Verifying Security Protocols: 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

Abstract. In this paper we present a modeling technique for security protocols using Abstract State Machines [BS03,Gur95] (ASMs). We describe how we model the different agents in the scenario, the attacker and the communication between them[1].

1 Introduction

Communication is an integral part of modern distributed systems. In security critical distributed applications, such as e.g. e-commerce applications, the security of the communication is crucial. For such applications communication is generally secured using security protocols. These protocols try to guarantee the 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 the standard protocols mentioned before are not sufficient. An application can have very specific security demands that are not fulfilled by a standard protocol, 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. in applications using smart cards, because of the hardware restrictions of the smart card, standard protocols cannot be used.

Unfortunately such protocols are 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. The use of formal methods is the only chance to guarantee the desired properties 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,Ros95], interactive theorem proving [Pau98] or specialized logics of belief [BAN89].

[1] This work is part of the Go!Card project which is sponsored by the Deutsche Forschungsgemeinschaft (German Research Foundation) under grant RE 828/4-2.

In this paper we present a particular approach to verify security protocols. We use an Abstract State Machine (ASM) to model the agents in a protocol, the attacker and the communication between the agents. Data is modeled using algebraic specifications, the ASM is described as rules manipulating the state of the agents. Proofs are generally inductive proofs; proving of properties is done with the KIV system [BRS+00], our interactive theorem prover.

Section 2 describes the messages used for communication between the agents, Section 3 describes how the state of the agents is modeled, in Section 4 our attacker model in introduced, Section 5 offers details on the ASM we use as description of the behavior of the agents. In Section 6 our concept is demonstrated using a small example and Section 7 presents ideas of a refinement concept. Finally Section 9 concludes.

2 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*[2] or *integers* or compound documents such as lists of documents, hash-values or encrypted documents.

The algebraic specification for the documents contains constructors to create a document from basic data types, e.g. a document that contains an integer. Besides these basic constructors there are also constructors for compound 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].

We assume 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:

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

$$\begin{array}{l}\text{decrypt-keypair-fail:}\\ \neg\ \text{is\_keypair(key,key}_0) \wedge (\text{is\_pubkey(key)} \vee \text{is\_privkey(key)}) \rightarrow\\ \text{is\_intdoc(decrypt(key}_0\text{,encdoc(key,doc,n)))}\end{array} \tag{2}$$

Axiom 1 states that decryption with a key that forms a asymmetric key pair with the encryption key produces the correct plaintext as result, axiom 2 states that decryption with a key that does not form a key pair with the encryption key leads to an unspecified integer document.

[2] 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.

3 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 quite useful for describing certain security properties (as described in [HRS02]).

In this context agents are the different participants of an application. An agent has an unique type depending on what kind of participant he is. Given n types of participants (including the attacker) each agent is of one of the agent types at_1,...,at_n. 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 to load additional money on the card, as well as the attacker and the owners of the smart cards. For the example we have agent = 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 the current state of this field of the agent. This means that an agent of type at_i with fields $f_1, \ldots, f_n$ of sorts $s_1, \ldots, s_n$ is represented by dynamic functions $f_1, \ldots, f_n$ with signatures $\mathrm{at}_i \to s_{1 \ldots n}$. Suppose we have a terminal with two fields, one for its secret key and one for a nonce. For this agent we have:

secret_key: terminal → key;
session_nonce: terminal → nonce;

Using algebraic specifications, we could instead define a new 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 of properties.

In addition to the application-specific fields some additional data is stored for each agent. Most important is the set of the current connections of an agent (using the dynamic function conns : agent → connection-set). In this set all currently established connections between this agent and the other agents are stored. Furthermore we keep a list of unprocessed messages of 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 channels we have inputs : agent → channel → document-list as the signature of this function.

4 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. Usually an attacker similar to the Dolev-Yao attacker [DY81] is used. 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 this case a reduced attacker model is more realistic. This can be done easily in our model. For example in certain smart card applications it is not realistic to assume that the attacker would invest the expense to eavesdrop on all communication. It is easy for an attacker to program a faked smart card and use it in a terminal but it is very complicated to eavesdrop on the communication between the terminal and a genuine smart card, so in this case the attacker receives some of the exchanged messages but not all. As a result of such a reduced attacker model, the proofs become more easy and efficient.

The attacker model consists of three distinct parts. The first describes how the attacker can decompose messages and build new ones, the second part describes the attackers abilities to eavesdrop or manipulate the communication. These descriptions are given as first-order formulas. These formulas are later used in the ASM rule that describes the attacker's actions. This ASM rule is the third part of the attacker description.

