UNIVERSITÄT AUGSBURG
Universitätsbibliothek

OPEN ACCESS
OPUS AUGSBURG

Verifying smart card applications: an ASM approach

Dominik Haneberg, Holger Grandy, Wolfgang Reif, Gerhard Schellhorn

Angaben zur Veröffentlichung / Publication details:

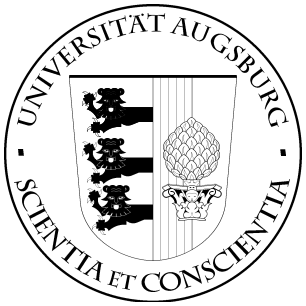
Haneberg, Dominik, Holger Grandy, Wolfgang Reif, and Gerhard Schellhorn. 2006. “Verifying smart card applications: an ASM approach.” Augsburg: Universität Augsburg.

Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under these conditions:
Deutsches Urheberrecht
Weitere Informationen finden Sie unter: / For more information see:
https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publiz/

Universität Augsburg



Verifying Smart Card Applications: An ASM Approach.

D.Haneberg, H.Grandy, W.Reif, G.Schellhorn

Report 2006-08 March 2006

Institut für Informatik
D-86135 Augsburg

Copyright © D.Haneberg, H.Grandy, W.Reif, G.Schellhorn
Institut für Informatik
Universität Augsburg
D–86135 Augsburg, Germany
http://www.Informatik.Uni-Augsburg.DE
— all rights reserved —

Verifying Smart Card Applications: An ASM Approach.

Dominik Haneberg, Holger Grandy, Wolfgang Reif, Gerhard Schellhorn

Lehrstuhl für Softwaretechnik und Programmiersprachen
Institut für Informatik, Universität Augsburg
86135 Augsburg Germany

E-Mail: {haneberg, grandy, reif, schellhorn}@informatik.uni-augsburg.de

March 13, 2006

Abstract

We present a formal model for security protocols of smart card applications using Abstract State Machines [BS03, Gur95] (ASMs) and a suitable method for verifying security properties of such protocols. The main part of this article describes the structure of the protocol-ASM and all its relevant parts. Integrated in the ASM are all relevant aspects of the scenario: The agents participating in the application (static and dynamic aspects), the attacker and the possible communication between all those involved in the application. Our modeling technique enables an attacker model exactly tailored to the application under consideration, instead of only an attacker similar to the Dolev-Yao model.

We also introduce a proof technique for security properties of the protocols. Properties are proved in the KIV system using symbolic execution and invariants.

Our formal approach is exemplified with a small e-commerce application. We use an electronic wallet to demonstrate the ASM-based protocol model and we also show how the proof obligations of some of the security properties look like.

1 Introduction

Smart cards are computers fitting in a wallet. They store information and independently execute specific programs. Their most exceptional characteristic is the fact that they are tamper-proof. Because of this they are well suited for storage of security critical information. Smart cards are used for digital signatures, access control, electronic wallets, electronic ticketing and so on. Communication is an integral part of such applications, the transmitted data contains such crucial data as customer data or electronic business goods. Such data must be protected while in transfer. This is generally done using security protocols. These protocols try to guarantee security goals by a proper combination of cryptographic primitives. Security protocols exist in a wide variety of types, from simple authentication (e.g. Needham-Schroeder) or key-agreement (e.g. ElGamal) protocols up to complex protocols such as SSL [FKK96] or SET [Mas97]. Depending on the application, sometimes these standard protocols are not sufficient or applicable. An application can have very specific security demands that are not fulfilled by standard protocols. The security properties guaranteed by standard protocols are often just a building block for the real security properties of the application, so the need for application-specific security protocols arises. There are other limits to the usability of standard protocols too, e.g. because of hardware restrictions of smart cards standard protocols cannot be used.

Unfortunately designing cryptographic protocols is very error-prone, i.e. it is very hard to design them correct [AN95]. To overcome the problem of bad security protocols, formal methods are used for the verification of the protocols. Different approaches for the verification of security

protocols can be found in the literature, e.g. model-checking based approaches [Low96][BMV03], interactive theorem proving [Pau98] or specialized logics of belief [BAN89].

In this article we present a particular approach to verify security protocols. We use an Abstract State Machine (ASM) [Gur95][BS03] to model the application scenario. ASM are a powerful, yet easy to understand, method for specification with a wide range of possible applications. ASM were successfully used for e.g. the description of programming language semantics [BS98][BFGS05], virtual machines [BS00] and logical calculi [BS97]. Represented in the ASM is all relevant information for verifying security properties of the protocols used in the application: Static and dynamic aspects of the agents taking part in the protocols, the so-called attacker, a malicious participant of the application, and the communication between the agents. The static aspect of an agent is the structure of its internal state (which fields does the agent have to store information). The dynamic aspect of an agent is how it processes messages it receives, depending on the content of the received message and its state. Data is modeled using algebraic specifications, the ASM is described by rules manipulating the state of the agents. Our approach offers a flexible attacker model in order to tailor the attacker to the investigated application. Proofs are generally invariant or inductive proofs; proofs for protocol properties are done with the KIV system [BRS+00], our interactive theorem prover. The KIV system is a tactical theorem prover for Higher-Order Logic. It is based on structured algebraic specifications and contains, besides the Higher-Order Logic calculus, a Dynamic Logic (DL) [HKT00] calculus for imperative programs as well as for Java and a calculus for Temporal Logic. KIV offers powerful heuristics and an elaborate correctness management. The verification of security properties is simplified through the use of DL invariants.

The main result of the work presented in this paper is a formal model for security protocols and the associated application that is very suitable for the verification of (application-specific) security properties, that is easy to understand and can even be generated automatically from a graphical notation.

The ASM protocol model is part of a larger project dealing with the verification of smart card applications. This project comprehends a graphical notation to model the scenario and the protocols [HRS02], the formal ASM model, an appropriate verification strategy, a refinement of the abstract protocols to Java code and the verification of implementation correctness. To correspond to Java our ASM model contains agent states explicitely. The overall goal of the project is the seamless verification of security protocols beginning with an abstract protocol model and ending with verified program code. Although our approach is tailored to the verification of application specific security protocols in smart card applications, the verification of common cryptographic protocols is possible as well. Using an ASM to formalize the protocol model is motivated by the facts that verification of ASMs within the KIV system works quite well and the automatic generation of the ASM from the graphical notation is feasible.

In Section 2 we describe the structure of the abstract model of the smart card applications and introduce the different parts of the model, Section 3 describes the messages used for communication between the agents, Section 4 introduces the communication structure, Section 5 describes how the state of the agents is modeled, in Section 6 our attacker model is introduced, Section 7 describes how all these parts are integrated into an ASM describing the behavior of the agents. In Section 8 our concept is demonstrated using a small example and Section 9 describes the security properties of the application and gives details on the proofs and our special DL invariant. Section 10 discusses some related work and finally Section 11 concludes.

2 Structure of the Application Model

In this section we will give an overview of the structure of the model we use to describe smart card applications formally. As foundation of our model we use algebraic specifications. On top of the algebraic specifications we can declare the static aspects of the agents (i.e. the fields they use to store information about their state). For each type of agent in the application we then add an ASM rule that describes such an agent's actions in the application (i.e. these ASM rules describe the protocol steps and the treatment of errors done by the different agent types). This is the

dynamic part of the application model. The ASM rules define an ASM that models the possible runs of the application by nondeterministically creating sequences of steps of the agents present in the scenario.

In the algebraic specification we have the definitions of the used data types, such as integers, lists and sets. The messages used in the data exchange between the agents are an algebraically specified data type (cf. Section 3). There are also predicates in this part that describe the structure of the communication network of the application, i.e. which agents can exchange messages using which type of connection (cf. Section 4). On top of the communication structure the abilities of the attacker to influence the communication are axiomatized (cf. Section 6.2 and Section 7.1).

