Generic Formal Model of Security Policy and its Informal Interpretation

Target of Evaluation: ICC embedded software for Signature Creation conforming with German SigG, SigV and DIN V 66391-1

Version 1.1
September 12, 2000

Authors:
Bruno Langenstein
Roland Vogt
**Developed by**
Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI GmbH)  
[German Research Center for Artificial Intelligence]
Stuhlsatzenhausweg 3  
66123 Saarbrücken  
GERMANY

**Contact:**  
Roland Vogt  
phone: +49-681-302-4131  
email: Roland.Vogt@dfki.de

---

**On behalf of**
Bundesamt für Sicherheit in der Informationstechnik (BSI)  
[German Information Security Agency]
Postfach 20 03 63  
53133 Bonn  
GERMANY

**Contact:**  
Markus Ullmann  
Ref. VI 4  
phone: +49-228-9582-268  
email: ullmann@bsi.de
## Contents

1 Introduction 5

2 ITSEC Requirements 7
   2.1 Interpretation by ITSEC JIL 9
   2.2 Ascertainment of Requirements 10

3 Generic Formal Model of Security Policy 11
   3.1 Formal Specification Framework (Verification Support Environment) 13
   3.2 Design Principles 13
   3.3 Overall Structure 14
   3.4 Theories 16
      3.4.1 Basic Theories 16
      3.4.2 Information and Subjects 21
      3.4.3 Enrichments of the Information Theory 27
      3.4.4 Theories for the Policy Layer 30
      3.4.5 Collections 31
   3.5 Temporal Logic Specification 33
      3.5.1 Security Objectives Layer 33
      3.5.2 Security Policy Layer 35

4 Informal Interpretation 54
   4.1 Prevent extraction or modification of the SigG private signature key of the cardholder (SO1) 54
      4.1.1 SO1.1 54
      4.1.2 SO1.2 55
   4.2 Prevent unauthorised use of the SigG digital signature funktion (SO2) 55
      4.2.1 SO2.1 55
      4.2.2 SO2.2 56
      4.2.3 SO2.3 57
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>Verification of the SigG accreditation of the IFD (SO3)</td>
<td>57</td>
</tr>
<tr>
<td>4.3.1</td>
<td>SO3.1</td>
<td>57</td>
</tr>
<tr>
<td>4.3.2</td>
<td>SO3.2</td>
<td>58</td>
</tr>
<tr>
<td>4.4</td>
<td>Demonstrate the TOE's SigG accreditation to a SigG accredited IFD (SO4)</td>
<td>59</td>
</tr>
<tr>
<td>4.5</td>
<td>Prevent unauthorised disclosure and modification of communication with the IFD (SO5)</td>
<td>59</td>
</tr>
<tr>
<td>4.5.1</td>
<td>SO5.1</td>
<td>59</td>
</tr>
<tr>
<td>4.5.2</td>
<td>SO5.2</td>
<td>60</td>
</tr>
<tr>
<td>4.6</td>
<td>Quality of key generation (SO6)</td>
<td>62</td>
</tr>
<tr>
<td>4.7</td>
<td>Provide secure digital signature (SO7)</td>
<td>63</td>
</tr>
<tr>
<td>4.7.1</td>
<td>SO7.1</td>
<td>63</td>
</tr>
<tr>
<td>4.7.2</td>
<td>SO7.2</td>
<td>63</td>
</tr>
<tr>
<td>4.8</td>
<td>React to potential security violations (SO8)</td>
<td>64</td>
</tr>
<tr>
<td>4.8.1</td>
<td>SO8.1</td>
<td>64</td>
</tr>
<tr>
<td>4.8.2</td>
<td>SO8.2</td>
<td>64</td>
</tr>
</tbody>
</table>

**A The VSE model** 68
Chapter 1

Introduction

In 1997 the government of the Federal Republic of Germany established the Information and Communication Services Act [2]. It contains the Digital Signature Act [1] as article 3. On top of that law the german parliament approved the Digital Signature Ordinance [4]. The aim of [1] and [4] is the definition of requirements that enable the equivalence of manual and digital signatures concerning legal binding, authenticity and integrity of electronic documents in the future. One specific requirement of [4] (cf. § 17) is the evaluation of technical components according to the Information Technology Security Evaluation Criteria [12]. Especially, the devices for the generation of cryptographic keys as well as the facilities for storing and applying the private signature key require an evaluation according to assurance level E4 of [12]. A formal model of security policy (FMSP) is needed to reach this level.

SmartCards are appropriate devices for the creation of digital signatures. The German standardisation organisation (DIN) developed an interface specification for SmartCards with digital signature application/function [5]. The aim of this standard is to guarantee interoperability. It specifies the interface between a terminal (interface device) and a digital signature card which is in compliance with the german digital signature law. This specification takes all the legal regulations into account. It is based on the ISO-Standard 7816 [3] and defines the data objects within a file structure as well as a collection of interface commands to be used for the creation resp. validation of digital signatures.

The Generic Formal Model of Security Policy presented in this document has been developed within a pre-evaluation project of SmartCards with signature application. It is based on the interface specification [5] and defines the technical security policy in correspondance with
the accompanying Generic Security Target [15]. The target of evaluation is the embedded software of the SmartCard exclusively. Neither the hardware nor the card operating system is subject to closer examination.

The actual FMSP is developed within the Verification Support Environment (VSE). Its textual representation is provided in the appendix of the present document.
Chapter 2

ITSEC Requirements

The purpose of this chapter is to precisely ascertain the requirements of ITSEC imposed on a Formal Model of Security Policy (FMSP). At first, we literally state all paragraphs of ITSEC related to FMSPs. In accordance with the interpretations of the ITSEC Joint Interpretation Library (JIL) reproduced in Sec. 2.1 we describe all the requirements for the present report in Sec. 2.2.

Introduction

In these criteria, Security features are viewed at three levels. The most abstract view is of security objectives: the contribution to security which a TOE is intended to achieve. To achieve these objectives, the TOE must contain certain security enforcing functions. These security enforcing functions, in turn, must be implemented by specific security mechanisms.

Formal Specification

A formal style of specification is written in a formal notation based upon well-established mathematical concepts. The concepts are used to define the syntax and semantics of the notation, and the proof rules supporting logical reasoning. Formal specifications must be capable of being shown to be derivable from a set of stated axioms, and must be capable of showing the validity of key properties such as the delivery of a valid output for all possible inputs. Where hierarchical levels of specification exist, it must be possible to demonstrate that each level maintains the properties established for the previous level.
The syntactic and semantic rules supporting a formal notation used in a security target shall define how to recognise constructs unambiguously and determine their meaning. Where proof rules are used to support logical reasoning, there shall be evidence that it is impossible to derive contradictions. All rules supporting the notation shall be defined or referenced. All constructs used in a formal specification shall be completely described by the supporting rules. The formal notation shall allow the specification of both the effect of a function and all exceptional or error conditions associated with that function.

Example formal notations are ... The use of constructs from predicate (or other) logic and set theory as a formal notation is acceptable, provided that the conventions (supporting rules) are documented or referenced (as set out above).

**Formal Models of Security Policy**

At Evaluation levels E4 and above, a TOE must implement an underlying model of security policy, i.e. there must be an abstract statement of the important principles of Security that the TOE will enforce. This shall be expressed in a formal style, as a formal model of security policy. All or part of a suitable published model can be referenced, otherwise a model shall be provided as part of the security target. Any of the formal specification styles identified above may be used to define such a model.

The formal model need not cover all the security enforcing functions specified within the security target. However, an informal interpretation of the model in terms of the security target shall be provided, and shall show that the security target implements the underlying security policy and contains no functions that conflict with that underlying policy.

**Requirements for content and presentation**

A formal model of security policy shall be provided or referenced to define the underlying security policy to be enforced by the TOE. An informal interpretation of this model in terms of the security target shall be provided. ...  

**Requirements for evidence**

The informal interpretation of the formal security policy model shall describe how the security target satisfies the underlying security policy.
2.1 Interpretation by ITSEC JIL

Based on the official requirements presented above the Joint Interpretation Working Group (JIWG) agreed upon a number of details concerning formal models of security policy which are stated below.

Background

ITSEC [12, Par. 2.78] provides examples of formal notations. Additional notations are CSP, VSE, B. . .

Formal Model of Security Policy

The FMSP's aim is to enhance the assurance by formally specifying and proving that the TOE correctly enforces the stated security policy.

As described in ITSEM [13, Par. 6.B.25], the system security policy for a system, or the product rationale for a product, should state in the security target the important principles of security (referred to as “the Security Policy”):

- for a system, it corresponds . . .

- for a product, it corresponds to the product rationale which gives an equivalence to the “system security objectives” by identifying the product’s security features and all environmental assumptions [13, Par. 6.B.25-6.B.28]. In some cases, a product rationale may specify security objectives.

At E4 and above, part or all the TOE Security Policy of the system or product, known in ITSEC as the Underlying Security Policy, shall be expressed in a formal style in the FMSP.

Proofs

The ITSEC [12] requires evidence in order to satisfy requirements. The following proofs shall be presented as evidence.

FMSP proofs FMSP proofs shall prove evidence for the correctness of the security model. This includes but is not limited to the internal consistency of the security model, in the sense of non-existence of contradictions and invariance (i.e. the impossibility of transition from secure to insecure states) of its properties.
2.2 Ascertainment of Requirements

In Accordance with the requirements presented above we conclude that the FMSP documented together with its informal interpretation in the present report should meet the following requirements.

1. A formal model of security policy (FMSP) shall be provided to define the underlying security policy to be enforced by the TOE. In accordance with ITSEC JIL [8, Par. 279] the FMSP should consist of two parts:

   (a) A formal specification of the important principles of security of the TOE. In the sense of a narrow interpretation this corresponds to the term “underlying security policy” used in ITSEC [12, E4.2]. This part of the FMSP is documented in Sec. 3.5.2.

   (b) A formal specification of the security features of the TOE identified in the product rationale. According to ITSEC [12, 2.2] this corresponds to the security objectives described in the security target. This part of the FMSP is documented in Sec. 3.5.1.

2. An informal interpretation of the FMSP shall be provided which shows that the security target implements the underlying security policy [12, 2.82]. In accordance with ITSEC [12, E4.2 and E4.3] this requirement shall be met in terms of informally demonstrating the correspondance between the security objectives provided in the security target and the characteristic properties of the FMSP. The informal interpretation is contained in Sec. 4.