4.1 Treatment of Messages

This part of the attacker describes the handling of documents. First we describe how the attacker can extract new information from documents he received and subsequently the production of new documents. The approach used is similar to Paulson's work in [Pau98].

Analyzing Messages The attacker decomposes all messages as far as possible. By this decomposition the knowledge of the attacker (K), i.e. all the documents the attacker knows, grows. If a new document *doc* becomes available for the attacker, either by decomposing another document or by eavesdropping on a 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 K and nothing further is done.
2. If a key-document is added to K and if there exist documents in K that can be decrypted with this key, the decrypted documents are added to K as well.
3. If a list of documents is added to K then all documents in the list are added as well.
4. If an encrypted document is added to K and if the appropriate decryption key is in K the plaintext is added to K as well.
5. If a signature or a hash document is added to K nothing further is done.

This decomposition is repeated until a fixpoint is reached. It is important to note that acquiring one additional document by eavesdropping may substantially enlarge K, because it may allow the additional decryption of a lot of documents in K. In the following adding a new document *doc* to the attacker's knowledge K is written down as $K \int\!\!+\ doc$ where $\cdot \int\!\!+ \cdot$ is a function that adds *doc* to K and calculates the fixpoint under the decomposition.

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 that determines if an attacker with a given knowledge can produce a given document or not. As the axioms for this predicate are somewhat lengthy we will omit them and just give an informal description.

- The attacker can produce arbitrary integer documents.
- The attacker can produce every document that is contained in his knowledge K.
- The attacker can produce a list of documents if he can produce all of its elements.
- The attacker can produce a hash-value if he can produce the document that is hashed.
- The attacker can produce a signature or an encrypted document if he can produce the plaintext and the key is in his knowledge K.

This means that the attacker cannot guess random numbers (nonces) and keys. But he may use them if he learned them by eavesdropping.

Together this describes the most indeterministic and powerful attacker as the attacker does all decompositions possible and can produce all documents that can be built with his knowledge. This formalization is very similar to Paulson's functions *analz* and *synth* in [Pau98].

4.2 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 channels and output channels for communication. A connection describes which output channel of one agent is linked to which input channel of another agent. For each input channel a list of messages that were sent to this channel 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 is true if the attacker can eavesdrop on a communication connection. If the attacker can manipulate data transmitted over this connection the predicate can-write is true for this connection. 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.

5 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 execution is modeled. The ASM describing the 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 be duplicate free, etc.
2. An ASM rule that performs one step in the protocol model.

In an ASM step first of all the ASM chooses indeterministically which agent performs the next step. In most cases more than one agent can perform an action and one is chosen indeterministically. From the viewpoint of proving this ensures that all possibilities are considered. Let $a_1, \ldots, a_n$ be the different types of agents in an application (including the attacker and the user). We write agent-type($agent$) $= a_i$ if $agent$ is of type a_i. Let ready($agent$) be a predicate that determines if an agent is ready to perform a step. To select the agent the ASM performs the following choose:

APPLICATION =
    choose *agent* **with** ready(*agent*) **in** $R(agent)$

where R 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 sequence of **if**-statements:

$R(agent)$ =
    if agent-type(*agent*)=at_1 **then** R_1 ...
    else if agent-type(*agent*)=at_{n-1} **then** R_{n-1} **else** R_n

Most agents work deterministically but the attacker and the user can do different things (e.g. sending messages or establishing new connections), so if the attacker or the user was selected for the next step, they perform another indeterministic choose to decide which of their possible actions they will perform.

5.1 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 give the ASM rule for eavesdropping:

ATTACKER-ADD-KNOWLEDGE =
  choose $agent, channel$ **with** attacker-can-read(conns, $agent$, $channel$) **in**
    attacker-known := attacker-known $\int\!+$ inputs($agent$,$channel$) **seq**
    if agent-type($agent$) = attacker **then** inputs($agent$,$channel$) := []

For eavesdropping the attacker first selects an agent and a channel he can eavesdrop on. We can distinguish between different communication channels for every agent and it is possible that the attacker may eavesdrop on some channels of an agent but not on the others. This is specified by the predicate can-read mentioned earlier. This predicate is one part of the axiom for attacker-can-read(). After selecting the channel, the attacker adds all the documents currently in transfer over this channel to his knowledge and calculates the closure of his knowledge under the decomposition operations described in section 4.1.