On top of the algebraic specification the ASM is defined. Our model contains one ASM rule for each type of agent. The agents consist of the users of the application and the attacker on one hand and the different systems implementing the service, e.g. smart cards or servers on the Internet, on the other hand. In an e-commerce application the different systems usually offer a number of functionalities. Each functionality is realized by a protocol specifying the data exchanges necessary for this functionality. Each agent type can be involved in different functionalities and therefore has to realize protocol steps for different protocols. The ASM rule for one type of agent completely specifies the behavior of an agent of this type, therefore all protocol steps an agent is expected to perform are represented in the ASM rule.

Having specified the behavior of the different agent types, we build the abstract model of the application. This is an ASM that combines the ASM rules for the different agent types. It nondeterministically chooses an existing agent and executes the step this agent can perform in the current state. Thereby the ASM represents all possible traces that can occur in the application. In our model the possible traces are declared only implicitly whereas Paulson in [Pau98] models traces explicitly.

3 Specification of Messages

The basis of the specification of the protocols is formed by the data exchanged between the agents. To model the data we use an algebraic data type called *document*. Documents may be basic data such as *nonces*[1] or *integers* or complex documents such as lists of other documents, hash values or encrypted documents.

The documents for communication are described by the following freely generated algebraic data type. The specification declares documents and lists of documents as mutually recursive data types. Documents can be combined to lists of documents and lists of documents can again be documents. The specification is:

```
document = ⊥
         | intdoc (. .int : int ) with is-intdoc
         | keydoc (. .key : key ) with is-keydoc
         | noncedoc (. .nonce : nonce ) with is-noncedoc
         | secretdoc (. .secret : secret ) with is-secdoc
         | hashdoc (. .doc : document ) with is-hashdoc
         | encdoc (. .key : key ; . .doc : document ) with is-encdoc
         | sigdoc (. .key : key ; . .doc : document ) with is-sigdoc
         | doclist (. .list : documentlist ) with is-doclist
         ;
documentlist = []
             | . + . (. .first : document ; . .rest : documentlist )
             ;
```

This is a specification of a freely generated data type as usual in functional languages. The specification for *document* contains constructors to create a document from basic data types, e.g.

[1] A nonce is a random number that cannot be guessed by the attacker. They are generated on demand with the purpose of being used in a single run of a protocol.

a document that contains an integer. Besides these basic constructors there are also constructors for complex documents, e.g. a document that represents the hash value of another document. Encryption is also represented by a constructor for document. We support symmetric as well as asymmetric encryption. Which encryption type is used is determined by predicates defined on the used key. A key can be a *public key* or a *private key* or a *symmetric key*. A symmetric key enforces symmetric encryption, a key from an asymmetric key pair leads to asymmetric encryption. Our specification of messages is very similar to the one presented by Paulson in [Pau98] as data type msg. We added a new constructor secretdoc for secrets. Secrets are used to model long term secrets such as the PIN of the owner of the smart card. Also for the signature of a document Paulson does not have a special constructor.

Since hashed documents are realized using the constructor function hashdoc: *document* → *document* of the freely generated data type *document* the hash function is injective. This assumption is common to abstract verification of cryptographic protocols. We also assume, again as usual in cryptographic protocol analysis, perfect encryption, i.e. an encrypted document can be decrypted to its plaintext only if the correct decryption key is known. Otherwise the decryption leads to an integer about which nothing is known. This is expressed by the axioms for the decrypt functions, e.g. for asymmetric encryption we have:

$$\begin{aligned}&\text{decrypt-keypair-ok:}\\&\text{is-keypair(key, key}_0\text{)} \rightarrow \text{decrypt(key}_0\text{, encdoc(key, doc))} = \text{doc}\end{aligned} \tag{1}$$

$$\begin{aligned}&\text{decrypt-keypair-fail:}\\&\neg\ \text{is-keypair(key,key}_0\text{)} \wedge \neg\ \text{(is-sessionkey(key)} \wedge \text{is-sessionkey(key}_0\text{))} \rightarrow\\&\text{is-intdoc(decrypt(key}_0\text{,encdoc(key,doc)))}\end{aligned} \tag{2}$$

Axiom 1 states that decryption with a key that forms an asymmetric key pair with the encryption key produces the correct plaintext as result, axiom 2 describes unsuccessful decryption in the asymmetric case. If the key used for decryption and the key used for encryption do not form a key pair and they are not both symmetric keys the decryption is unsuccessful and the produced result is an unspecified integer document.

4 Communication

In general the formal approaches for verifying cryptographic protocols use a rather simple model of communication. In these approaches a data transfer is usually modeled with a statement of the form 'participant A sends message m to participant B'. Different types of connections (e.g. radio-based vs. wire-based) are ignored as well as the possibility that certain agents may not be allowed to communicate directly with certain other agents, e.g. the holder of a smart card cannot communicate directly with his smart card, he must use a smart card terminal to do so. Also the possibility that certain connections are secure against eavesdropping and others are not is abstracted away. This approach is sufficient for Dolev-Yao attackers [DY81] only, because this attacker can communicate with every other agent without any restriction.

In contrast our specification and verification technique is not limited to the Dolev-Yao attacker. Some of the applications we want to analyze justify to put certain restrictions on the attacker's abilities. For example in some smart card applications, the communication between a smart card and the card terminal may be considered as secure. The main idea behind an attacker with reduced abilities is that a less powerful attacker may permit the usage of simpler protocols. Because of this, using a less powerful attacker is very interesting, if this is appropriate for the given application scenario.

For example in [Bel01] an attacker with incomplete control of the communication is used for the verification of the Shoup-Rubin protocol, a protocol for key exchange using smart cards. The smart cards output the calculated session key in clear, therefore proving secrecy properties would fail for an attacker that can eavesdrop into the outputs of the smart cards. [GHRS06] presents a mobile ticketing application for which a security property is proven that only holds if the attacker can only eavesdrop into but not manipulate certain message transfers.

But the option to use a non Dolev-Yao attacker has a price. The part of the model describing possible data transfers is more complex. In our model we have explicit connections between agents that allow messages to be transfered. It is not possible to make a connection between an arbitrary pair of agents, only such connections that are possible in the application scenario can be created. For each connection we explicitly determine if the attacker can read messages that are transfered using this connection and if he can manipulate such messages.

The two main concepts we use are *ports* and *connections*. Each agent has certain ports, their number and their properties depend on the type of the agent. For example a smart card usually has one port, representing the serial interface of the smart card, a card terminal often has two ports, a smart card reader and an interface for the user. Each port has specific properties, e.g. the port of a smart card can be connected with a smart card terminal but not with another smart card. Associated with each port is the list of messages that were sent to this port but have not yet been processed by the agent. If a port is not connected this list is empty. Connections are established between two different ports of two different agents. They permit bidirectional communication between these agents. When $agent_1$ sends a document using his port $port_1$ that is connected to port $port_2$ of $agent_2$ the document is added to the list of unprocessed messages of $port_2$ of $agent_2$.

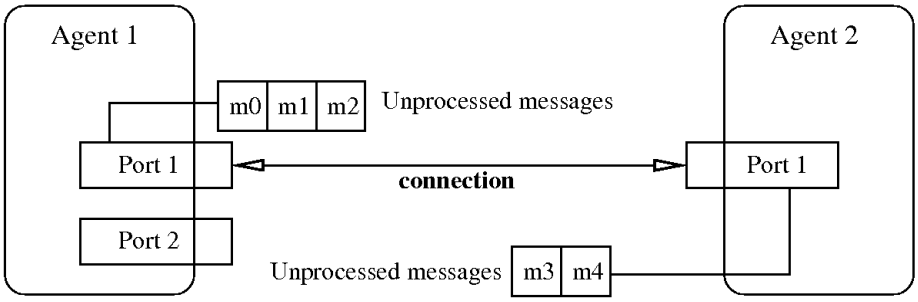


Figure 1: Example of communication structure

Figure 1 shows a small example of a communication structure. There are two different agents, agent 1 has two ports and agent 2 has one. Between the two agents one connection is established and there are some unprocessed messages available for both agents.