3. The following proofs shall be provided as evidence.

   (a) A proof of the invariance of the FMSP’s properties formally verifying those characteristics of the security policy which correspond to the security objectives. This is in accordance with ITSEC JIL [8, Par. 278, 287]. The formal proofs are contained in the data base of the VSE system.

   (b) A proof of the absence of contradictions (internal consistency) within the FMSP. This is in accordance with ITSEC JIL [8, Par. 287]. The formal proofs are contained in the data base of the VSE system.
Chapter 3

Generic Formal Model of Security Policy

This chapter describes the formal specification defining the underlying security policy of the TOE. It formalizes the important security principles and relates them to the security objectives described in the Generic Security Target [15]. The formal model was developed using the ‘Verification Support Environment’ (VSE) which is a tool supporting formal methods in the complete software life cycle [9, 10]. It is officially approved for the definition of formal specifications as required by ITSEC [12] or Common Criteria [6].

In our context we look at SmartCards which are ready for signing. This means that the manufacturing and personalization processes of the SmartCard are completed, i.e. the integrated circuit of the SmartCard is loaded with the digital signature application and the data base related to a specific cardholder. It is assumed that this data base has been processed by the personalization authority in a secure way and that the SmartCard is securely handed over to the legitimate cardholder.

The digital signature operation is processed in the TOE. For that process the secret signature key of the cardholder is used which is stored in a secure manner within the TOE. The ICC is accessed with a SmartCard reader connected to e.g. a personal computer. It is responsible for transmitting the data to which the cardholder wishes to apply a digital signature. In the following we call all external devices (SmartCard reader, card terminal, ... ) which might be involved in accessing the ICC ‘interface device’ (IFD).

The German signature legislative distinguishes ‘private’ and ‘public’ IFDs. The cardholder is expected to trust his own private IFD. Here
the cardholder himself is responsible to use an IFD which is compliant with the legislative. Public IFDs provide commercial services. In this case the cardholder is not able to control or even know whether the IFD is compliant with SigG legislative.

The main principle of the security policy of the TOE is that the cardholder is the only legitimate user. Not any other person is allowed to use her/his TOE. The TOE contains the cardholder's secret key for signing. It is regarded as the most valuable asset. The TOE must keep this key confidential at all events. Digital signatures are only produced with the explicit permission of the cardholder.

In preparation of the development of the security requirements for the TOE a threat analysis has been performed. That analysis didn’t take the hardware or the operating system of the SmartCard into account. The examination was restricted to the embedded application software and data of the ICC and revealed the following threats (cf. [15, Sec. 2.5]):

(T1) 1. Extraction or modification of the cardholder's secret key either by directly manipulating the data or by analysing computational results.

(T2) 2. Misuse of the signature function by using the signature application without having the permission of the cardholder.

(T3) 3. Unnoticeable creation or modification of signed data being ascribed to the cardholder.

(T4) 4. Presentation of a forged ICC to a public IFD without the public IFD being able to notice that.

(T5) 5. Presentation of the ICC to a forged public IFD without the cardholder being able to notice that.

The specific threats can be summarised as such: The cardholder's signature is created for a piece of data the cardholder does not want to sign.
3.1 Formal Specification Framework
(Verification Support Environment)

The Verification Support Environment (VSE), which was developed on behalf of the German Information Security Agency (Bundesamt für Sicherheit in der Informationstechnik – BSI) is a complex framework for applying formal methods in the development of provably correct software. VSE was formally approved by the BSI and is officially recommended for software developments of IT products with the highest quality levels according to ITSEC [12] and Common Criteria [6]. In several projects VSE has proven to be adequate for practical applications.

The VSE system provides means to structure specifications and supports the development process from the specification of a system to the automatic generation of code. Formal developments following the VSE methodology are stored in an administration framework that guides the user and maintains a consistent state of the development. An integrated deduction system provides proof support for the deduction problems arising during the development process. Unlike other formal development tools the VSE system supports comprehensive methods in order to deal with distributed and concurrent systems [14]. The specification language provides a variant of TLA (temporal logic of actions) [11] in combination with classical abstract data type specifications using predicate logic.

3.2 Design Principles

The interface of the ICC is modeled by input and output variables. The regular input and output of the ICC is modeled by an input variable \textit{channelIn} and an output variable \textit{channelOut}. The values of these variables abstractly represent the incoming resp. outgoing messages to resp. from the SmartCard.

In order to be able to distinguish between a message that is sent once and one that is sent several times a gap is modeled which separates two messages. This is done using the value \textit{noInfo} representing the absence of information. We expect the input variable \textit{channelIn} to change from \textit{noInfo} to a value different from \textit{noInfo} and back again, but never to immediately change between two values both different from \textit{noInfo}. The ICC is modeled in such a way that this is also valid.
for the output variable channelOut.

According to [3] the communication between the SmartCard and the terminal is like a question and answer game. The SmartCard answers to commands received from the IFD. If the IFD does not see an answer within a well defined period of time it may try to send the command again or (after a few trials) reset the card. Commands and answers are both modeled as elements of type information that will also be called messages in the following.

Since the card is the slave part in the communication it is the responsibility of the IFD to react if an expected answer is not returned. Therefore, we need not put any timing constraints on reactions and answers from the card. It is not necessary that a command is reacted upon by immediately producing a corresponding answer. It suffices to require that if the TOE reacts to a command, the answer properly reflects its internal state.

Nevertheless, as we are concerned with the flow of information between the ICC and its environment we have to take temporal aspects into account. So the main part of the specification is formulated in a temporal logic specification language [7]. In order to define the states which may change as time proceeds we need data elements that form these states as well as static relations and functions on these elements. This leads to an underlying structure providing abstract data types necessary for our specification.

The most important data types are subject and information. Elements of data type subject are equipped with a certain amount of knowledge consisting of an undetermined number of elements of type information. Subjects can enrich their knowledge base by learning new information. This process doesn’t change the identity of the resp. subject. The main purpose for introducing these data types is the definition of the predicate inferable which describes the principal possibility of information extraction. According to the defining axiom inferable(i, k) is valid if there is at least one subject which knows k after learning i although it did not knew k before.

3.3 Overall Structure

The formal specification is divided in two layers. The upper layer takes a global viewpoint and describes the TOE together with the environment in which it is expected to be used. It formalizes the security objectives specified in [15] as well as assumptions about the environment.
We call this part of the specification the ‘security objectives layer’.

The lower layer takes a local viewpoint and describes the ICC at an operational level. It defines the security policy of the TOE in terms of a formalization of the important security principles extracted from [15]. We call this part of the specification the ‘security policy layer’.

The two layers are connected by a satisfies link. It expresses that the operational behavior specified in the policy layer should fulfill the properties formalized in the objectives layer. More generally such a link states the every correct implementation of an operational specification should be a model (w.r.t. the semantics of the specification language) of the associated requirement specification. In our case the operational specification corresponds to the policy layer while the requirement specification corresponds to the objectives layer. This connection leads to proof obligations which are automatically generated by the VSE system. Formally verifying these obligations means demonstrating that the system described in ICC_Policy meets the properties required in ICC_Objectives. Thus satisfying the proof obligations will guarantee the external consistency of the security policy w.r.t. to the security objectives.

As we are concerned with protocols here we have to take temporal aspects into account. So the main part of the specification for both layers is formulated in a temporal logic specification language. In order to define the states which may change as time proceeds we need data structures that form these states as well as (static) relations and functions that axiomatically describe their semantics. This leads to the abstract data types part of our specification.

The structure of the specification can be depicted as shown here:

![Diagram](image-url)

The following two sections take a closer look on these parts.
3.4 Theories

The theories define the data elements that are used to form the states of the temporal logic models. They also provide predicates and functions on these elements that will be used in the temporal logic specifications.

The theories are described in the following subsections.

3.4.1 Basic Theories

First we describe some simple auxiliary theories which will be needed in the definitions of several other theories.

3.4.1.1 TAnswer

The purpose of the theory $T_{\text{Answer}}$ is the definition of constants for answers generated by the ICC. The data type $\text{answer}$ is freely generated by the following constructors:

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description: Answer to . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_{\text{Modified}}$</td>
<td>the modify command</td>
</tr>
<tr>
<td>$a_{\text{Success}}$</td>
<td>a successfully completed command (OK response)</td>
</tr>
<tr>
<td>$a_{\text{AuthSuccess}}$</td>
<td>a successful PIN authentication (OK response)</td>
</tr>
<tr>
<td>$a_{\text{AuthPUKSuccess}}$</td>
<td>a successful PUK authentication (OK response)</td>
</tr>
<tr>
<td>$a_{\text{AuthIfdSuccess}}$</td>
<td>a successful IFD authentication (OK response)</td>
</tr>
<tr>
<td>$a_{\text{Failure}}$</td>
<td>a failed command</td>
</tr>
<tr>
<td>$a_{\text{AuthFailure}}$</td>
<td>a failed authentication command</td>
</tr>
<tr>
<td>$a_{\text{FatalFailure}}$</td>
<td>a fatally failed command</td>
</tr>
<tr>
<td>$a_{\text{Denied}}$</td>
<td>a command not allowed in the current state</td>
</tr>
<tr>
<td>$a_{\text{Reset}}$</td>
<td>a power up reset</td>
</tr>
<tr>
<td>$a_{\text{Closed}}$</td>
<td>all commands closing the signature application</td>
</tr>
<tr>
<td>$a_{\text{Disabled}}$</td>
<td>an unsuccessful user authentication caused by a disabled ICC</td>
</tr>
</tbody>
</table>
3.4.1.2 TBucks

The purpose of the theory TBucks is the definition of control values to be assigned to a shared variable in order to coordinate the components of the policy layer. The data type bucks is freely generated by the following constructors:

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description: Next action or component to continue is</th>
</tr>
</thead>
<tbody>
<tr>
<td>bckSecChannelIn</td>
<td>the component SecChannelIn</td>
</tr>
<tr>
<td>bckObjects</td>
<td>any Object component</td>
</tr>
<tr>
<td>bckAutomaton</td>
<td>the automaton component</td>
</tr>
<tr>
<td>bckSecChannelOut</td>
<td>the component SecChannelOut</td>
</tr>
<tr>
<td>bckObjectReset8</td>
<td>the actions to reset Object 8 and 9</td>
</tr>
<tr>
<td></td>
<td>the action to reset Object 8 (after Object 9 has already been reset)</td>
</tr>
<tr>
<td>bckObjectReset9</td>
<td>the action to reset Object 9 (after Object 8 has already been reset)</td>
</tr>
<tr>
<td>bckO3Unblock</td>
<td>the action to unblock the PIN (after successful verification of the PUK)</td>
</tr>
<tr>
<td>bckSecChannelOutReady</td>
<td>the action to check whether SecChannelOut is ready to handle the next output</td>
</tr>
</tbody>
</table>
3.4.1.3 TCommand