The following ASM rule represents the manipulation of messages:

ATTACKER-SEND =
  choose $docs$ **with** attacker-known $\rhd$ $docs$ **in**
    choose $agent, channel$ **with** attacker-can-send(conns, $agent$, $channel$)
      in inputs($agent$, $channel$):= $docs$

This rule chooses a list of documents ($docs$) that can be produced by the attacker (stated by attacker-known $\rhd$ $docs$) and then chooses an existing agent and a channel of this agent which he can modify and replaces the list of messages on this channel with $docs$. If the attacker can modify messages on a channel is determined by attacker-can-send() which uses can-write as part of its specification.

5.2 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_i consists of a condition $C_{\mathrm{at}_i,s}$ that is true if this step can be performed and a rule $R_{\mathrm{at}_i,s}$ 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 outdoc($agent$). 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 outchannel($agent$). Finally SEND moves the generated document to the input channel of the receiver. The condition $C_{a,s}$ takes into account the current state of the agent and the message it is currently processing.

Assume the agent of type at_i has the possible steps $s_1, \ldots, s_n$, then the rule for this agent has the following structure:

$R_{\mathrm{at}_i}(agent)$ =
  if $C_{\mathrm{at}_i,s_1}(agent)$ **then** $R_{\mathrm{at}_i,s_1}(agent)$
  ...
  else if $C_{\mathrm{at}_i,s_n}(agent)$ **then** $R_{\mathrm{at}_i,s_n}(agent)$
  SEND

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

$R_{\mathrm{at}_i,s_j}(agent)$ =
  $f_1(agent) := \ldots$
  $\vdots$
  $f_n(agent) := \ldots$
  outdoc(*agent*):=...
  outchannel(*agent*):=...

After the agent performed its step and produced a document that must be transfered, the ASM selects the connection that belongs to outchannel(*agent*) and puts the generated document at the end of the list of unprocessed documents of the receivers (*conn*.agent_remote) input channel (*conn*.ch_remote) belonging to the selected connection:

SEND =
  if ∃conn(outchannel(*agent*),conns(*agent*)) **then**
    choose *conn* **with** *conn* ∈ conns(*agent*)
      ∧ *conn*.ch_local = outchannel(*agent*) **in**
      inputs(*conn*.agent_remote, *conn*.ch_remote) :=
        inputs(*conn*.agent_remote, *conn*.ch_remote) + outdoc(*agent*)

The test **if** ∃conn(outchannel(*agent*),conns(*agent*)) ensures that a connection currently exists for the designated output channel. If no such connection exists the message is lost.

6 A Brief 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 canteen or with copying machines. 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.

As agents in this application we have the smart card with the electronic wallet application, a card terminal (that is used for loading money on the card and for the payment) the owner of the card and the attacker.

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 functions important for an electronic wallet, e.g. loading money on the smart card, paying for goods and checking the balance.

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 used 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.

6.1 Internal State of the Smart Card Application

The smart card application has 4 data fields: *secret* that contains the secret shared with the terminal, *value* that contains the amount of money currently stored on the card, *challenge* that contains the nonce currently in use (if any) and finally *newNonce* that states if the card program will accept the loading of new money. So the ASM needs 4 functions to store the states of the cardlets: secret: cardlet → secret, value: cardlet → int, challenge: cardlet → nonce, newNonce: cardlet → bool.

6.2 The Attacker

As described in Section 4 the attacker model consists of different parts. The treatment of messages is independent of the concrete application, so we will not discuss it any further. The attacker's abilities to manipulate the communication were already mentioned above. The attacker can read and manipulate the communication between the smart card and the terminal, so 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 5 we stated that the attacker has to choose between his possible actions. This is done in the following ASM rule:

ATTACKER ($params_{in}$) =
choose *step* **with**
($step \in$ {send, read, connect, disconnect})
$\wedge$ ($step$ = send → attacker-can-send(conns))
$\wedge$ ($step$ = read → attacker-can-read(conns))
$\wedge$ ($step$ = connect → connect-possible($agent$, conns))

$\wedge$ (*step* = disconnect $\rightarrow$ disconnect-possible(*agent*, conns))) **in**
if *step* = send **then** ATTACKER-SEND($params_{send}$)
else if *step* = read **then** ATTACKER-ADD-KNOWLEDGE($params_{add}$)
else if *step* = connect **then** CONNECT($params_{connect}$)
else if *step* = disconnect **then** DISCONNECT($params_{disconnect}$)

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(conns) which is:

attacker-can-send(conns) $\leftrightarrow$
$\exists$ *agent*, *channel*.
$\exists$agent(*agent*)
$\wedge$ $\neg$ agent-type(*agent*) = attacker
$\wedge$ $\exists$ *conn*. (*conn* $\in$ conns(*agent*) $\wedge$ *conn*.local-port = *channel*
$\wedge$ ($\neg$ $\exists$ *conn*. (*conn* $\in$ conns(attacker)
$\wedge$ *conn*.rem-act = *agent*
$\wedge$ *conn*.rem-port = *channel*))
$\rightarrow$ can-write(*agent*, *channel*, *conn*.rem-act, *conn*.rem-port))

ATTACKER-SEND and ATTACKER-ADD-KNOWLEDGE were described in Section 5.1, CONNECT and DISCONNECT are merely technical and will be omitted.

