionally, it must be proven that the initialization and finalization steps are correct:

$$\forall as, cs, st \ . \ \mathbf{init_c}(cs, st) \wedge as \ \mathbf{R} \ (cs \times st) \rightarrow \mathbf{init_a}(as)$$
$$\forall as, cs, st \ . \ as \ \mathbf{R} \ (cs \times st) \wedge \mathbf{fin_c}(cs, st) \rightarrow \mathbf{fin_a}(as)$$

To prove these obligations we need a model in which the Java implementation can interact and communicate with the other agents in the scenario that are still specified by the rules of the protocol ASM and working on a state given by state functions. The following code snippet is taken from the ASM rule for the mobile phone on the concrete level:

```
...
if(agent = mobile-phone) then //rule for Java part
    TOSTORE(cs, st);
    choose st1 with ⟨st; Cindy.theinstance.step(); ⟩ (st = st1) in
        FROMSTORE(st1, cs)
else ... // rules for other agents
```

If the agent is the mobile phone, a Java step is done. Otherwise, the rule for the agent is specified as on the abstract level. The Java step requires the store for the concrete Java method call of the mobile phone implementation. For this, the refined ASM also contains the state of the mobile phone given by state functions. Before calling a Java method, the current abstract state is converted into Java objects and put into the store (TOSTORE($cs$, $st$)). The Java method works on this converted state by executing the Java protocol step (`Cindy.theinstance.step();`). Afterwards the state of the refined agent is extracted from the store and transformed back into the state functions (FROMSTORE($st_1$, $cs$)).

For the refinement we need the retrieve relation which links abstract and concrete state. With the integration mechanism for Java described above, this relation is quite simple:

$$as\ \mathbf{R}\ cs \times st \leftrightarrow as = cs \wedge \mathrm{INV}(st) \wedge \mathrm{INV}(as)$$

Since the refined ASM is an extension of the protocol ASM, the states of the agents have to be the same. Additionally, we need some technical invariants on the Java store and an invariant on the abstract state. The invariant on the Java store basically says that the objects in the store are well-formed, meaning that e.g. the list of tickets is not `null` and contains no other documents than tickets. The invariant on the abstract state *as* is used e.g. to express that messages are always well-formed (e.g. every MMS received by the phone contains the originator's phone number).

The proofs of the refinement properties pose some difficulties. One major problem is the encoding of the abstract state into Java objects and the corresponding backward transformation. Many lemmas are needed that are used during the proofs to close side goals. Additionally, a quite complex invariant about the concrete Java store is needed to ensure that the pointer structures and types of objects are all correct.

An example of a typical problem of this type of refinement is the encoding of abstract documents. It is fairly straightforward to write an encode-function that transforms instances of the abstract data type **Document** into Java objects. This means that an abstract data type is implemented by a Java pointer structure. But on the concrete level, more pointer structures are possible than abstract documents. Those are e.g. pointer structures containing null pointers or cyclic pointers. These are valid Java objects, but have no abstract data type counterpart. However, they can be constructed by an attacker in the real application and are a typical reason for errors in the implementations. As a consequence the refinement has to treat those documents during the step of the refined agent. Since they can occur in reality, they must be legal inputs for the refined agent. This is achieved by integrating them in the *TOSTORE* rule. The implementation has to ensure that those undesired pointer structures are treated with an error handling mechanism without crashing.

Since many of the problems during refinement do not depend on the particular application we have a library of reusable functions and predicates for those aspects of the refinement proof. The automation of the proofs is enhanced with every case study.

## 6 Conclusion

We presented an approach for the development of secure m-commerce applications. Electronic cinema tickets were used as an example. The approach supports the full development process starting from specification and continuing down to an implementation. The approach supports different means of communication and different attacker models. In the example, we considered availability properties instead of the standard confidentiality or authentication properties.

We start with the specification of the security protocol as an ASM, and prove application specific security properties. The proof method uses symbolic execution of the ASM and invariants over the possible traces. Additionally, we refine

the abstract specification into real Java source code and verify that this code is a correct implementation of the security protocol. This is not trivial: It is very easy to make the protocol specification too abstract (so it cannot be implemented). The refinement is based on the ASM refinement method, which is a well-known and established technique for the stepwise development of concrete implementations. The whole method is fully supported by the KIV system [BRS+00], our interactive theorem prover. All specifications and proofs can be found on our webpage [KIV].