The purpose of the theory TCommand is the definition of constants for commands generated by the IFD. The data type ifdCommand is freely generated by the following constructors:

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sign1</td>
<td>Command to sign the information given as parameter</td>
</tr>
<tr>
<td>Auth1</td>
<td>Command to do user authentication with PIN</td>
</tr>
<tr>
<td>PUKAuth2</td>
<td>Command to do user authentication with PUK, #1 is the PUK, #2 the new PIN</td>
</tr>
<tr>
<td>AuthChange2</td>
<td>Command to do user authentication and change the PIN, #1 is the old PIN, #2 the new one</td>
</tr>
<tr>
<td>MessageChange1</td>
<td>Command to change the display message</td>
</tr>
<tr>
<td>IntAuth1</td>
<td>Command to authenticate the ICC, #1 is the challenge</td>
</tr>
<tr>
<td>ExtAuth1</td>
<td>Command to authenticate the IFD, #1 is the certificate</td>
</tr>
<tr>
<td>Exit0</td>
<td>Command to exit (close) the SigG application</td>
</tr>
<tr>
<td>Start0</td>
<td>Command to start (open) the SigG application</td>
</tr>
<tr>
<td>Verify2</td>
<td>Command to do signature verification with key reference #1 on signed message #2</td>
</tr>
<tr>
<td>ModifyObj...1</td>
<td>Commands to modify objects</td>
</tr>
<tr>
<td>ReadObj...0</td>
<td>Commands to read objects</td>
</tr>
</tbody>
</table>

3.4.1.4 TElement

The theory TElement defines the data type element. Its only purpose is to be used as a parameter of the theories TList and TMaybe.
3.4.1.5 TEvent

The theory TEvent defines the data type events which consists of constants representing the security related events as described in the GST [15, Tbl. 5]. The data type events is freely generated by the following constructors:

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sre1</td>
<td>Resetting of the ICC</td>
</tr>
<tr>
<td>sre2</td>
<td>Deactivation of the ICC</td>
</tr>
<tr>
<td>sre3a</td>
<td>Opening of the SigG application with unblocked reference data (authentication enabled)</td>
</tr>
<tr>
<td>sre3b</td>
<td>Opening of the SigG application with blocked reference data (authentication disabled because of previous repeated authentication failures)</td>
</tr>
<tr>
<td>sre4</td>
<td>Closing of the SigG application</td>
</tr>
<tr>
<td>sre5</td>
<td>Successful cardholder authentication</td>
</tr>
<tr>
<td>sre6</td>
<td>Cardholder authentication failure leaving the authentication enabled (reference data unblocked)</td>
</tr>
<tr>
<td>sre7</td>
<td>Repeated authentication failure disabling authentication (reference data blocked)</td>
</tr>
<tr>
<td>sre8</td>
<td>Authentication expiration</td>
</tr>
<tr>
<td>sre9</td>
<td>Successful mutual device authentication (Option Public IFD)</td>
</tr>
<tr>
<td>sre10</td>
<td>Potential security violation occurred</td>
</tr>
<tr>
<td>sre11</td>
<td>Cardholder authenticated by reset code (PUK)</td>
</tr>
<tr>
<td>sre12</td>
<td>Cardholder authentication by reset code failed</td>
</tr>
</tbody>
</table>

3.4.1.6 TList

The theory TList defines generic lists. The constant nil represents the empty list. The constructors cons increases the list given as second argument by prepending the element given as the other argument. Selectors are first and rest returning the first element of a list resp. the rest of the list after removing the first element.

The type of list elements will be instantiated with the basic type information (cf. 3.4.2.2) resulting in a data type useful as history lists.
3.4.1.7 TMaybe

The theory TMaybe is a generic theory which is used to add an exceptional value to an existing data type. The exceptional value is represented by the constant undef. Ordinary values are carried over to the new type m with the embedding function def. This embedding is applied to the types information and event (cf. Sec. 3.4.2.3 and 3.4.4.1). We will call sorts enhanced with TMaybe generalized. Consequently we will speak about generalized informations resp. messages and generalized events.

3.4.1.8 TObjectIds

The theory TObjectIds defines the type objectIds which consists of constants, one for each of the objects enumerated in the GST (cf. [15, Tab. 8]). The data type objectIds is freely generated by the following constructors:

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj1</td>
<td>the complete SigG application</td>
</tr>
<tr>
<td>obj2</td>
<td>SigG private signature key of the cardholder</td>
</tr>
<tr>
<td>obj3</td>
<td>SigG cardholder reference data</td>
</tr>
<tr>
<td>obj4</td>
<td>SigG cardholder reset code</td>
</tr>
<tr>
<td>obj5</td>
<td>SigG signature key certificate of the cardholder</td>
</tr>
<tr>
<td>obj6</td>
<td>SigG public key of the root certification authority</td>
</tr>
<tr>
<td>obj7</td>
<td>Other credentials for signature verification</td>
</tr>
<tr>
<td>obj8</td>
<td>Private device authentication key of the TOE</td>
</tr>
<tr>
<td>obj9</td>
<td>Public key of the root certification authority for device authentication (Option Public IFD)</td>
</tr>
<tr>
<td>obj10</td>
<td>Device authentication credentials of the TOE (Option Public IFD)</td>
</tr>
<tr>
<td>obj11</td>
<td>Cardholder display message (Option Public IFD)</td>
</tr>
<tr>
<td>obj12</td>
<td>SigG public key of the cardholder</td>
</tr>
</tbody>
</table>

The GST contains the object identifier obj13 representing User data of the communication (cf. [15, Sec. 3.1.3]). There is no such constructor in the present data type as this object is not managed by the TOE. Due to principal observations, the access to these data cannot be under control of the TOE.
3.4.1.9 TStates

The theory TStates provides the data type states, which contains a constant for each current authentication state (CAS) as found in of the GST [15, Tab. 11]. The data type states is freely generated by the following constructors:

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cas1</td>
<td>the card is working, but the SigG application has not yet been started (Somebody using the TOE)</td>
</tr>
<tr>
<td>cas2</td>
<td>the SigG application is running; no authentications have been done (Somebody using the SigG application)</td>
</tr>
<tr>
<td>cas3</td>
<td>Only the cardholder (but not yet the IFD) has been successful authenticated (Cardholder using a IFD)</td>
</tr>
<tr>
<td>cas4</td>
<td>Only the IFD has been successfully authenticated (Somebody using a SigG accredited IFD)</td>
</tr>
<tr>
<td>cas5</td>
<td>cardholder and IFD have been successfully authenticated (Cardholder using a SigG accredited IFD)</td>
</tr>
<tr>
<td>cas6</td>
<td>the SigG application is disabled after a potential security violation (A potential attacker)</td>
</tr>
<tr>
<td>cas7</td>
<td>the IFD has not been authenticated; cardholder authentication by PIN is disabled because of repeated authentication failures (Somebody using the SigG application with blocked Cardholder reference data)</td>
</tr>
<tr>
<td>cas8</td>
<td>the IFD has been authenticated; cardholder authentication by PIN is disabled because of repeated authentication failures (Somebody using a SigG accredited IFD and Cardholder reference data being blocked)</td>
</tr>
</tbody>
</table>

3.4.2 Information and Subjects

The theories must be suitable to describe the security objectives. As most of these objectives deal with (non)disclosure of information, the theories have to provide a framework for the formalization of informal descriptions in terms of these notions. Therefore the most important types defined in the theories are information representing the pieces of informations or messages, and the type subject representing the potential carriers of information. These types are defined in TInformation and TSubject. Sometimes variables to which values of type information should be assigned can be in a state where they have no value. This is the case for an output variable if there is currently no output. There-
fore many variables have type \textit{info.m} instead of \textit{information}. This type is defined in theory \texttt{TInstInformation} by instantiating theory \texttt{TMaybe} with type \textit{information}. By doing so we add a constant \texttt{noInfo} representing "no information" to the type \textit{information}. All other theories that have to do with information include \texttt{TInstInformation} (indirectly). They are described in more detail in this subsection.

### 3.4.2.1 TSubjects

The theory \texttt{TSubjects} defines the type \textit{subject}. This type has the potential carriers of information as members. Included are at least

- the ICC,
- the IFD,
- authorized and unauthorized users (intruders) of the SigG application.

This type will be used in theory \texttt{TInformation}

### 3.4.2.2 TInformation

An element of type \textit{information} represents pieces of information or messages. This includes at least those informations that can be stored on the ICC, the documents to be signed by the users and any auxiliary information used in the process of generating the digital signature. We regard signature and encryption keys as elements of type \textit{information}. This type should represent all informations cryptanalysts can use to infer secrets stored on the card\footnote{But details are left open in the formal model. See comment on the predicate \texttt{learns}.}. In order to be able to formulate concise axioms about the uniqueness of keys and signatures we restrict the type \textit{information} to contain only a subset of all possible informations but at least those, which can be expected to occur within the ICC application and its environment during its life time. One could also think of documents as being represented as their hash values in the model. Doing so the uniqueness of the signatures generated for the documents (or their hash values) presented to the application can be expressed by formulating an axiom about injectivity of the signature function. This is possible even though the real signature function is not injective in a strong sense.
An element of type \textit{subject} can be regarded as any carrier of elements of type \textit{information}. This may be the ICC, the terminal, the user etc.

In the following all predicates and functions defined in TInformation are described. Note that most of them are not intended to be implementable but are merely for description purpose. So they only occur in the objective layer (specification ICC Objectives) and in places where information is described that is not computed by the ICC itself. This is the case for the specification (ICC Assumptions), describing assumptions about the input values of the ICC and theories that define values used for initializing the variables of the policy layer specification.

\textbf{knows:} A predicate

\[ \text{knows} : \text{subject, information} \]

is defined which is used to express the fact that a subject has a piece of information in the sense that it is stored on the subject or the subject knows it or has it on a piece of paper etc.