The currently established connections are stored in the set *connections*. A connection is described by a 4-tuple ($agent_1$, $port_1$, $agent_2$, $port_2$). The first and the second part of the tuple are the identity of the first agent connected by the connection and the port used by the first agent, parts three and four represent the second connected agent. For example the connection in figure 1 is represented by (Agent 1, 1, Agent 2, 1). We use the notation ($agent$,$port$,$\cdot$,$\cdot$) $\in$ *connections* to denote that currently a connection for port *port* of agent *agent* exists. We also keep a list of unprocessed messages for each agent. These are all messages that were sent to an agent but not yet consumed by it. We use this to model the messages that are in transit over the network. The unprocessed messages are stored using the dynamic function inputs. Because each agent may have multiple input ports we have $agent \times port \rightarrow list(document)$ as the signature of the dynamic function inputs.

5 Modeling the Agents

Distinct from most other approaches to security protocol verification (for example [Pau98]), we explicitly model the internal state of the agents. Each agent has its own state described by the fields in which it stores values, e.g. its private key or a nonce. Having this state is useful for describing certain security properties (as described in [HRS02]). Also it is a good starting point for the future refinement of the abstract protocol specifications to real code. An implementation of course has a notion of program state and in order to express a refinement relation a corresponding

notion of state like the balance of an electronic purse seems natural for the abstract model too. Although modeling the state of the agents increases the complexity of the model, it is in our opinion the most natural approach to model the applications we are interested in.

Within this article we use the term agent to denote a participant of the application under investigation. An agent has a unique type. Each agent (including the attacker) is of one of the agent types $at \in \textit{agent-types}$. For example in a smart card based electronic wallet, there are the smart card itself and different terminals, e.g. a point-of-sale terminal for paying and a terminal for loading additional money on the card, as well as the attacker and the owners of the smart cards. For this example we have $\textit{agent-types} = \{\text{attacker, user, terminal, card}\}$.

The internal state of the agents is modeled as follows: For every field of every agent a dynamic function is used that maps the agent to its current value of the field. This means that an agent with fields $f_1, \ldots, f_m$ of sorts $s_1, \ldots, s_m$ is represented by dynamic functions f_1: $\textit{agent} \to s_1$, ..., f_m: $\textit{agent} \to s_m$. Suppose we have a terminal with two fields, one for its secret key and one for a nonce. For this agent we have:

$$\mathsf{secret\text{-}key}\text{: } \textit{agent} \to \textit{key};$$
$$\mathsf{session\text{-}nonce}\text{: } \textit{agent} \to \textit{nonce};$$

Using algebraic specifications, we could instead define a new tuple sort for the state of each type of agent, but using dynamic functions instead has advantages. It avoids the frame problem that occurs when updating the state and this prevents very large goals and potential efficiency problems in proofs.

6 The Attacker

When analyzing security protocols, the possible threat is a crucial aspect. Typically an active malicious agent is assumed, the attacker. The attacker may have control over at least parts of the communication between the honest agents, i.e. he can read or even manipulate the exchanged messages. By eavesdropping on the communication and by analyzing the messages the attacker accumulates a set of documents. This set is the so-called knowledge of the attacker. In the rest of this paper the attacker's knowledge is represented by the set *attacker-known*. In order to build new messages the attacker can use all the documents contained in his knowledge.

Usually an attacker similar to the Dolev-Yao attacker is used. Given the usual assumptions on cryptographic primitives (e.g. perfect encryption) this is the strongest possible attacker as he has complete control over the communication. He can intercept and manipulate all messages, build arbitrary new messages from his knowledge and decompose messages he knows. He also decrypts all messages for which he has the appropriate decryption key. However, the Dolev-Yao attacker is not always adequate, in some cases a reduced attacker model is more realistic. This can be realized easily in our model. If the attacker cannot eavesdrop into certain communication links it is possible that the data which is transferred needs less protection. This means that possibly fewer cryptographic primitives must be used and as a result the security protocols become less complex and more efficient.

The attacker model consists of three distinct parts. The first describes how the attacker can decompose messages and build new ones. This aspect of the attacker's abilities is independent of the application under investigation. The second part describes the attacker's abilities to eavesdrop or manipulate the communication. These first two parts of the attacker model are axiomatized using first-order formulas. On top of them, an ASM rule describes the possible activities the attacker may perform. This builds the third part of the attacker description.

6.1 Treatment of Messages

This part of the attacker's specification describes the handling of documents. First we describe how the attacker can extract new information from documents he received and subsequently we describe the generation of new documents.

6.1.1 Analyzing Messages

The attacker decomposes all messages as far as possible. By decomposing documents the knowledge of the attacker (*attacker-known*) grows. If a new document *doc* becomes available to the attacker, either by decomposing another document or by eavesdropping on the part of the network that is under his control, the following rules are used to add it to the attacker's knowledge:

1. If *doc* is a basic document (except a document that represents a key) it is added to *attacker-known* and nothing further is done.
2. If *doc* is a document representing a key, *doc* is added to *attacker-known* and if there exist documents in *attacker-known* that can be decrypted with this key, the decrypted documents are added to *attacker-known* as well.
3. If *doc* is a list of documents, *doc* is added to *attacker-known* and all documents in the list are added to *attacker-known* as well.
4. If *doc* is an encrypted document, *doc* is added to *attacker-known* and if the appropriate decryption key is available the plaintext is added to *attacker-known* as well.
5. If *doc* is a signature or a hash document, *doc* is added to *attacker-known* and nothing further is done.

This decomposition is repeated until a fixpoint is reached[2]. It is important to note that acquiring one additional document by eavesdropping may substantially enlarge *attacker-known*, because it may allow the additional decryption of a lot of documents in *attacker-known*. In the following adding a new document *doc* to the attacker's knowledge *attacker-known* is written down as *attacker-known* $\int+$ *doc* where $\cdot \int+ \cdot$ is an infix function with signature *set(document)* $\times$ *document* $\rightarrow$ *set(document)* that adds *doc* to *attacker-known* and calculates the fixpoint under the decomposition. *attacker-known* $\int+$ *doc* is $\mathsf{analz}(\textit{attacker-known} \cup \{\textit{doc}\})$ in [Pau98].

6.1.2 Composing New Messages

As the set of documents that the attacker can produce is infinite, we cannot actually calculate it. Therefore we have specified a predicate $\cdot \rightsquigarrow \cdot$ that determines if an attacker with a given knowledge can generate a given document or not, i.e. $\textit{docset} \rightsquigarrow \textit{doc}$ states that the document *doc* can be constructed given the documents in *docset* ($\textit{docset} \rightsquigarrow \textit{doc}$ is equivalent to $\textit{doc} \in \mathsf{synth}(\textit{docset})$ in [Pau98]). As the axioms for this predicate are somewhat lengthy we will omit them and instead give the rules for $\rightsquigarrow$ in a notation similar to calculus rules.

$$\frac{i \in \mathbb{Z}}{\textit{docset} \rightsquigarrow \text{intdoc}(i)}\ \text{intdoc} \qquad \frac{\textit{docset} \rightsquigarrow \textit{doc}}{\textit{docset} \rightsquigarrow \text{hashdoc}(\textit{doc})}\ \text{hash}$$

$$\frac{\textit{doc} \in \textit{docset}}{\textit{docset} \rightsquigarrow \textit{doc}}\ \text{in docset} \qquad \frac{\forall i.(1 \leq i \leq n \rightarrow \textit{docset} \rightsquigarrow \textit{doc}_i)}{\textit{docset} \rightsquigarrow \text{doclist}(\textit{doc}_1 + \ldots + \textit{doc}_n)}\ \text{list}$$

$$\frac{\textit{docset} \rightsquigarrow \textit{doc} \quad \textit{docset} \rightsquigarrow \text{keydoc}(\textit{key})}{\textit{docset} \rightsquigarrow \text{encdoc}(\textit{key}, \textit{doc})}\ \text{encrypt}$$

$$\frac{\textit{docset} \rightsquigarrow \textit{doc} \quad \textit{docset} \rightsquigarrow \text{keydoc}(\textit{key})}{\textit{docset} \rightsquigarrow \text{sigdoc}(\textit{key}, \textit{doc})}\ \text{signature}$$

Each of these rules states that the proposition about the ability to produce a specific document (written under the horizontal line) is true if the preconditions specified above the horizontal line are true. E.g. by rule 'encrypt' an encrypted document can be produced if the encryption key and the plaintext document can be produced. The rules for the construction of documents imply that the attacker cannot guess random numbers (nonces) and keys because they can be constructed only by rule 'in docset', i.e. he can use them only if he learned them by eavesdropping.