6.3 The Cardlet

This part of the ASM determines the behavior of the smart card program. First the cardlet selects an input to process and then performs the step consistent with the input and the internal state. We will omit most of the rules and just state two:

CARDLET($parmas_{in}$) =
choose *channel* **with** inputs(*agent*, *channel*) $\neq \perp$ **in**
let *indoc* = inputs(*agent*, *channel*) **in**
inputs(*agent*, *channel*) := $\perp$
if (is_comdoc(*indoc*) $\wedge$ *indoc*.inst $\neq$ auth $\wedge$ *indoc*.inst $\neq$ load
$\wedge$ *indoc*.inst $\neq$ select $\wedge$ *indoc*.inst $\neq$ balance
$\wedge$ *indoc*.inst $\neq$ pay) **then**
outchannel(*agent*) := 1
outdoc(*agent*) := responsedoc($\perp$, SW_OK)
...
else if (is_comdoc(*indoc*) $\wedge$ *indoc*.inst = load
$\wedge$ cardlet-newNonce(*agent*)
$\wedge$ hashdoc(mkdoc(secretdoc(cardlet-secret(*agent*)) +
noncedoc(cardlet-challenge(*agent*)) +

$$
\begin{array}{l}
\qquad\qquad\qquad\text{getp}(\textit{indoc}.\text{doc, 1)'}))=\text{getp}(\textit{indoc}.\text{doc, 2)}\\
\wedge\ \neg\ (\quad \text{get\_int}(\text{getp}(\textit{indoc}.\text{doc, 1}))\leq 0\\
\qquad\ \vee\ \text{cardlet-value}(\textit{agent})\ +\\
\qquad\quad\ \text{get\_int}(\text{getp}(\textit{indoc}.\text{doc, 1}))>32767)\ \textbf{then}\\
\text{cardlet-newNonce}(\textit{agent}) := \text{false}\\
\text{cardlet-value}(\textit{agent}) := \text{cardlet-value}(\textit{agent})\ +\\
\qquad\qquad\qquad\qquad\quad \text{get\_int}(\text{getp}(\textit{indoc}.\text{doc,1}))\\
\text{outchannel}(\textit{agent}) := 1\\
\text{outdoc}(\textit{agent}) := \text{responsedoc}(\perp,\ \text{SW\_OK})
\end{array}
$$

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 and 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 and that the authenticity of the message can be verified. The card program then increments the money stored and answers with OK.

In total the ASM part for the smart card has 11 rules and conditions and the part for the terminal 17. In total the specification of the ASM (without the algebraic specification it is built upon) has 425 lines.

6.4 Proving of Properties

Given this formal model of the model one can start proving the desired security properties of the application. These can be typical properties of cryptographic protocols, such as secrecy or authenticity but also 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. In order to prove this, one needs to know that the shared secret is never disclosed to the attacker, so one of the properties needed for the proof of the main security property is:

$$
\begin{array}{l}
\ \ (\quad \neg\ \text{secret(THECARD)} \in \text{attacker-known}\\
\ \ \ \wedge\ \forall\ \mathit{agent}, \mathit{channel}.(\neg\ \text{secret(THECARD)} \in \text{inputs}(\mathit{agent}, \mathit{channel})))\\
\to [\ \text{APPLICATION}(\mathit{params})\ ]\\
\qquad\ (\quad \neg\ \text{secret(THECARD)} \in \text{attacker-known}\\
\qquad\ \ \wedge\ \forall\ \mathit{agent}, \mathit{channel}.\ (\neg\ \text{secret(THECARD)} \in \text{inputs}(\mathit{agent}, \mathit{channel})))
\end{array}
$$

This formula 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 after execution of the ASM rule for the application the secret will not be in the attacker's knowledge or the unprocessed messages of any agent. This means that the secrecy is invariant with respect to the protocol steps of the electronic wallet example. This property can be proved automatically by the KIV system using symbolic execution.

7 Refinement to Real Implementations

Our current research is motivated by the attempt to refine ASM protocol specifications to Java programs.