\textbf{learns:} The most important feature of informations and subjects in our model is, that informations can be exchanged among the subjects. So a subject can change its knowledge in such a way that it acquires a new piece of information. This is expressed by the function

\[ \text{learns} : \text{subject, information} \rightarrow \text{subject} \]

If the subject \( s \) learns the message \( i \) or adds it to its knowledge base, the subject becomes \( \text{learns}(s, i) \). Of course the axioms guarantee, that after learning \( i \) the subject knows \( i \) and all informations it knew before.

But we do not require \( i \) to be the only new information the subject will know afterwards. This allows interpretations of the model in which subjects are able to infer informations from other informations. A simple example is this: If keys are products \( a \ast b \) and a subject knows the factors \( a \) and \( b \), then he also knows the key.

On the other hand we deliberately do not go into much detail on how informations can be inferred from other informations because this could easily lead to an inconsistent specification. Grossly speaking the formal model does not rule out subjects that only change their knowledge base by adding the piece of information just learnt and nothing
Such a model will be constructed to work out the consistency proofs.

As a consequence of the above discussion we avoid to define the effect of $\text{learns}$ by describing exactly which information is added. (The formula $\forall j. \text{knows}(\text{learns}(s, i), j) \leftrightarrow i = j \lor \text{knows}(s, j)$ is an obvious example for an undesirable axiom.) We rather describe the information gained in learning in an indirect manner by determining the subjects that know more or the same than a subject $\text{learns}(s, i)$. In order to do so the relation $<<$ is introduced. The formula $s << t$ can be interpreted as “$t$ knows anything $s$ knows and possibly more”. This interpretation directly corresponds to its defining axiom: $s << t \leftrightarrow \forall i. \text{knows}(s, i) \rightarrow \text{knows}(t, i)$. Obviously, $<<$ is a reflexive and transitive relation.

The relationship between $\text{learns}$ and $\text{knows}$ can now be stated as follows:

$$s << t \land \text{knows}(t, i) \leftrightarrow \text{learns}(s, i) << t$$

This could be circumscribed as: The new knowledge of $\text{learns}(s, i)$ is the “smallest” knowledge containing the old knowledge of $s$ and the new piece of information $i$. We avoid to make explicit what else will be learnt implicitly when learning $i$.

Note that the axiom above allows us to infer important properties like $\text{knows}(\text{learns}(s, i), i)$ and $\text{knows}(s, j) \rightarrow \text{knows}(\text{learns}(s, i), j)$. This is easily achieved by appropriately replacing $t$ and logically simplifying the formula using properties like the reflexivity of $<<$.

**learnsall:** The function

$$\text{learnsall} : \text{subject, subject} \rightarrow \text{subject}$$

transforms a subject $s$ and a subject $t$ into a new subject $\text{learnsall}(s, t)$ which has the identity from $s$ and the knowledge of both $s$ and $t$.

**Equality of knowledge** If two subjects $s$ and $t$ have the same knowledge this fact is written as $s == t$.

**Identity** The relation $==$ must not be confused with the predicate $\text{identical}$. The equivalence relation $\text{identical}$ serves to keep track of a subject while its knowledge changes with the advance of time. For example we can assume two consecutive states of a system representing

---

2There are a few exceptions. But they do not affect the consistency proof.
the environment of the card containing a representation of the subject cardholder which may be written as $ch$ and $ch'$. If the knowledge of the cardholder changes we have $ch \neq ch'$. But the identity of the cardholder remains the same. This fact is expressed by the formula $\text{identical}(ch, ch')$.

**inferable**: When specifying non disclosure conditions, does not suffice to say that the output variable never assumes a certain piece of information $i$. Otherwise a subject $s$ who already knew $i$ 'partially' could infer $i$, if the missing parts would be sent out. Therefore we make it precise what 'partially' means and use the resulting predicate to express the condition. The predicate is called

$$\text{inferable} : \text{information, information}$$

The defining axiom is

$$\text{inferable}(i, j) \leftrightarrow \exists s : \neg \text{knows}(s, j) \land \text{knows}(\text{learns}(s, i), j);$$

This axiom could be circumscribed as: At least one subject that has learnt $i$ also knows $j$, even though it did not know $j$ before.

Note that inferable is reflexive ($\text{inferable}(i, i)$ is valid for all $i$). As a consequence the formula $\neg \text{inferable}(i, j)$ which occurs often in the specification always implies $i \neq j$.

**inferableWithout**: We must express conditions that exclude the possibility of generating a signature, if the user has not been authenticated or a security violation has occurred. In this case the output of the card must not contain any signed data, in the sense that no such information can be inferred from the output. Using the formula $\neg \text{inferable}(\text{out}, \text{sig}(sk, i))$ for an output variable $\text{out}$ and a term $\text{sig}(sk, i)$ representing a piece of data $i$ signed with the secret key $sk$ (for the explanation of the function $\text{sig}$ see below) certainly is not the desired approach. Rather, what should be expressed is that without knowledge of the secret key no signed data can be inferred from the output. An important property of this restriction is that every unsigned information is allowed to leave the ICC as long as the secret signature key is not known outside. To formalise this restricted inference notion we introduce the following predicate:

$$\text{inferableWithout} : \text{information, information, information}$$
According to its defining Axiom \( \text{inferableWithout}(i, j, sk) \) can be circumscribed as: At least one subject that has learnt \( i \) \emph{and does not know} \( sk \) \emph{afterwards} also knows \( j \), even though it did not knew it before.

**Elimination of Omniscience:** Parts of a structured specification should not make restrictions that implicitly affect other parts of the specification. Outside the theory \( \text{Information} \) there will be assumptions about the privacy of several kinds of secret information. This requires that there are enough pieces of information that are not known by other subjects. Therefore the following axiom is introduced:

\[
\forall s : \exists i : \neg \text{knows}(s, i)
\]

Otherwise \( \text{Information} \) would have a model in which all subjects know either nothing or anything. This possible model would only be ruled out by those TL-specifications and theories that allow us to infer the existence of at least two different pieces of information each one known to only one subject. (For example: The signature key and the secret keys for secure messaging known to the IFD)

### 3.4.2.3 \( \text{TInstInformation} \)

The \( \text{TInstInformation} \) is the place where \( \text{TMaybe} \) is instantiated with the type \emph{information}. Predicates and functions from \( \text{Information} \) are extended to the new type \emph{info.m} by defining new predicates and functions in terms of the old ones.

### 3.4.2.4 \( \text{TClassification} \)

In order to be able to solve the proof obligations, it must be possible to show for example that the cardholder’s signature key never appears in the output of the card (and nothing usable to infer the key either). But on the other hand other messages must appear there, for example the display messages. For this reason informations are classified according to their possible use for different kinds of secret or public information. Axioms are included in \( \text{TClassification} \) that guarantee the disjointness of these classes. Even the possibility to infer secrets of one class from those of a different class or public informations is excluded. The predicates are

\[
isSecret : \text{kindSecret, informations} \\
isPublic : \text{information}
\]
The sort \textit{kindSecret} is a freely generated type having one constant for each class of secrets.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{kindSkCh}</td>
<td>Secret key to generate the cardholder’s signature</td>
</tr>
<tr>
<td>\textit{kindCertIfd}</td>
<td>Secret data to generate correct answers for the validation of IFD certificates</td>
</tr>
<tr>
<td>\textit{kindCertIcc}</td>
<td>Secret data to produce correct answers for the validation of ICC certificates</td>
</tr>
<tr>
<td>\textit{kindPIN}</td>
<td>the PIN (or other user authentication data)</td>
</tr>
<tr>
<td>\textit{kindSecureMsg}</td>
<td>keys for secure messaging</td>
</tr>
<tr>
<td>\textit{kindDisplayMessage}</td>
<td>display messages</td>
</tr>
</tbody>
</table>

### 3.4.3 Enrichments of the Information Theory

The following theories are based on the information theory described in the previous section. They substantiate the information (and classification) theory by adding definitions of the messages sent by the terminal and the ICC. We also include theories for the cardholder’s signatures and secure messaging in this subsection.

#### 3.4.3.1 TTerminal

The theory \textbf{TTerminal} defines functions and predicates related to the messages sent from the IFD to the ICC. This includes a mapping named \textit{commandInfo} associating a message for each command defined in theory \textbf{TCommand} to a message of type \textit{info.m}. So this function can be understood as a function converting abstract commands to there representation as messages. A reverse function translating messages back to abstract commands is defined with the name \textit{theIfdCommand}.

The theory introduces abbreviations \textit{commandInfo(cmd,...)} for the terms of the form \textit{ifdDoCmd(...)} where \textit{ifdDoCmd} is one of the function (constant) symbols \textit{ifdDoSign}, \textit{ifdDoAuth}, etc.

Predicates are provided to decide whether a certain message is a command. The predicate \textit{isIfdCommand} is true if and only if the parameter given is the representation of a command from the terminal to the card. A stronger predicate is \textit{isSigGCommand} which becomes true only for commands concerning the SigG application. The predicate \textit{isCertIfdKey} can be used to decide the validity of IFD certificates. The formula \textit{isCertIfdKey}(k, i) hold if and only if \textit{i} represents a valid certifi-
cate when checked against a public key $k$. In the top level specification we use isCertIfd which checks against the key initialIfdAuthCred.

### 3.4.3.2 TCard

The theory TCard defines functions and predicates related to the messages the card sends to the terminal.

The abstract answers to commands and to security violations defined in TAnswer (cf. 3.4.1.1) are mapped to messages by the function named answerInfo. The predicate isAnswer can be used to check if a given message is in the range of answerInfo.

More complex output of the card is dealt with the functions called certIccKey and certIcc that generate ICC certificates. While these functions are defined within this theory there is a separate theory TSigNature dedicated to cardholder’s signatures (cf. 3.4.3.3). The function certIccKey takes a counter representing a challenge as argument and returns a certificate with the cardholder’s answer to the challenge. As the device authentication protocol is abstracted in such a way that certificates and answers to challenges are sent in one message this function can be used for modeling this protocol.

### 3.4.3.3 TSigNature

The theory TSigNature defines functions and predicates for the cardholder signature.

The function sig returns a signed document. For a document $i$ and a (secret) key $sk$ the document $i$ signed with key $sk$ is represented by $\text{sig}(sk, i)$. The extension of sig on generalized information is called mSig.

The predicate validpair is a test for matching pairs of secret and public key usable with the function sig. The function sig only generates signature with the required properties, if the secret keys used make validpair true.