Together this describes the most nondeterministic and powerful attacker as the attacker does all decompositions possible and can produce all documents that can be built with his knowledge.

[2]Since all documents added are part of $\{\textit{doc}\} \cup \textit{attacker-known}$, the set of documents that may be added is finite. Therefore the construction terminates.

6.2 Eavesdropping

In order to acquire new documents or to manipulate the messages in transit, the attacker must have access to the connections between the agents. Each agent has input ports and output ports for communication. A connection describes which output port of one agent is linked to which input port of another agent. For each input port a list of messages that were sent to this port but not yet processed by the receiver is stored.

In the Dolev-Yao model the attacker has access to all communications. As we want to be able to reduce the abilities of the attacker in order to work with attackers more adequate for a given scenario, we specify the attacker's access to the network in greater detail.

We use two predicates to describe what the attacker can do with a given connection. The predicate can-read(*conn*) is true if the attacker can eavesdrop on the connection *conn*. If the attacker can manipulate data transmitted over the connection *conn* the predicate can-write(*conn*) is true. In the case of a Dolev-Yao attacker both predicates are true for all possible connections. For a less powerful attacker these predicates are false for some connections. In [HRS04] we present some useful attackers weaker than the Dolev-Yao attacker.

7 The ASM

In the preceding sections we described the state of the agents and the attacker. In this section we will describe the dynamic aspects of the participants, i.e. how the protocol steps are modeled. The ASM describing the possible protocol runs consists of two parts:

1. An initial state that ensures that the internal states of the attacker and the agents are reasonable. What reasonable means is largely application specific, but some properties must always be ensured, e.g. the private keys of different agents must be different, no connections are established between the agents, the lists of unprocessed messages are empty for all agents, the private key and the public key of an agent form an asymmetric key pair, the list of nonces available to the agents must not contain duplicates, etc. The predicate admissible(...) states that all the necessary properties hold for a given state.

2. An ASM rule that performs one step in the protocol model.

In most cases more than one agent can perform an action. To perform a step, the ASM first of all chooses nondeterministically which agent performs the next step. From the viewpoint of proving, this ensures that all possibilities are considered. Let $\mathrm{at}_1, \ldots, \mathrm{at}_n$ be the different types of agents in an application (including the attacker and the user). We write agent-type(*agent*) = *at* if *agent* is of type *at*. Let ready(*agent*) be a predicate that determines if an agent is ready to perform a step. To select an agent the ASM performs the following choose:

APPLICATION =
  choose *agent* **with** ready(*agent*) **in** STEP

where STEP is the ASM rule that describes the actions of the agents.

After choosing the agent for the next step the ASM branches into the code describing the possible actions of the selected agent. Given that R_i is the rule for agent of type at_i ($i = 1, \ldots, n$) this is done by a **case** statement[3]:

STEP =
  case agent-type(*agent*) **of**
  at_1: R_1
  $\vdots$
  at_n: R_n

[3]Case statements are not part of traditional ASM syntax but can easily be defined as an abbreviation for a sequence of **if** statements.

7.1 The Attacker

The main actions of the attacker are eavesdropping and generation of new messages. These two operations will be described in the following. First we discuss eavesdropping. Because we have to deal with attackers different from the Dolev-Yao attacker we need a more detailed specification of the attacker's abilities to influence the communication, this means we must precisely specify which documents the attacker can obtain by eavesdropping and which he cannot. As described in Section 4 we distinguish between different communication ports for every agent and it is possible that the attacker may eavesdrop on some ports of an agent but not on the others. This is specified by the predicate *can-read* mentioned earlier. Given the information if the attacker has access to a connection or not, there are two possible approaches to realize the eavesdropping: one is to specify an explicit operation for the attacker in which he adds documents to his knowledge that are currently transmitted over a connection he can eavesdrop on. The other is to extend the mechanism used by the other agents to send their documents (SEND, cf. Section 7.2): instead of just putting the sent document at the end of one list of unprocessed documents of the receiver, the operation additionally checks if the attacker can eavesdrop on this data transfer. If this is the case, the document is directly added to the attacker's knowledge. Both choices are equivalent with respect to the power of the attacker. We chose the second approach because it simplifies invariants and reduces the size of proofs. Therefore the ASM rule SEND (cf. Section 7.2) that sends a document *outdoc* over a connection (*agent*, *outport*, *remote-agent*, *remote-port*) calls ATTACKER-ADD to ensure that the attacker receives all documents he may receive:

ATTACKER-ADD =
  if agent-type(*remote-agent*) = attacker
    ∨ can-read(*agent*, *outport*, *remote-agent*, *remote-port*)
  then *attacker-known* := *attacker-known* $\int+$ *outdoc*

ATTACKER-ADD tests if the attacker is the destined receiver of the document or if he can eavesdrop on the connection. If this is the case the document is added to attacker's knowledge and all messages in the resulting knowledge are decomposed as far as possible (cf. Section 6.1.1).

The second attacker operation is the generation of messages. The following ASM rule realizes this operation:

ATTACKER-SEND =
  choose *docs* **with** ∀ *doc* ∈ *docs*. *attacker-known* $\rightsquigarrow$ *doc* **in**
    choose *agent*, *port* **with** (∃ (*agent*, *port*, *remote-agent*, *remote-port*) ∈ *connections*.
                can-write(*agent*, *port*, *remote-agent*, *remote-port*))
      in inputs(*agent*, *port*):= *docs*

This rule chooses a list of documents (*docs*) that can be produced by the attacker and then chooses an existing agent and a port of this agent which can be modified by the attacker and replaces the list of messages on this port with *docs*. Whether the attacker can modify messages on a port is determined by the predicate can-write($agent_1$, $port_1$, $agent_2$, $port_2$). This predicate is true iff a message that is transmitted between port $port_1$ of $agent_1$ and port $port_2$ of $agent_2$ can be manipulated by the attacker.

7.2 Agents

The remaining part of the ASM describes the behavior of the other components of the application. When designing the security protocols it is determined for each agent what steps he has to perform in order to satisfy the protocol. Therefore it is declared in the protocol specification what changes to the state of the agent take place and what reply is generated, given a certain input and the current state of the agent. So each step s of an agent of type at consists of a condition $C_{at,s}$ that is true if this step can be performed and a rule $R_{at,s}$ that modifies the internal state of the agent. Usually a reply is produced as well and sent to some other agent. The produced reply is stored in the variable *outdoc*. As an agent may be connected to different other agents, the protocol must

also determine to whom the document is to be sent. This is stored in the variable *outport*. Finally SEND moves the generated document to the input port of the receiver. The condition $C_{at,s}$ takes into account the current state of the agent and the message it is currently processing.

Assume the agent of type *at* has m possible steps $R_{at,1}, \ldots, R_{at,m}$, then the rule for this agent has the following structure:

$R_{at} =$
  if $C_{at,1}(agent)$ **then** $R_{at,1}$
  ...
  else if $C_{at,m}(agent)$ **then** $R_{at,m}$
  SEND

Each of the $R_{at,s}$ rules describes one step in the protocol or the treatment of a protocol error. They typically manipulate the state of the agent and produce an answer, so if an agent of type *at* has fields $f_1, \ldots, f_n$ we have:

$R_{at,s} =$
  $\mathsf{f}_1(agent) := \ldots$
  $\vdots$
  $\mathsf{f}_n(agent) := \ldots$
  $outdoc :=\ldots$
  $outport :=\ldots$

It is not necessary that all fields of an agent are modified in a step, instead it is also possible that just a subset of all fields of the agent is changed.