The electronic wallet case study as well as other programs for larger smart card based Java applications are already implemented. This includes an E-Ticketing application for smart cards, with which the user can buy train tickets over the Internet. This application is described e.g. in [Han02].

Our plan is to include a suitable refinement theory in our approach in order to gain a verification technique for Java applications that includes abstract modeling of the protocol, the proof of security and the proof of correctness of the real implementation.

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.

The Java programs will be integrated into the ASM model by translating the abstract state of the agents and the content of the communication channels into Java objects and replacing the rules for the agents (except the users and the attacker) by Java method calls which then will use those objects.

The KIV systems defines a calculus for the verification of JavaCard programs, supporting all language features of Java except threading, which is described in [Ste04].

With the Java calculus mentioned above, we have already verified correctness assertions for Java programs that deal with byte arrays. So, the next step is a suitable refinement method for protocol specifications given in the way that is described in this paper. Of course, many problems arise in this context. For example, when using abstract documents for the messages, one can 'see' the structure of the document by looking at the constructors. For example, a hash document can be distinguished from an encrypted document. When we refine such a protocol to a Java program using byte arrays, such differences cannot be seen. Protocol specifications that make use of such structural informations, are therefore not a realistic scenario and should not be used.

Another problem is the assumption of perfect cryptography used in the abstract specification. The assumption states that e.g. a hash function or a encryption function is injective. For abstract security considerations, this approach is suitable and common practice (see e.g. [Pau98], [BMV03]). Of course, when looking at real Java code, a hash function cannot be injective because its result is a byte array of fixed length. The same problem arises for encryption operations. Since common encryption operations, like RSA, take a key and a byte array and return an encrypted byte array with the same size as the key, this function cannot be injective, too. A solution would be to describe a set of 'allowed' messages for the abstract protocol and to claim that the cryptography operations work as desired only on this set of messages. If that set is suitable, it reflects the intuition of e.g. the impossibility of guessing a hash value while at the same time it adheres to the restrictions that real implementations dictate. On the one hand that set should not be too small, because otherwise messages that are allowed in the real world would not be considered. On the other hand, it should not be too large, because then our assumption of perfect cryptography would be unrealistic. Finding the right restrictions for the used messages (e.g. allowing only documents up to a fixed nesting depth, forbidding double encryption and so on) is part of our future work.

8 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 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.

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]. Protocols are described declaratively using an inductive definition of the set of possible traces. This approach is quite successful and can cope with large protocols such as SET [Pau01]. This approach is closest to ours, but it does not have an explicit state for the agents. We think our approach is one step closer to real implementations because of the explicit state, which is modeled using dynamic functions, and our ASM rules, which are written in a component based way.

In [RRS03] an ASM based description of protocols is given. The paper introduces an executable formal model of abstract encryption using the AsmL specification language. It supports reasoning about protocol secrecy but lacks verification support.

9 Conclusion

We presented an approach to specify security protocols to allow their formal analysis.

Our modeling technique is based on an ASM that describes the dynamic behavior of the agents. ASMs are well supported in the KIV system and their verification is feasible within the system.

The model uses dynamic functions to determine the state of the agents during the protocol.

The approach itself is suitable for the verification of security for communication protocols. Furthermore, the technique allows the modeling of 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. Due to this, our future work will include the application of our modeling technique to more case studies.

A main topic for future research is to define a suitable refinement technique for security protocols. The 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 with real Java code.

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.

[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, pages 363–366. 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.

[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.

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

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

[Han02] D. Haneberg. electronic ticketing — a smartcard application case-study. Technical Report 16, Institut für Informatik, Universität Augsburg, http://www.informatik.uni-augsburg.de/forschung/techBerichte/reports/2002-16.ps.gz, 2002.

[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] Gavin 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, Berlin Germany, 1996.

[Mas97] Mastercard & Visa. SET Formal Protocol Definition, May 1997. Available electronically at `http://www.setco.org/set_specifications.html`.

[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.

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

[Pau01] Lawrence C. Paulson. SET Cardholder Registration: The Secrecy Proofs. In *IJCAR*, pages 5–12, 2001.

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

[Ros95] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *CSFW '95: Proceedings of the The Eighth IEEE Computer Security Foundations Workshop (CSFW '95)*, page 98. IEEE Computer Society, 1995.

[RRS03] Dean Rosenzweig, Davor Runje, and Neva Slani. Privacy, Abstract Encryption and Protocols: An ASM Model - Part I. In *Abstract State Machines*, pages 372–390, 2003.

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

[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.