The functions skeygen and pkeygen generate (valid) pairs of secret and public keys. The existence of these functions guarantee that there are infinitely many valid key pairs.

The predicate validsig becomes true exactly for those documents $i$ and public keys $pk$, where $i$ is signed with a secret key matching $pk$.

The axioms in this theory describe the properties of sig which render it a signature function. Most important is the fact that signatures
cannot be guessed, which is expressed by the axiom

\[
\text{validpair}(sk, pk) \land \text{knows(learns}(s, i), \text{sig}(j, sk)) \rightarrow
\text{knows}(s, sk) \lor \text{knows}(s, \text{sig}(j, sk)) \lor i = sk \lor i = \text{sig}(j, sk)
\]

Other properties are the injectivity of the signature function (i.e. different documents result in different signatures) and the fact that knowing the secret key and a document entails also knowing the documents signature. In a strong sense signature function may produce the same signature for different documents. But this is not relevant in practice and therefore we assumed the type \textit{information} does not contain pairs leading to the same signature (cf. 3.4.2.2). Further axioms define the secret resp. public key to be classified as secret of kind \textit{kindSkCh} resp. to be public.

The axiom in TSignature entail that keys cannot be inferred from signatures. This fact is included in the theory SSignature connected to TSignature by a satisfies link.

3.4.3.4 TSecureConnection

The theory TSecureConnection provides what is necessary for secure messaging.

The function \texttt{secureMsgEncode} is applied on three pieces of information. The first should be a secret key for secure message encryption (this can be checked with the predicate \texttt{validSecureMsgPair}), the second one should be a public signature key for secure messaging (checked with \texttt{validSecureMsgPairSig}) and the last one is the message to be sent. The keys can both be \texttt{noKey}. In this case the message itself is the result of the function, i.e. no encryption and signature computation is done.

The function \texttt{secureMsgDecode} is the inverse of \texttt{secureMsgEncode}. If it is given the complementary keys of those provided when applying \texttt{secureMsgEncode} on a message \textit{i}, it will return \textit{i}. If the signature keys do not match, \texttt{noInfo} is returned. If the encryption key is unknown, the message cannot be guessed. If both keys used for \texttt{secureMsgDecode} are \texttt{noInfo}, the message is returned unchanged.
3.4.4 Theories for the Policy Layer

In this subsection we present the theories that serve to model the concepts of the underlying security policy. The instantiation of \( T\)Maybe with security related events, the table of access rights and state transitions belong to this category of theories. History lists are included here because they depend on security related events.

3.4.4.1 TInstEvent

The theory \( T\)InstEvent instantiates the theory \( T\)Maybe with the type \( event \) in order to add a constant \( noEvent \) which indicates that there is currently no security relevant event.

3.4.4.2 TAccessRights

The theory \( T\)AccessRights defines the data type \( rights \) used in the policy level of the model. The type \( rights \) represents the access rights defined in the GST \([15, \text{Sec. 3.1.3}]\) for access control of the objects. Each element of this type is a constant representing one access right.

The predicate

\[
allowed: \text{objectIds, states, rights}
\]

models the access control tables of the GST \([15, \text{Tbl. 13 and 14}]\), which assign access rights to each pair of object and authentication state.

3.4.4.3 TTransitions

The theory \( T\)Transitions models the state transition table of the GST \([15, \text{Tbl. 12}]\). It provides the function \( \text{nextstate} \) which returns the appropriate successor state for a given state and a security related event. This function is defined according to the table. Blank fields in the table which stand for combinations not expected to happen are interpreted in such a way that no state change will occur.

3.4.4.4 THistory

The theory \( T\)History defines the data type used for history lists. These lists are used to collect all output messages (before secure messaging) of the SigG application. This provides us with means to talk about (all) past events when formulating security objectives.
History lists are defined by instantiating generic lists with type information. This theory also contains recursively defined predicates that express certain properties that are useful for formulating the security objectives.

The predicate histAuthUser becomes true if and only if the outputs recorded in the history are in accordance with a sequence of events leading to a state in which the card holder can be regarded as authenticated.

The predicate histAuthIfd is a similar predicate regarding IFD authentication.

The predicate maxFailuresExceeded reveals whether the output history of the SigG applications is appropriate for a sequence of events after which cardholder authentication is disabled because of exceeding the maximum of allowed authentication failures. We use the auxiliary function allowedFailures to define this predicate. It returns the number of authentication failures still allowed.

3.4.5 Collections

The following theories suit the purpose of putting several theories in one theory for easier use by the TL-specifications.

3.4.5.1 TAutomaton

The theory TAutomaton is used by the Automaton component of the policy layer. In addition to theories needed by all components of this layer we need a transition table for the state changed the automaton is responsible to perform. This table is defined in TTransitions.
3.4.5.2 TInitials

The theory \texttt{TInitials} defines values used to initialize the variables of the ICC. If necessary axioms describe how some of this initial values are classified.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialPin</td>
<td>initial PIN/cardholder reference data</td>
</tr>
<tr>
<td>initialPuk</td>
<td>initial PUK (reset key)</td>
</tr>
<tr>
<td>initialCHCredentials</td>
<td>initial cardholder’s credentials (non CA part)</td>
</tr>
<tr>
<td>initialCHCredentialsCA</td>
<td>initial cardholder’s credentials (CA part)</td>
</tr>
<tr>
<td>initialIccAuthKey</td>
<td>initial authentication key of the ICC</td>
</tr>
<tr>
<td>initialIccAuthCred</td>
<td>initial authentication credentials of the ICC</td>
</tr>
<tr>
<td>initialIfdAuthCred</td>
<td>initial authentication credentials of IFDs</td>
</tr>
<tr>
<td>initialDisplayMessage</td>
<td>initial display message</td>
</tr>
<tr>
<td>initialOtherCred</td>
<td>credentials for other signatures</td>
</tr>
<tr>
<td>initialSKeyCounter</td>
<td>initial value for the counter used for cardholder’s signature key generation</td>
</tr>
<tr>
<td>initialMaxAuthChFailures</td>
<td>Maximum of successive card holder authentication failures by PIN. The SigG functionality of the ICC is disabled thereafter until successful authentication with PUK.</td>
</tr>
<tr>
<td>initialMaxAuthPukFailures</td>
<td>Maximum of successive card holder authentication failures by PUK. PUK-Authentication is disabled thereafter.</td>
</tr>
</tbody>
</table>

3.4.5.3 TObject

The theory \texttt{TObject} just collects the theories that are necessary to define the temporal logic specifications of the objects \texttt{O2 \ldots O12}.

The theory \texttt{TObject} is used by the object components of the policy layer. In addition to theories needed by all components of this layer we need a table of access rights. This table is defined in \texttt{TAccessrights}. 

32 / 153 12. 9. 2000
3.4.5.4 TICC_Objectives

The theory TICC_Objectives is a collection of theories necessary in the temporal logic specifications of the ICC for all layers.

3.4.5.5 TICC_Policy

The theory TICC_Policy is a collection of theories necessary in the temporal logic specifications of the ICC for the policy layers.

3.5 Temporal Logic Specification

3.5.1 Security Objectives Layer

The objectives layer describes the ICC itself reflecting the security objectives provided in the GST [15, Sec. 2.6]. It consists of the temporal logic specification named ICC_Objectives.

The interface of the ICC is modelled by input and output variables. The regular input and output of the ICC is modelled by an input variable channelIn and an output variables channelOut. The values of these variables represent the incoming and outgoing messages to and from the card in an abstract way.

In order to be able to distinguish between a message that is sent once and one that is sent several times a gap has to be modelled which separates two messages. This will be done with the exceptional value noInfo. So we expect the input variable channelIn to change from a value different from noInfo to noInfo and back again, but never change immediately between two values both different from noInfo. The ICC is modelled in such a way that this is also valid for the output variable channelOut.

Three additional boolean input variables represent signals sent to the SigG application:

securityViolation representing a channel on which the application receives hints on security violations. On the real card this can be a combination of different kinds including hardware registers and signals generated by the card operating system.

reset indicating if a reset is being sent to the ICC by the terminal

power indicating if the card is supplied with power.
Besides of the ICC’s interface the state space of the specification ICC_Objectives consists of the relevant secrets and other internal data of the TOE:

\( skCh \)  The cardholder’s key for creating signatures.

\( DisplayMessage \)  Message which should be shown to the user in order to make the SigG compliancy of the IFD apparent (cf. Sec. 4.3).

\( pinCh \)  The Personal Identification Number (or other data used for authentication) of the legitimate cardholder.

\( PUK \)  The Personal Unblocking Key of the legitimate cardholder.

\( secureMsgKeyOut \)  Key for secure messaging: Used for output encryption.

\( secureMsgKeyIn \)  Key for secure messaging: Used for input encryption.

\( secureMsgKeySigOut \)  Key for secure messaging: Used for output signature creation.

\( secureMsgKeySigIn \)  Key for secure messaging: Used for input signature verification.

According to ISO-7816 [3] the communication between the ICC and the terminal is like a question and answer game. The ICC answers to commands from the terminal. If the terminal does not see an answer within a well defined period of time it may try to send the command again or after a few trials reset the card. Commands and answers are both modelled as elements of type \( \text{information} \) that will also be called messages in the following. Since the card is the slave part in the communication it is the responsiblity of the terminal to react if an expected answer is not returned.

Therefore when formalizing the security features we need not put strong time limits on reactions and answers from the card. It generally will suffice to say that if the card sends the answer to a command, the internal state reflects the proper reaction.

The behaviour of the ICC is described according to the security objectives stated in the GST [15]. The corresponding formulas are interpreted in the informal interpretation (cf. Chap. 4). Mostly, the security objectives require to take the trace, i.e. the sequence of transactions leading to the current state into account. As a consequence, their formalization has to be able to look at the relevant parts of the complete
history of the current state. Access to this history is provided via the state variable \texttt{channelOutRawH} keeping all answers of the ICC since its initialization in chronological order.

### 3.5.2 Security Policy Layer

The development methodology of the VSE system suggests a decomposition of formal specifications into several components. Therefore, the specification \texttt{ICC_Policy} is a combined temporal logic specification, i.e. it consists of components that work in pseudo parallel manner. In every state one of these components is active and can perform its state transitions while the other components stutter. A component is said to be enabled if its preconditions hold. In this case such a component is allowed to take a step. Fairness conditions are in force which ensure that every component eventually will take a step provided it is enabled.