After the agent performed its step and produced a document that must be transfered, the ASM selects the connection that belongs to *outport* and puts the generated document at the end of the list of unprocessed documents of the receiver's (*remote-agent*) input port (*remote-port*) belonging to the selected connection:

SEND =
  if (*agent*, *outport*, $\cdot$, $\cdot$) $\in$ *connections* **then**
    choose (*agent*, *outport*, *remote-agent*, *remote-port*) $\in$ *connections* **in**
      inputs(*remote-agent*, *remote-port*) :=
        inputs(*remote-agent*, *remote-port*) + *outdoc*
      ATTACKER-ADD

The **if** ensures that a connection currently exists for the designated output port. If no such connection exists the message is lost. If such connections exist, one is chosen and the document is transmitted. SEND is completed by using ATTACKER-ADD to ensure that the attacker receives the document if either the attacker is the destined receiver or the attacker can eavesdrop on the connection (cf. Section 7.1).

7.3 Generation of the ASM

Although the structure of the ASM is simple, creating the ASM specifying a complex application with different protocols is time-consuming. Therefore the ASM should be generated automatically. In [HRS02] we described an approach to model security protocols using UML-like diagrams and generate an algebraic specification as formal protocol model from the diagrams. With minor modifications the proposed transformation can be adapted to generate an ASM model of the protocols according to the description of the ASM given above.

8 A Small Example

In this section we illustrate our modeling technique using a small smart card application. The application is an electronic wallet, i.e. a smart card that can store money and be used for payment, for example in a university cafeteria or at copying machines. The smart card program (cardlet for short) stores the money as so-called *value points*, each representing a certain amount of money, e.g. one cent. The application scenario was already described in [HRS02] but the protocols considered here are different, since they are designed to be secure against a more powerful attacker. The example is not very complicated and only serves as illustration of the kind of applications we are interested in, how our formal model looks like, what type of security properties we want to verify and what proof technique is used. Larger applications we verified are the Mondex case study [SGHR06] and an electronic ticketing application [GHRS06].

As agents in this application we have the smart cards with the electronic wallet application, card terminals (which are used for loading money on the cards and for payment), the owners of the cards and the attacker. For reasons of simplicity we combined the functionality of the load and the point-of-sale terminals into one agent type and consider just one such terminal (THETERM) and one smart card (THECARD).

The user is quite simple, he can insert his smart card into the terminal or remove it and he can start protocol runs by giving an appropriate command to the terminal. The attacker is a Dolev-Yao attacker, he can read and manipulate the communication between the smart card and the terminal. The terminal and the smart card offer a set of functionalities important for an electronic wallet, e.g. loading money on the smart card, paying for goods and checking the balance.

As described in Section 7 the main ASM rule of the application chooses an agent and then the ASM rule for this agent is used to perform a protocol step. The ASM rule for the electronic wallet example is as follows:

E-WALLET =
  choose *agent* ∈ {attacker, user, term, card} **with** ready(*agent*) **in**
    case agent-type(*agent*) **of**
      attacker : ATTACKER
      user : USER
      term : TERMINAL
      card : CARDLET

ATTACKER, USER, TERMINAL, and CARDLET are the ASM rules for the four agent types. USER just sends commands to start protocol runs to the terminal, ATTACKER uses ATTACKER-SEND (cf. Section 7.1) to send messages, TERMINAL is the ASM rule for agents of type terminal and CARDLET is the rule for the e-wallet cardlets.

As in [HRS02] a secret information that is shared between the smart card program and the terminal is used to guarantee the authenticity of messages, but in this improved version the secret is never transmitted. Instead only hash values are exchanged between the agents and the secret cannot be extracted from a hash value. Because of this, the application is secure even against an attacker that can observe and manipulate all communication between the smart card and the terminal. Although the secret is never transmitted proving the properties is not trivial e.g. because of possible replay attacks.

8.1 The Attacker

As described in Section 6 the attacker model consists of different parts. The treatment of messages is independent of the concrete application, and was described in Section 6.1. The attacker's abilities to manipulate the communication were already mentioned above. In this application the attacker can read and manipulate the communication between the smart card and the terminal, so the predicates can-read and can-write are true for connections between smart card and terminal. The attacker still cannot read or manipulate data exchanged between the user and the terminal. This

is irrelevant in this application but important in others, e.g. if the user has to input a PIN to activate the smart card program.

In Section 7 we stated that the attacker has to choose between his possible actions. This is done in the following ASM rule:

ATTACKER =
  choose *step* ∈ {send, connect, disconnect} **with**
                ((*step* = send → attacker-can-send(*connections*))
                 ∧ (*step* = connect → connect-possible(*agent*, *connections*))
                 ∧ (*step* = disconnect → disconnect-possible(*agent*, *connections*))) **in**
      if *step* = send **then** ATTACKER-SEND
      else if *step* = connect **then** CONNECT
      else if *step* = disconnect **then** DISCONNECT

This chooses a step for the attacker and ensures that it is possible to perform this step. For example ATTACKER-SEND is only chosen if there exists a different agent that currently participates in a communication and the attacker may influence this communication. This is stated by the axiom for the predicate attacker-can-send(*connections*) which is:

attacker-can-send(*connections*) ↔
∃ (*local-agent*, *local-port*, *remote-agent*, *remote-port*) ∈ *connections*.
    ¬ agent-type(*remote-agent*) = attacker
  ∧ (agent-type(*local-agent*) = attacker
    ∨ can-write(*local-agent*, *local-port*, *remote-agent*, *remote-port*))

ATTACKER-SEND was described in Section 7.1, CONNECT and DISCONNECT open/close a connection by adding it to/removing it from *connections*.

8.2 The Terminal

8.2.1 Internal State of the Terminal

The terminal has 6 data fields: *secret*, *last-inst*, *challenge*, *issued*, *collected* and *loadpoints*. The fields *secret* contains the secret shared with the terminal. The field *last-inst* stores the last instruction that was sent to the card. The terminal needs this information to determine at which position in a protocol run it is, *challenge* contains the nonce currently in use. *loadpoints* stores the number of points to load on the cardlet (load protocol) or the number of points requested from the cardlet (pay protocol), *issued* stores the sum of points that were loaded onto cardlets by the terminal, *collected* stores the sum of the points that were cashed as payments by the terminal. The terminal requires 6 dynamic functions with the following signatures: secret: *agent* → *secret*, last-inst: *agent* → *instruction*, challenge: *agent* → *nonce*, issued: *agent* → *int*, collected: *agent* → *int* and loadpoints: *agent* → *int*.

8.2.2 ASM rule of the Terminal

In this section we show a part of the terminal ASM rule. We focus on the protocol step that generates the document to load additional points on the smart card. First the terminal selects an input to process and then performs the step consistent with the input and its internal state. The conditions for the different protocol steps should be mutually exclusive, so there is always exactly one step possible for a given input.