## References


[AN95] R. Anderson and R. Needham. Programming Satan's Computer. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*. Springer LNCS 1000, 1995.

[BAN89] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society of London*, (Series A, 426, 1871), 1989.

[BDW99] C. Bolton, J. Davies, and J.C.P. Woodcock. On the refinement and simulation of data types and processes. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proceedings of the International conference of Integrated Formal Methods (IFM)*, pages 273–292. Springer, 1999.

[Bee] *Tickets on your Mobile.* URL: `http://www.beep.nl` [last seen 2006-03-16].

[BGW01] Nikita Borisov, Ian Goldberg, and David Wagner. Intercepting Mobile Mommunications: The Insecurity of 802.11. In *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 180–189, New York, NY, USA, 2001. ACM Press.

[BMV03] David Basin, Sebastian Mödersheim, and Luca Viganò. An On-The-Fly Model-Checker for Security Protocol Analysis. In *Proceedings of Esorics'03*, LNCS 2808, pages 253–270. Springer-Verlag, Heidelberg, 2003.

[Bör03] E. Börger. The ASM Refinement Method. *Formal Aspects of Computing*, 15 (1–2):237–257, November 2003.

[BRS+00] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in LNCS. Springer-Verlag, 2000.

[BS03] E. Börger and R. F. Stärk. *Abstract State Machines—A Method for High-Level System Design and Analysis.* Springer-Verlag, 2003.

[DB01] J. Derrick and E. Boiten. *Refinement in Z and in Object-Z : Foundations and Advanced Applications.* FACIT. Springer, 2001.

[dRE98] W. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1998.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. In *IEEE Transactions on Information Theory*, volume 29, 1983.

[GSR05] H. Grandy, K. Stenzel, and W. Reif. Object-Oriented Verification Kernels for Secure Java Applications. In B. Aichering and B. Beckert, editors, *SEFM 2005 – 3rd IEEE International Conference on Software Engineering and Formal Methods.* IEEE Press, 2005.

[Gur95] M. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9 – 36. Oxford University Press, 1995.

[HGRS05] D. Haneberg, H. Grandy, W. Reif, and G. Schellhorn. Verifying Security Protocols: An ASM Approach. In D. Beauquier, E. Börger, and A. Slissenko, editors, *12th Int. Workshop on Abstract State Machines, ASM 05*. University Paris 12 – Val de Marne, Créteil, France, March 2005.

[HHS86] He Jifeng, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *Proc. ESOP 86*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer-Verlag, 1986.

[HKT00] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[HRS02] D. Haneberg, W. Reif, and K. Stenzel. A Method for Secure Smartcard Applications. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology, Proceedings AMAST 2002*. Springer LNCS 2422, 2002.

[KIV] Web presentation of KIV projects. URL: `http://www.informatik.uni-augsburg.de/swt/projects/`.

[Low96] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055, pages 147–166. Springer-Verlag, 1996.

[Mea96] Catherine Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

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

[Sch01] G. Schellhorn. Verification of ASM Refinements Using Generalized Forward Simulation. *Journal of Universal Computer Science (J.UCS)*, 7(11):952–979, 2001. URL: `http://hyperg.iicm.tu-graz.ac.at/jucs/`.

[Sch05] G. Schellhorn. ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison. *Journal of Theoretical Computer Science*, vol. 336, no. 2-3:403–435, May 2005.

[Ste04] K. Stenzel. A formally verified calculus for full Java Card. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology (AMAST) 2004, Proceedings*, Stirling Scotland, July 2004. Springer LNCS 3116.

[Ste05] Kurt Stenzel. *Verification of Java Card Programs*. PhD thesis, Universität Augsburg, Fakultät für Angewandte Informatik, URL: `http://www.opus-bayern.de/uni-augsburg/volltexte/2005/122/`, 2005.

[Sun] Sun Microsystems Inc. Java Micro Edition. URL: `http://java.sun.com/j2me/index.jsp`.

[WD96] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.

[WS96] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *2nd USENIX Workshop on Electronic Commerce*, November 1996. A revised version is available at http://www.schneier.com/paper-ssl.html.