While the components of the specification \texttt{ICC_Policy} formalize the authentication state machine [15, Sec. 4.1] and the access control principles [15, Sec. 4.2], the specification \texttt{ICC_Assumption} formalizes what has to assumed about the input data for the ICC in order to formally prove the desired properties. These assumptions are combined with the actual policy into the surrounding specification \texttt{ICC_Environment} by connecting \texttt{ICC_Assumption} with \texttt{ICC_Policy}. This connection works like a filter for the input stream of \texttt{ICC_Policy} allowing the formal proofs to be relied on their essential prerequisites.

All components of the policy layer are depicted in the above diagram and are described in the following paragraphs.
3.5.2.1 ICC_Assumption

This specification formalizes assumptions about the input to the TOE. Its state space thus consists of a single variable named \textit{channelIn} that should be understood as the input channel of the ICC. Note, that it is an output variable from the perspective of the actual component.

The specification formalizes the following two assumptions about the input data of the TOE:

1. We have to assume that the input of the ICC does not contain any secret information (like e.g. the signature creation key) in order to be able to show that no such information can be extracted from the card. Otherwise, if for example the card got the command to sign a document which contained the key, the answer would be a message from which the key could easily be inferred.

2. There are two exceptions to the preceding assumption:
   - the cardholders authentication data
   - new display messages

These data must be transmitted to the ICC as parameters of specific commands. Additionally, we have to assume that changing authentication data or display messages preserves their secrecy. Note, that this actually is a requirement for the cardholder expressing his/her responsibility for the careful handling of such informations.

3.5.2.2 ICC_Environment

This is a combined temporal specification connecting ICC_Policy with ICC_Assumption in order to get an assumption about the input of the TOE. Except for the input channel, which is an internal interface of the combination, its state space consists of variables modeling the interface of the ICC, namely \textit{channelOut}, \textit{channelOutRawH}, \textit{power}, \textit{reset} and \textit{securityViolation}. These variables are merely adopted from ICC_Policy.
The principal duty of the current specification is to connect the unconstrained input variable \( \text{channelIn} \) of \( \text{ICC}_\text{Policy} \) with the assumptions provided by \( \text{ICC}_\text{Assumption} \).

### 3.5.2.3 ICC_Policy

This is the central specification of the policy layer. Its purpose is the combination of all distributed components that contribute to the formalization of the underlying security policy. Its principal duty is to provide the intended connections between these parts of the formal model.

The above diagram only shows a small selection of all connections established by \( \text{ICC}_\text{Policy} \) and the affected components. It illustrates two major aspects of the combined specification:

1. It depicts the external interface of the TOE which forms the state space of \( \text{ICC}_\text{Policy} \) as well as the connections of the individual state variables to those components that are responsible for their processing.

2. It shows the shared variable \( \text{buck} \) which is used to coordinate the control flow between the different components. The value of the \( \text{buck} \) indicates which components are enabled. It is manipulated in a cooperative way by the firing action passing over the control to those components supposed to continue the current trace.
The missing components and their interconnections are described in the remaining paragraphs concerned with the policy layer.

3.5.2.4 The Automaton component

The Automaton component formalizes the authentication state machine and thus is responsible for maintaining the current authentication state (CAS) introduced in the GST [15, Sec. 4.1]. It recognizes security related events [15, Sec. 3.1.2] and reacts on them by changing the variable state accordingly. This variable is observed by the object components that use its value to achieve correct access control decisions.

Furthermore the Automaton component undertakes the task of a scheduler by setting the shared variable buck to appropriate values. The buck indicates which component is supposed to do the next step. Actions which have to wait because they depend on the results from other components are disabled until the buck has the respective value. Each time such an action is finished the control is returned back to the Automaton component which in turn decides how to continue.

The behaviour of the Automaton component is characterized by a normal execution loop of actions. This loop is executed as long as no exceptional event occurs. As soon as a securityViolation, reset or power event is detected the normal execution is stopped. After performing an exceptional sequence of actions handling the initiating event the normal execution is started again from its beginning.

Normal Execution. The normal execution loop could approximately be described in an imperative programming style:

```
WHILE true DO
BEGIN
    call(bckSecChannelIn);
    call(bckObjects);
    getEvent;
    stateChange;
    call(bckObjectReset);
    call(bckSecChannelOut)
END
```
**Exception Handling.** After an exceptional event the following sequence is executed:

```
BEGIN
    stateChange;
    call(bckObjectReset);
    call(bckSecChannelOut)
END
```

We use a program counter $pc$, which enables the respective actions and is incremented each time an action finishes. At the end of the loop (and of the exceptional sequence) it is set back to resume execution at the beginning of the main loop.

### 3.5.2.5 Objects

Each object defined in the generic security target is modelled by one component of ICC_Policy except for object O1. As this object represents the whole SigG application this object can be regarded to be modelled by ICC_Policy itself. In the following description object 1 will be ignored.

Each object consists of a value and operations accessing this value. The target also defines access rights which restrict the access according to the current authentication state. The value of the object has to be protected against access that violates the access rights.

In the model the value of an object is represented by an internal variable called $value$. As this variable is internal there is no direct access to it. It can neither be read nor be modified directly by actions of other components. The only way to get access to the value is through actions of the object component. The actions of the behaviour are only enabled if the following three conditions hold:

1. There is a command in the input variable $command$ corresponding to the action.

2. The current action is allowed according to the access right matrix. Here we use the predicate $allowed$ defined in theory TAccessRights (cf. 3.4.4.2).

3. The state variable $buck$ contains a value allowing the object components to become active. These values are $bckObjects$ enabling ordinary object actions, $bckO3Unblock$ for unblocking the PIN and
\textit{bckObjectReset, bckObjectReset8} and \textit{bckObjectReset9} for special reset actions.

Exceptions to this rule are only actions that do not access the internal variable \textit{value}. They serve to reset the internal state of the component. The object components may produce an output intended to be sent to the terminal. This output is reflected in the output variable \textit{valueOut}. Some objects can detect security related events (e.g. object O3 can detect successful user authentication). Upon detection the event is assigned to a shared variable named \textit{event}. The automaton will read it and set it back to \textit{noInfo} thereafter.

All object components communicate with other components via the following variables (there may be additional variables in certain object component):

**Input variables**

\textit{command}: the input of the card after secure messaging processing. This should be a command from the IFD associated to an action in one of the objects.

**Shared variables**

\textit{buck}: the variable used to synchronise the actions of the various components.

\textit{valueOut}: the value to be sent to the IFD before processing for secure messaging.

Common to all object components are the following actions (in accordance to the security target) though some of them may never be allowed in certain object components:

**Read** The internal \textit{value} is written to the output variable \textit{valueOut}. This action often is replaced by an action called \textit{readUnchanged} in the action list of the canonical behaviour. While \textit{read} is generic and thus only variables occur in its description common to all objects, we have to add a condition concerning the immutability of variables specific to the object in question.

**Modify** The internal \textit{value} is modified to the value submitted in the command read from \textit{command}. Again this action often does not occur directly in the behaviour but is part of another action that also describes immutability of particular variables.
**O2.** The component O2 represents the object O2 as defined in the security target. It provides access operations for the private signature key of the cardholder and stores them.

There are only variables common to all object components in the interface of O2.

Besides the common actions read and modify the object can be used to sign documents. This is modeled by the action

**use:** The signature of the document³ based on the internal variable value as key is computed using the function \( mSig \) and assigned to the output variable valueOut.

³There is no difference between the document and its hash value on this abstraction level
O3  The component O3 represents the object O3 as defined in the security target. It provides access operations for the cardholder reference data and stores them. These data (usually the PIN) are used to authenticate the cardholder.

Communication to the other components is modeled using the following variables in addition to variables common to all object components.

Shared Variables:

- **event**: the security related event resulting from doing user authentication (success or failure). This variable is classified shared because it can be reset by the Automaton after reacting to the event.

Output variables:

- **authChFailures**: the number of authentication failures allowed before the authentication will be blocked.

Besides the common action read (via readUnchanged) the object can be used to perform user authentication. This is modeled by the following actions:

- **use**: The parameter provided with the command read from input variable command is compared to the internal variable value. The appropriate security related event is sent to the automaton by assigning it to the output variable event. Failed authentications are counted.
by decrementing the output variable authChFailures. If it reaches 0 authentication will be blocked. The Automaton component will read authChFailures and use this information to decide whether to enter an authentication state which blocks user authentication.

**useAndModify:** This action performs user authentication based on the first argument of the given command. In addition it changes the PIN in case of successful authentication to the value provided as second argument of command.

**unblock:** If user authentication is blocked (authChFailures = 0), the user can present a reset code (PUK) to unblock the authentication. The PUK is maintained by object O4. If the corresponding object component detects the correct PUK it changes the value of buck to bckO3Unblock which enables this action.

**O4** The component O4 represents the object O4 as defined in the security target. It provides access operations for the cardholder reset code (PUK) and stores them.

The interface of O4 consists of the following variables in addition to variable common to all object components.
Shared Variables:

\[ \text{event: the security related event resulting from doing user authentication (success or failure). This variable is classified shared because it can be reset by the Automaton after reacting to the event.} \]

Besides the common actions \text{read (via readUnchanged) and modify} the object can be used to perform user authentication by reset code. This is modeled by the following action:

\text{use: The parameter provided with the command read from input variable command is compared to the internal variable value. The appropriate security related event is sent to the automaton by assigning it to the output variable event. Failed authentications are counted by decrementing the internal variable chPukFailures. If it reaches 0 authentication by reset code will be blocked. Upon successful authentication the object component O3 is notified by setting the shared variable buck to bekO3Unblock. O3 may then unblock user authentication.}

\text{O5} The component O5 represents the object O5 as defined in the security target. It provides access operations for the signature key certificate of the cardholder and stores them.

The interface of O5 only consists of variable common to all object components.

Besides the common actions \text{read and modify} the object can be used to verify signatures of the cardholder. This is modeled by the action
**use:** The signature of the document provided as argument of the current command is checked based on the cardholder’s signature key certificate stored in the internal variable `value`. An appropriate answer representation is assigned to the output variable `valueOut`.

This action remains under-specified as it is not security relevant.

**O6** The component O6 represents the object O6 as defined in the security target. It provides access operations for the public key of the root certification authority and stores them.

```
  Automaton
     ↓
    Buck
        ↓
   state
       ↓
  O6
     ↓
   (valueOut)
       ↓
SecChannelIn
       ←→
SecChannelOut
```

The interface of O6 only consists of variable common to all object components.