TERMINAL =
  if newConnection(*agent*) **then** ...
  else choose *port* with inputs(*agent*, *port*) ≠ [] **in**
          let *indoc* + *rest* = inputs(*agent*, *port*) **in**
              inputs(*agent*, *port*) := *rest*

```
if (   is-resdoc(indoc) ∧ port = 2 ∧ indoc.status = SW_OK
    ∧ last-inst(agent) = auth) then
   last-inst(agent) := load
   issued(agent) := issued(agent) + loadpoints(agent)
   outdoc := commanddoc(load, mkdoc(intdoc(loadpoints(agent)) +
                                    hashdoc(mkdoc(secretdoc(secret(agent)) +
                                                  indoc.doc +
                                                  intdoc(loadpoints(agent))))))
   outport := 2
else if (   is-resdoc(indoc) ∧ port = 2 ∧ indoc.status = SW_OK
         ∧ last-inst(agent) = load) then
...
```

In the ASM rule *agent* denotes the currently active agent, i.e. in this case a terminal. As mentioned above this part of the ASM rule for the terminal is responsible for the last step of the protocol for loading points. In the condition it is tested among other things, that the last document presumably received from the smart card reported a successful completion of the last protocol step of the card (*indoc*.status = SW_OK) and that the terminal is currently running the load protocol (last-inst(*agent*) = auth). The state of the terminal is modified by incrementing the number of issued points and by storing the instruction that will be sent to the card. The document that will be sent to the smart card is a commanddoc[4] with instruction 'load' and a data part that contains the number of points to load and the hash value of a list of documents containing a nonce to prevent replays, a common secret for authentication and the number of points to load. The nonce to prevent replays is *indoc*.doc, i.e. it is assumed that the message that is processed contains a nonce from the cardlet. In correct protocol runs the message that is processed by the terminal would have been created by the last step of the cardlet and would contain a new nonce generated by the cardlet to be used in this challenge and response mechanism.

8.3 The Cardlet

8.3.1 Internal State of the Cardlet

The smart card application has 4 data fields: *secret* and *challenge* are similar to the terminal fields. *value* contains the amount of money currently stored on the card and finally *newNonce* is true if the card program is currently in a run of the load protocol and will accept the loading of new money. So the ASM needs 2 additional functions to store the states of the cardlets (secret and challenge already exist): value: *agent* → *int* and newNonce: *agent* → *bool*.

8.3.2 ASM rule for the Cardlet

This part of the ASM determines the behavior of the smart card program. As the terminal the cardlet first selects an input to process and then processes it. As in the terminal section we will omit most of the ASM for the cardlet and just state two of the possible steps:

```
CARDLET =
  choose port with inputs(agent, port) ≠ [] in
     let indoc + rest = inputs(agent, port) in
        inputs(agent, port) := rest
        if (   is-comdoc(indoc) ∧ indoc.inst ≠ auth ∧ indoc.inst ≠ load
            ∧ indoc.inst ≠ select ∧ indoc.inst ≠ balance
            ∧ indoc.inst ≠ pay) then
            outport := 1
```

[4] commanddoc is an abbreviation for a list consisting of two documents, an integer document encoding an instruction and a second document containing some data. commanddocs represent the so-called Command-APDUs which are a central concept in smart card applications.

$outdoc$:= responsedoc($\perp$, SW_OK)

...

else if (is-comdoc($indoc$) $\wedge$ $indoc$.inst = load
$\wedge$ newNonce($agent$)
$\wedge$ hashdoc(mkdoc(secretdoc(secret($agent$)) +
noncedoc(challenge($agent$)) +
getpart($indoc$.doc, 1))) = getpart($indoc$.doc, 2)
$\wedge \neg$ (get-int(getpart($indoc$.doc, 1)) ≤ 0
$\vee$ value($agent$) +
get-int(getpart($indoc$.doc, 1)) > 32767) **then**
newNonce($agent$) := false
value($agent$) := value($agent$) + get-int(getpart($indoc$.doc,1))
$outport$:= 1
$outdoc$:= responsedoc($\perp$, SW_OK)

The first rule states that if the card program receives a document with an unexpected request it leaves its state unchanged and answers with an OK. The second rule describes the case of a successful attempt to load new money. The condition states that the state of the card program permits loading (newNonce($agent$)) and that the authenticity of the message can be verified (hashdoc(mkdoc(secretdoc(secret($agent$)) + noncedoc(challenge($agent$)) + getpart($indoc$.doc, 1))) = getpart($indoc$.doc, 2)). The card program then increments the money stored and answers with OK. The hash value contained in the processed document is exactly the same hash value that is produced by the terminal to load points (see terminal ASM rule in Section 8.2).

9 Proving Properties

Given out formal model of the application one can start to prove the desired security properties of the application. These can be typical properties of cryptographic protocols, such as secrecy or authenticity, but primarily we are interested in the more important application specific demands, such as the exclusion of fraud in an electronic business application. In the described electronic wallet, the application specific security goal is that never the sum of the money collected by point-of-sale terminals is larger than the sum of the money loaded onto the smart cards by the loading terminals, i.e. 'the user cannot get more than he has paid for'[5]. In order to prove this, one needs to know that the shared secret is never disclosed to the attacker, so one auxiliary property needed for the proof of the main security property is:

secret-unknown:
($\neg$ secret(THECARD) $\in$ *attacker-known*
$\wedge \forall$ $agent, port$.($\neg$ secret(THECARD) $\in$ inputs($agent, port$)))
$\rightarrow \langle\!|$E-WALLET$|\!\rangle$
($\neg$ secret(THECARD) $\in$ *attacker-known*
$\wedge \forall$ $agent, port$. ($\neg$ secret(THECARD) $\in$ inputs($agent, port$)))

E-WALLET was defined at the beginning of Section 8 and is the ASM rule of the application. $\langle\!|\cdot|\!\rangle$ is the strong diamond operator of DL. The meaning of a formula $\langle\!|\alpha|\!\rangle\ \varphi$ is that all runs of the program α terminate and the condition φ holds afterwards[6], therefore this theorem states that if initially the secret is not in the knowledge of the attacker and not contained in the unprocessed messages of any agent then this will still be the case after execution of E-WALLET. This means that the secrecy is invariant with respect to all possible steps of the application. With a couple of adequate simplifier rules this property can be proved automatically by the KIV system using symbolic execution.

[5]Note that 'the user gets exactly as much as he has paid for' is not provable. The attacker can always intercept the transfer of points and delete them.

[6]$\langle\!|\alpha|\!\rangle\ \varphi$ corresponds to wp(α, φ) in Dijkstra's wp-calculus.

9.1 Main Security Property

The security properties are generally application dependent. In our electronic wallet example the informal security property of the application provider is: 'It is impossible that more money is spent with the card than previously was loaded onto the card'. This property rules out fraud because it guarantees that only value points which were correctly loaded onto the card (in exchange for real money) can be spent.

To easily express this property we added two fields (*collected* and *issued*) to the state of the terminal which accumulate the sum of the points loaded onto the card and the points spent with the card. For the proof it is useful to consider the points currently on the card as well. The points on the card are stored in the field *value* of the card. With these three fields the security property can be expressed as collected(THETERM) + value(THECARD) $\leq$ issued(THETERM). What we must prove actually is that this inequality holds after all possible finite sequences of steps of the application. This can be expressed in Dynamic Logic with the following property:

no-fraud:

$$\begin{aligned} &\text{admissible}(\dots) \\ \rightarrow \forall\, n.\ \big(\ &\langle\!|\text{E-WALLET}^n|\!\rangle\ (\quad \textsf{collected}(\text{THETERM}) + \textsf{value}(\text{THECARD}) \\ &\qquad\qquad \leq \textsf{issued}(\text{THETERM}))\big) \end{aligned}$$

The formula states that if the application is started in an admissible state then the security property will not be violated in any state reachable after any number n of steps of E-WALLET.

This property is proven basically by showing that the inequality is invariant relative to a step of the application. As the invariant necessary to establish the property is quite complicated for a direct proof attempt, we used a different approach with an invariant that itself contains a program. The theorem we prove is:

INV-is-invariant:

$$\begin{aligned} &\text{INV}(\textit{attacker-known}, \textsf{user-known}, \textsf{secret}, \dots) \\ \rightarrow\ &\langle\!|\ \text{E-WALLET}|\!\rangle\ \text{INV}(\textit{attacker-known}, \textsf{user-known}, \textsf{secret}, \dots) \end{aligned}$$

It states that INV(*attacker-known*, user-known, secret,...) is invariant with respect to E-WALLET.