Besides the common actions `read` and `modify` the object can be used to verify certificates. This is modeled by the action

**use:** The certificate provided as argument of the current command is checked based on the public key of the root certification authority stored in the internal variable `value`. An appropriate answer representation is assigned to the output variable `valueOut`. 
**O7** The component O7 represents the object O7 as defined in the security target. It provides access operations for the **other credentials for signature verification** and stores them.

The interface of O7 only consists of variable common to all object components.

Besides the common actions **read** and **modify** the object can be used to verify signatures of persons different from the cardholder. This is modeled by the action **use:**

The signature provided as argument of the current command is checked based on the public keys stored in the internal variable **value**. An appropriate answer representation is assigned to the output variable **valueOut**.
**O8** The component O8 represents the object O8 as defined in the security target. It provides access operations for the private device authentication key of the TOE and stores them.

The interface of O8 consists of the following variables in addition to variable common to all object components.

**Output Variables:**

- **secureMsgKeySigOut** secret key for signing messages to the IFD according to secure messaging. It is generated during device authentication.
- **secureMsgKeyIn** secret key used to decrypt messages from the IFD. It is generated during device authentication.

Besides the common actions read and modify the object can be used to answer challenges during device authentication. This is modeled by the action

**use:** A certificate and the answer to the challenge received as parameter of the IFD command (read from variable command) is generated as output to be sent to the IFD (written to valueOut). In order to establish secure messaging one key pair for signatures and one for encryption is generated. The public keys are attached to the combined certificate/answer to challenge. The secret keys are forwarded to the components responsible for secure messaging.
The component O9 represents the object O9 as defined in the security target. It provides access operations for the public key of the root certification authority for device authentication and stores them.

The interface of O9 consists of the following variables in addition to variable common to all object components.

**Shared Variables:**

- **event**: the security related event resulting from doing authentication of the IFD (success/failure). This variable is shared because it can be reset by the Automaton after reacting to the event.

**Output Variables:**

- **secureMsgKeyOut** public key for encryption of secure messages leaving the ICC. It is received from the IFD during device authentication.
- **secureMsgKeySigIn** public key used to check secure messaging signatures from the IFD. It is received during device authentication from the IFD.

Besides the common actions read and modify (via readUnchanged and modifyUnchanged) the object can be used to verify certificates of the IFD. This is modeled by the action...
**use:** The certificate provided as argument of the current command is checked based on the public keys stored in the internal variable `value`. The appropriate event constant is assigned to the output variable `event`. Keys attached to the certificate are forwarded to the components responsible for secure messaging (SecChannelIn and SecChannelOut).

**O10** The component O10 represents the object O10 as defined in the security target. It provides access operations for the *device authentication credentials of the TOE* and stores them.

![Diagram](image)

The interface of O10 only consists of variable common to all object components.

Only the common actions *read* and *modify* are defined for this object.
O11 The component O11 represents the object O11 as defined in the security target. It provides access operations for the cardholder display message and stores them.

The interface of O11 only consists of variable common to all object components.
Only the common actions read and modify are defined for this object.
The component $O_{12}$ represents the object $O_{12}$ as defined in the security target. It provides access operations for the public signature key of the cardholder and stores them.

The interface of $O_{12}$ only consists of variable common to all object components.

Besides the common actions read and modify the object can be used to verify signatures of the cardholder. This is modeled by the action

**use:** The signature of the document provided as argument of the current command is checked based on the cardholder’s public signature key stored in the internal variable $value$. An appropriate answer representation is assigned to the output variable $valueOut$.

This action remains under-specified as it is not security relevant.
3.5.2.6 The SecChannelIn and SecChannelOut components

Input and output of the TOE never enter or leave the object components or the Automaton component directly. All input and output passes through the SecChannelIn resp. SecChannelOut component. These components are responsible for secure messaging. Outgoing messages are signed and encrypted, incoming messages are decrypted and their signatures checked. If secure messaging is not in force (e.g. before mutual device authentication) then these components only forward input and output values.

SecChannelIn. The SecChannelIn component performs a loop containing two actions. The first action waitForNoInformation simply waits until the input is equal to noInfo. Input is only read thereafter when the input has changed to an ordinary information value. This prevents the same input from being processed twice.
SecChannelOut. The SecChannelOut component performs a loop which is complementary to that performed by SecChannelIn. It consists of an operation which encodes the output produced by an object or the Automaton component followed by an action which sets the output variable channelOut back to noInfo.

3.5.2.7 The DataLink component

Some of the communication relations are of a one-to-many kind, i.e. they involve one sender and many receivers. The appropriate way to model such a relation in VSE is to introduce a component acting as a distributor. The component DataLink receives data on its input variables and distributes them to several output variables. This allows us to use output variables in sending components and input variables in receiving components which are connected via the DataLink component.
Chapter 4

Informal Interpretation

The security objectives are formalized in the objectives layer of the FMSP. For each objective one or more temporal logic formulas are provided.

4.1 Prevent extraction or modification of the SigG private signature key of the cardholder (SO1)

SO1 is informally specified in the GST [15, Sec. 2.6.1] as:

(SO1) The TOE ensures the confidentiality and the integrity of the SigG private signature key of the cardholder stored in the TOE with two aspects: [cf. SO1.1 and SO1.2].

4.1.1 SO1.1

SO1.1 is informally specified in the GST [15, Sec. 2.6.1] as:

(SO1.1) The TOE shall prevent any kind of extraction of the cardholder's secret key from the ICC.

It corresponds to the following temporal logic formula specified in ICC_Objectives:

\[
\Box \neg \text{mInferable}(\text{channelOut}, \text{def}(\text{skCh}))
\]

Extracting the key is only possible, if there is an output from the card that allows somebody to infer the secret key. Therefore, the formalization requires that at any time the secret key (skCh) cannot be inferred from the output of the ICC.
4.1.2 SO1.2

SO1.2 is informally specified in the GST [15, Sec. 2.6.1] as:

The TOE shall prevent any kind of modification of the cardholder’s secret key in the ICC.

It corresponds to the following temporal logic formula specified in ICC_Objectives:

\[ \Box \ skCh = skCh' \]

The formalization requires that at any time the secret key \( skCh \) is equal to the value of the secret key in the next state. Thus, the secret key can never change.

**FMSP Usage Hint 1**

*If an ICC is designed to support authorized changes of the cardholder’s signature key, i.e. SO1.2 is relaxed, the formalization has to be adapted to the actual needs.*

4.2 Prevent unauthorised use of the SigG digital signature funktion (SO2)

SO2 is informally specified in the GST [15, Sec. 2.6.2] as:

The TOE shall allow the use of the digital signature function only to the cardholder. This security objective has the following aspects: [cf. SO2.1 – SO2.3]

4.2.1 SO2.1

SO2.1 is informally specified in the GST [15, Sec. 2.6.2] as:

The TOE shall allow the use of the digital signature function only to the cardholder after successful authentication by knowledge.

It corresponds to the following temporal logic formula specified in ICC_Objectives:
\[ \text{mInferableWithout}(\text{channelOut}, \text{mSig}(i, \text{def}(\text{skCh})), \text{def}(\text{skCh})) \rightarrow \text{histAuthUser}(\text{channelOutRawH}) \]

The use of the digital signature function is observable, if a piece of data appears in the output of the ICC in any form that allows the signed message to be inferred. This will be expressed with the predicate \text{mInferableWithout}^{[1]}.

The actual user may only be considered as the cardholder after the IFD has sent the correct PIN and the authentication has not expired yet because of closing the signature application or resetting the ICC. This can be checked by inspecting the history of outputs recorded in \text{channelOutRawH}.

### 4.2.2 SO2.2

SO2.2 is informally specified in the GST [15, Sec. 2.6.2] as:

\[(\text{SO2.2})\]

Successive authentication failures will be interpreted as an attempted unauthorised access by the TOE and will disable the signature function.

It corresponds to the following temporal logic formula specified in \text{ICC_Objectives}:

\[ [] \text{first} (\text{channelOutRawH}) = \text{value} (\text{answerAuthSuccess}) \rightarrow \neg \text{maxFailuresExceeded} (\text{rest}(\text{channelOutRawH})) \]

The history variable \text{channelOutRawH} contains a list of all messages the card has sent as output. As both authentication failures and successes are notified by the ICC, inspecting the appropriate answers in the history variable reveals the amount of successive failures. The predicate \text{maxFailuresExceeded} becomes true if its parameter contains a sequence of successive authentication failures which exceed the maximal amount.

The formula states that whenever the ICC signals a successful user authentication the TOE hasn’t detected an attempted unauthorised access. Together with SO2.1 this formalisation implies that the signature is disabled if the maximal amount of authorization failures is exceeded.

\footnote{For the reason, why \text{mInferableWithout} is used instead of \text{mInferable} see Sec. 3.4.2.2.}
4.2.3 SO2.3

SO2.3 is informally specified in the GST [15, Sec. 2.6.2] as:

The authentication data is stored in the TOE and shall not be disclosed.  

It corresponds to the following temporal logic formulas specified in ICC_Objectives:

\[
\begin{align*}
[] \neg m\text{Inferable}(\text{channelOut}, \text{def(pinCh)}); \\
[] \neg m\text{Inferable}(\text{channelOut}, \text{def(PUK)})
\end{align*}
\]

Analogously to SO1.2 the formalization requires from the TOE that the cardholder’s authentication can never be extracted from the output of the ICC.

4.3 Verification of the SigG accreditation of the IFD (SO3)

This security objective is relevant in case of the activation of Option Public IFD only.

4.3.1 SO3.1

SO3.1 is informally specified in the GST [15, Sec. 2.6.3] as:

The TOE shall be able to verify the SigG accreditation of the IFD.

This security objective doesn’t impose any direct requirement to the TOE except that the functionality to verify SigG accreditation of the actual IFD should be implemented. Due to fundamental principles of logic, properties like the presence or absence of some functionality cannot be expressed together with its own axiomatization in the same framework.

Therefore, SO3.1 doesn’t correspond to any formally specified requirement in ICC_Objectives.

\[\text{Otherwise the semantics of the resulting logic would be paradoxical in that it were capable of modifying its own reasoning mechanisms.}\]
FMSP Usage Hint 2
When adding specific requirements on the mutual authentication process of IFD and ICC concerning e.g. the type of certificates exchanged or the generation of keys for the secure messaging the FMSP has to be adapted accordingly.