In the theorem stated above INV is the invariant we actually prove. INV is defined as follows:

INV-definition:

$$\begin{aligned} &\text{INV}(\textit{attacker-known},\textsf{user-known},\textsf{secret},\dots) \\ \leftrightarrow\quad &\text{Base-INV}(\textit{attacker-known},\textsf{user-known},\textsf{secret},\dots) \\ \wedge\ &\langle\!|\text{begin} \\ &\qquad \text{ATTACK-CARD}; \\ &\qquad \text{CARDLET}; \\ &\qquad \text{ATTACK-TERM}; \\ &\qquad \text{TERMINAL}; \\ &\quad \text{end}|\!\rangle\ (\textsf{collected}(\text{THETERM}) + \textsf{value}(\text{THECARD}) \leq \textsf{issued}(\text{THETERM})) \end{aligned}$$

In INV we use a basic trick of Dynamic Logic not possible e.g. in Hoare-calculus: A program formula is used as an invariant. This formula says informally that 'for every well-formed state even after the worst possible attack the desired property holds'. Using such invariants with ASM rule applications is a great advantage because it clearly simplifies the state invariant. A related idea is used in [Sch01] to express that two states are related through a coupling invariant if two finite sequences of state transitions starting in those states lead to two similar successor states. A dual idea used in [DW03][Sch05] is to use predecessor states. The program part of the invariant is contained in the strong diamond. It consists of four parts: The first part performs an attack on the card if possible (i.e. if the attacker can produce the document necessary to trick the card into loading points), the second part is a step of the smart card. In this step the smart card will process the attack document generated in step number one (if the attacker could create it). Steps three and four work similarly and realize the attack on the terminal. ATTACK-CARD and

ATTACK-TERM are special instances of ATTACKER-SEND. CARDLET and TERMINAL are the normal ASM rules for these agent types. The advantage of such an invariant is that the main property must not be established for all between states, instead we can focus on such states in which certain operations are completed. Interestingly establishing the invariant after an attacker step is trivial because the invariant especially says that attacker steps do no harm.

The theorem itself is proved by symbolic execution of E-WALLET in the DL-calculus of KIV. The symbolic execution results in some open goals, one for each step of E-WALLET. These goals are then closed by showing that INV still holds.

Given a well-formed initial state of the ASM the theorem **INV-is-invariant** guarantees that in any state, reachable by a finite sequence of steps of the application, the well-formedness condition holds and that even if the attacker performs all attacks he can perform, the security condition still holds.

As mentioned above every state reachable from the initial state is well-formed. We also proved that in every reachable state it is true that after completing all possible attacks, the security property still holds. Having proved these facts it is easy to prove that the inequality of the security property actually always holds using a theorem (**less-equal-before-if-after**) that states that the inequality propagates backwards over the program part of the invariant:

less-equal-before-if-after:

$$
\begin{array}{l}
\quad \text{Base-INV}(\textit{attacker-known},\mathsf{user\text{-}known},\mathsf{secret},\ldots) \\
\quad \wedge\ \textit{collected} = \mathsf{collected}(\text{THETERM}) \\
\quad \wedge\ \textit{value} = \mathsf{value}(\text{THECARD}) \\
\quad \wedge\ \textit{issued} = \mathsf{issued}(\text{THETERM}) \\
\rightarrow \Big(\quad (\langle\!| \text{begin} \\
\qquad\qquad \text{ATTACK-CARD; CARDLET; ATTACK-TERM; TERMINAL;} \\
\qquad\quad \text{end} |\!\rangle\ \mathsf{collected}(\text{THETERM}) + \mathsf{value}(\text{THECARD}) \leq \mathsf{issued}(\text{THETERM})) \\
\quad\ \rightarrow \textit{collected} + \textit{value} \leq \textit{issued} \Big)
\end{array}
$$

Proving the theorem **less-equal-before-if-after** is simple because it is obvious that the inequality propagates backwards over the attacks, since an attack could modify the agents' states only in an unfavorable way. We have proven all the security properties with KIV which offers good proof support through mature heuristics and so a high degree of automation was reached.

Note that the property we have proven is not model-checkable, since the model of the application does not have a finite state space. The basic data types (integer, list, ...) as well as the knowledge of the attacker and the set of documents that can be produced by the attacker are all unbounded. The possibility to verify applications that are not reduced to a finite state space is the main advantage of interactive verification.

9.2 Protocols and their Expected Functionalities

Besides the security property we also proved that the protocols for the different functions serve their purpose, e.g. it is possible to increment the points stored on the card using the load protocol. Ensuring that the protocols are up to the job is important, otherwise one could design some protocols and verify that they guarantee the desired security properties, but in fact the protocols do not realize the functionality of the application, so they are unsuitable. For example the functioning of the load protocol can be expressed by the following theorem:

load-works:

$$\exists\, n.\langle \text{E-WALLET}^n \rangle\ (\mathsf{value}(\text{THECARD}) > 0)$$

Assuming a well-formed initial state (where the points on the card are 0) this theorem states that there exists a finite sequence of steps of the application E-WALLET that leads to a state in which there are points stored on the card. $\langle\alpha\rangle\ \varphi$ means there exists a terminating run of α such that φ holds afterwards[7].

[7] $\langle\alpha\rangle\ \varphi$ corresponds to $\neg$ wlp(α, $\neg\ \varphi$) in Dijkstra's wp-calculus.

9.3 Proving with KIV

The representation of the ASM as DL program in the KIV system is a good basis for verification. Proving of theorems is supported by mature heuristics and a high degree of automation is reached. The integration of a program formula in the invariant simplifies the invariant because it allows us to focus on states in which all possible attacks were performed. This is a simplification because we do not have to put complicated conditions for potential states with incomplete protocol runs into the invariant. Interestingly establishing the invariant after an attacker step is trivial because the invariant especially says that attacker steps do no harm.

10 Related Work

Formal methods that analyze security protocols on an abstract level are in use for quite some time. A lot of different approaches have been proposed.

Rather different from the approach presented here is the usage of specialized logics, such as the BAN logic of belief [BAN89]. In [ABV01] Accorsi, Basin and Viganò describe an approach that combines security logics with inductive methods.

Many approaches to the verification of protocols are model-checking based. In [Low96] an analysis of the Needham-Schroeder Public-Key protocol is described. The protocol is described in CSP and the Failure Divergences Refinement Checker (FDR) [Ros94] is used to check the protocol description. An enhancement to the usage of general-purpose model-checker is the usage of model-checkers specifically designed for reasoning about security protocols, e.g. OFMC developed by Basin et al. [BMV03]. The model checking based approaches usually focus on standard properties like secrecy or authenticity while our interactive approach can deal with arbitrary properties.

The CSP approach is not limited to model-checking anymore. [RSG+01] describes, among other things, an embedding of the CSP trace semantics in PVS. It also describes *rank functions* as a mean to simplify reasoning about the availability of certain messages. Such a technique could be combined with our model of the possible runs of the application, but so far reasoning based on [Pau98] was sufficient. We just formulate theorems stating the unavailability of a certain message or the conditions under which the message is available to the attacker.

Strand spaces [FHG99] are also an approach to abstract specification of cryptographic protocols. The model is quite elegant and easy to understand. The attacker model corresponds to a Dolev-Yao attacker and does not use a detailed model of different communication channels as we do. The method also focuses on standard goals for cryptographic protocols and therefore the main concepts of the model are production and consumption of messages. Application specific goals, like the one described in this paper, and therefore the treatment of application specific data are not modeled.

Other specialized tools for security protocol verification are Interrogator by Millen [MCF87] and the NRL protocol analyzer by Meadows [Mea96].

Paulson uses Isabelle to verify security protocols [Pau98]. This approach is quite successful and can cope with large protocols such as SET [Pau01]. Our representation of documents and attacker knowledge is inspired by this approach: we use documents and functions very similar to analz and synth in [Pau98].

Our approach differs in how system runs are described. First, [Pau98] describes protocol runs declaratively using an inductive definition of the set of possible traces while our approach describes runs using iterative application of (operationally specified) ASM rules. There is a close connection, since iterative rule application can be defined using inductive relations (for while loops and recursion this has e.g. been done in [Nip02]). We prefer an operational definition using ASM rules where the inductive nature is encoded in the semantics, since it offers the full possibilities of structured programs which can be exploited for proof automation using symbolic execution [RSSB98]. A second difference is that we use an explicit representation of the agent states, while [Pau98] instead defines an explicit system trace consisting of all messages that have been sent. The difference is that of state based vs. event based representation: recovering the current points

loaded onto a card could be done in [Pau98] by accumulating the points of all successful load and pay messages and adding/subtracting them.

A middle ground between [Pau98] and our approach is also possible as the case study [BR98] on the Needham-Schroeder protocol shows: there an ASM is used to formalize runs, but a global execution trace is used instead of agent states. [BR97] analyzes the Kerberos protocol using distributed ASMs with an agent state similar to ours. Proofs have been done manually though in both cases. Also between our and Paulson's inductive approach is [Bel01]. Bella extends the inductive approach to deal with smart card applications. He allows inputs to and outputs from smart cards which the attacker cannot observe. This reflects that the Dolev-Yao model is inappropriate in some cases. What remains of Paulson's original approach is that the state of the agents is not modeled explicitely but instead is derived from the trace.

Finally an alternative to represent attacker knowledge and protocol runs is given in [RRS03], which is also based on ASMs. Protocol steps are given by *patterns*. This promises to provide an elegant way to model-check authentication protocols.

11 Conclusion

We presented an approach to specify security protocols in smart card applications to allow their formal analysis and an example illustrating our approach.

Summarizing, our approach offers a detailed model of the communication to support an application specific attacker which is important for smart card applications. The verification framework is generic and can be used for different applications. Our proof strategy, proving state invariants by symbolic execution is supported by the KIV system with a high degree of automation. The usage of DL programs in the invariants simplifies the invariants by focusing on the interesting states.

The approach itself is suitable for the verification of security properties of different kinds of communication protocols. Our approach can be used to model various protocol scenarios, such as smart card applications as well as distributed systems, communicating over the Internet or over other insecure networks like Wireless LAN, Bluetooth or GSM as well as common cryptographic protocols. Besides the communication protocols of the presented electronic wallet application we already formalized other cryptographic protocols, e.g. the Needham-Schroeder Public-Key authentication protocol and other m-commerce applications [GHRS06].

A main topic for future research is a suitable refinement technique for security protocols. The existing ASM refinement theory seems to be a good starting point for this attempt. [Bör03] describes the general method for ASM refinements. [Sch01] describes a verification method for ASM refinement based on forward simulation and Dynamic Logic. In [GHRS06] such a refinement is defined and verified for an electronic ticketing application.

The presented state-based approach for modeling security protocols is very useful for such a refinement concept because the use of encapsulation in an object-oriented language is a natural approach for implementing state-based systems. The Java programs will be integrated into the ASM model by translating the abstract state of the agents and the content of the communication ports into Java objects. With encapsulation this can be simply done by storing the state in the fields of the objects. We then replace the rules for the agents (except the users and the attacker) by Java method calls which then will use those objects. The KIV system already defines a calculus for the verification of JavaCard programs, supporting all language features of Java except threading, which is described in [Ste04].

Our goal is to have a verification approach that starts with proofs of security properties in an abstract specification and continues all the way down to the verification of correctness of an implementation of the agents in real Java code. Besides the development of this refinement concept in our future work we will also apply our modeling technique to more case studies.

References

[ABV01] Rafael Accorsi, David Basin, and Luca Viganò. Towards an awareness-based semantics for security protocol analysis. In Jean Goubault-Larrecq, editor, *Post-CAV Workshop on Logical Aspects of Cryptographic Protocol Verification*. Elsevier Science Publishers, Amsterdam, 2001.

[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. Technical report, SRC Research Report 39, 1989.

[Bel01] G. Bella. Mechanising a Protocol for Smart Cards. In *Proc. of e-Smart 2001, international conference on research in smart cards*, volume 2140 of *LNCS*. Springer-Verlag, 2001.

[BFGS05] Egon Börger, Nicu G. Fruja, Vincenzo Gervasi, and Robert F. Stärk. A high-level modular definition of the semantics of C#. *Theor. Comput. Sci.*, 336(2-3):235–284, 2005.

[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.

[BR97] G. Bella and E. Riccobene. Formal Analysis of the Kerberos Authentication System. *Journal of Universal Computer Science*, 3(12), 1997.

[BR98] G. Bella and E. Riccobene. A Realistic Environment for Crypto-Protocol Aalyses by ASMs. In U. Glasser and P. Schmitt, editors, *Proc. 5th Int. Workshop on Abstract State Machines*. Magdeburg University, 1998.

[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.

[BS97] E. Börger and P. Schmitt. A Description of the Tableau Method Using Abstract State Machines. *J. Logic and Computation*, 7(5):661–683, 1997.

[BS98] E. Börger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.

[BS00] Egon Börger and Wolfram Schulte. Modular design for the Java virtual machine architecture. pages 297–346, 2000.

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

[DW03] J. Derrick and H. Wehrheim. Using Coupled Simulations in Non-atomic Refinement. In D. Bert, J. Bowen, S. King, and M. Walden, editors, *ZB 2003: Formal Specification and Development in Z and B*, volume 2651, pages 127–147. Springer LNCS, 2003.

[DY81] D. Dolev and A. C. Yao. On the security of public key protocols. In *Proc. 22th IEEE Symposium on Foundations of Computer Science*, pages 350–357. IEEE, 1981.

[FHG99] F. Fábrega, J. Herzog, and J. Guttman. Strand Spaces: Proving Security Protocols Correct. *Journal of Computer Security*, 7:191–230, 1999.

[FKK96] Alan O. Freier, Philip Karlton, and Paul C. Kocher. *The SSL Protocol Version 3.0*. Netscape Communications, November 1996. URL: `http://wp.netscape.com/eng/ssl3/`.

[GHRS06] H. Grandy, D. Haneberg, W. Reif, and K. Stenzel. Developing Provably Secure M-Commerce Applications. LNCS. Springer, 2006. Accepted for ETRICS 2006. To appear.

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

[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.

[HRS04] Dominik Haneberg, Wolfgang Reif, and Kurt Stenzel. A Construction Kit for Modeling the Security of M-Commerce Applications. In Manuel Núñez, editor, *Proceedings of ITM/EPEW/TheFormEMC*, Springer LNCS 3236, 2004.

[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.

[Mas97] Mastercard & Visa. SET Formal Protocol Definition, May 1997.

[MCF87] Jonathan K. Millen, Sidney C. Clark, and Sheryl B. Freedman. The Interrogator: Protocol Security Analysis. *IEEE Trans. Software Eng.*, 13(2):274–288, 1987.

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

[Nip02] Tobias Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In J. Bradfield, editor, *Computer Science Logic (CSL 2002)*, volume 2471 of *LNCS*, pages 103–119. Springer, 2002.

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

[Pau01] L. C. Paulson. Verifying the SET Protocol. In R. Gore, A. Leitsch, and T. Nipkow, editors, *IJCAR 2001: International Joint Conference on Automated Reasoning*, Siena, Italy, 2001. Springer LNCS 2083.

[Ros94] A. W. Roscoe. Model-checking CSP. pages 353–378, 1994.

[RRS03] D. Rosenzweig, D. Runje, and N. Slani. Privacy, Abstract Encryption and Protocols: An ASM Model - Part I. In *Abstract State Machines 2003. 10th International Workshop*, volume 2589 of *LNCS*. Springer, 2003.

[RSG+01] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach*. Addison-Wesley, 2001.

[RSSB98] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*, volume II: Systems and Implementation Techniques, chapter 1: Interactive Theorem Proving, pages 13 – 39. Kluwer Academic Publishers, Dordrecht, 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.

[SGHR06] G. Schellhorn, H. Grandy, D. Haneberg, and W. Reif. The Mondex Challenge: Machine Checked Proofs for an Electronic Purse. Technical Report 2006-2, Universität Augsburg, 2006. URL: `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/` (Submitted to Formal Methods 2006).

[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.