4.3.2 SO3.2

SO3.2 is informally specified in the GST [15, Sec. 2.6.3] as:

(SO3.2) If the TOE verified successfully the SigG accreditation of a IFD then the TOE grants access to the display message stored in the TOE.

It corresponds to the following temporal logic formula specified in ICC_Objectives:

\[
[] \text{mInferable(channelOut, def(DisplayMessage))} \\
\quad \rightarrow \text{histAuthIfd(channelOutRawH) OR histAuthUser(channelOutRawH)}
\]

According to the DIN standard [5] the display message has to be requested by the IFD. Thus it is the responsibility of the IFD to take appropriate actions if the display message cannot be read from the ICC for any reason (cf. [3]).

Therefore, it is sufficient to require that the display message is sent by the ICC only if the authentication of the IFD was successful (which can be revealed by inspecting the output history). Otherwise it could not be used as an evidence for the SigG compliance of the IFD.

Unfortunately, this property has to be relaxed since when used with an office IFD the TOE grants access to the display message only on the basis of user authentication.

FMSP Usage Hint 3 The security objective SO5.2 is formulated in a rather general way. When adding specific requirements on the user authentication process concerning e.g. the preconditions for accessing the cardholder authentication data the FMSP has to be adapted accordingly.
4.4 Demonstrate the TOE’s SigG accreditation to a SigG accredited IFD (SO4)

This security objective is relevant in case of the activation of Option Public IFD only. SO4 is informally specified in the GST [15, Sec. 2.6.4] as:

The TOE shall be able to prove its SigG accreditation to a SigG accredited IFD.

Analogously to SO3.1 this security objective doesn’t impose any direct requirement to the TOE except that the functionality to prove its SigG accreditation should be implemented. For the same reasons already discussed in Sec. 4.3.1 SO4 doesn’t correspond to any formally specified requirement in ICC_Objectives.

FMSP Usage Hint 4

When adding specific requirements on the mutual authentication process of IFD and ICC concerning e.g. the type of certificates exchanged or the generation of keys for the secure messaging the FMSP has to be adapted accordingly.

4.5 Prevent unauthorised disclosure and modification of communication with the IFD (SO5)

This security objective is relevant in case of the activation of Option Public IFD only.

4.5.1 SO5.1

SO5.1 is informally specified in the GST [15, Sec. 2.6.5] as:

The TOE shall enforce secure messaging between the ICC and the successfully authenticated SigG accredited public IFD preventing the disclosure of sensitive data. The sensitive data to be protected are at least (i) the verification data of the cardholder, (ii) the reference data of the cardholder transmitted from the IFD to the ICC and (iii) the display message transmitted from the ICC to the IFD.
It corresponds to the following temporal logic formula specified in ICC_Objectives:

\[
[] \forall sk,i: \\
\neg \text{channelOut}' = \text{channelOut} \land \\
\neg \text{histAuthIFD}(\text{channelOutRawH}) \land \\
\neg \text{validSecureMsgPair}(sk, \text{secureMsgKeyOut}) \land \\
\neg \text{isIccSensitive}(i) \\
\rightarrow \neg \text{mInferableWithout}(\text{channelOut}, i, sk);
\]

Informations may leave the ICC on secure channels only if an authenticated IFD is used. In this case sensitive data must not be sent as clear text. Therefore, the formalization requires that whenever secure messaging is activated it is impossible to infer any sensitive information from the output just generated by the ICC.

Due to fact that the symmetric case for encryption of data received from the IFD is not part of the requirements for the ICC, it cannot be modelled in the present FMSP.

4.5.2 SO5.2

SO5.2 is informally specified in the GST [1.5, Sec. 2.6.5] as:

\[(SO5.2)\]

The TOE shall enforce secure messaging between the TOE and the public IFD for all commands of the IFD and responses of the ICC preventing any unnoticed modification of the data in transfer.

It corresponds to the following temporal logic formula specified in ICC_Objectives:

\[
[] \neg \text{channelOut}' = \text{noInfo} \land \\
\neg \text{channelOut}' = \text{channelOut} \land \\
\neg \text{secureMsgKeyOut} = \text{noKey} \land \\
\neg \text{secureMsgKeySigOut} = \text{noKey} \\
\rightarrow \exists i: \text{channelOut}' = \\
\quad \text{secureMsgEncode}(i, \text{secureMsgKeyOut}, \text{secureMsgKeySigOut})
\]

Whenever secure messaging is active the ICC should (encrypt and) sign every outgoing message with the help of the secureMsgEncode function. Due to the cryptographic properties of the signature the IFD is able to notice any modification to the data received from the ICC.
FMSP Usage Hint 5
In the symmetric case the ICC is required to check the integrity of the data received from the IFD by verifying their signature. However, since SO5.2 doesn’t specify any direct requirement for the ICC on how to react to a detected modification, the FMSP doesn’t contain any formula concerning the input of the ICC. When adding such requirements to SO5.2 the FMSP has to be adapted.
### 4.6 Quality of key generation (SO6)

This security objective is relevant in case of the activation of **Option Generation of Signing Key Pairs on the ICC only**.

Any key material generated by the TOE shall bear a strong cryptographic quality. The cryptographic quality is characterised as follows:

1. If secret keys are generated either in the personalisation phase or in operational use by the TOE this process shall be performed in a confidential way.
2. The secret keys generated by the TOE shall be unique with a very high probability and cryptographic strong.
3. It shall be impossible to calculate the secret key from the public key.

The FMSP describes the behaviour of the ICC in operational use only and does not deal with the process of personalisation. This is in accordance with the underlying security policy described in the GST [15, Sec. 4] since key generation is not considered in the specification of the state machine [15, Sec. 4.1]. Moreover the access type generation to O2 is completely denied by the TOE [15, Sec. 4.2].

Therefore, the only requirement to be met during operational use is that \( skCh \) must never change. This already is stated and formalized in SO1.2 (cf. 4.1.2).

**FMSP Usage Hint 6** For ICCs which allow the cardholder’s signature key to be changed the FMSP has to be adapted.

- If the signature key is generated outside the ICC a class of subjects must be defined, that can generate the keys and the policy layer must be enhanced in order to specify the secure key transfer to the ICC. Moreover, requirements must be added to the objectives layer guaranteeing that the signature key can only be changed in a trustworthy way. Probably, in order to prove these new requirements some assumptions on the ICCs input have to be added.

- If the signature key is generated inside the ICC mainly the state machine in the policy layer has to be altered accordingly. Additionally, a formalization of the new security features has to be integrated in the objectives layer.
4.7 Provide secure digital signature (SO7)

According to the GST [15] this is the principal security objective of the TOE.

4.7.1 SO7.1

SO7.1 is informally specified in the GST [15, Sec. 2.6.7] as:

The TOE provides a function to generate a SigG digital signature for the data presented by the IFD using the SigG private signature key of the cardholder stored in the TOE.

It corresponds to the following temporal logic formula specified in ICC_Objectives:

\[
\begin{align*}
\Box \ \text{validpair}(isk, ipk) & \ \land \\
\neg \text{channelOut}' & = \text{channelOut} \ \land \\
\text{minferable}(\text{channelOut}', \text{def(sig}(j,isk)), \text{def}(isk)) & \ \rightarrow \\
\text{channelInDecoded} & = \text{ifdDoSign}(j) \ \land \\
\text{skCh} & = isk
\end{align*}
\]

Whenever a piece of information \( j \) signed with valid secret key \( isk \) can be inferred from the output just generated by the ICC then the TOE should have received a request for signing this data from the IFD and the signature key used is cardholder’s secret key.

Note, that it makes no sense to require that every signing request from an IFD should eventually be answered with a resp. signature due to the restrictions imposed by user (and IFD) authentication.

4.7.2 SO7.2

SO7.2 is informally specified in the GST [15, Sec. 2.6.7] as:

The function to generate a SigG digital signature works in a manner that other individuals not possessing SigG private signature key of the cardholder cannot generate the signature.

This security objective describes a cryptographic property of the signature function itself. It is by no means correlated to the behaviour of the TOE. Therefore, it doesn’t correspond to any formally specified requirement in ICC_Objectives.
4.8 React to potential security violations (SO8)

4.8.1 SO8.1

SO8.1 is informally specified in the GST [15, Sec. 2.6.8] as:

\[(SO8.1)\]

The TOE reacts to potential security violations, which are (i) recognised and signalled to the TOE by the underlying hardware or (ii) identified by the TOE itself.

The detection of a security violation does not have a direct impact on output variables. Therefore SO8.1 doesn’t correspond to any formally specified requirements in ICC_Objectives.

**FMSP Usage Hint 7** SO8.1 doesn’t specify which are the potential security violations the TOE should recognize. However, the state machine as a part of the underlying security policy [15, Sec. 4.1] formalized in the policy layer (cf. \[5.2\]) identifies several transitions each indicating a potential security violation. When changing SO8.1 in order to include requirements on recognizable security violations, both the policy and the objectives layer of the FMSP have to be adapted.

4.8.2 SO8.2

SO8.2 is informally specified in the GST [15, Sec. 2.6.8] as:

\[(SO8.2)\]

If a potential security violation is detected then
1. the TOE reaches a secure blocking state disabling all application of the ICC and
2. the blocking state is made apparent to the user.

Due to the generic nature of the security target [15] this security objective is incompletely specified. The sponsor is demanded in [15, 2.6.8] to precisely state how the secure blocking state of the TOE is characterized and how this state is made apparent to the user. Therefore, SO8.2 doesn’t correspond to any formally specified requirements in ICC_Objectives.

**FMSP Usage Hint 8** As already discussed in FMSP usage hint 7 the policy layer indicates several potential security violations. Moreover,
it specifies transitions to CAS6 (Potential attacker) which represents the secure blocking state of the TOE. This is in accordance with the GST (cf. [13, Sec.4.1]). When changing these transitions the policy layer (cf. 3.5.2) has to be adapted. Additionally, when adding the missing requirements on the secure blocking state to SO8.2 the objectives layer has to be adapted.
Bibliography


Appendix A

The VSE model

The complete formal specification is provided in a separate file as a textual representation of the VSE development graph. This representation can be reimported in the VSE tool where it can be inspected and modified. For the sake of completeness it is also appended below.

```c
/*############################################################*/
/* */
/* Generic Formal Model of the Security Policy */
/* for Signature Creation SmartCards */
/* */
/* Version 1.1 */
/* September 12th, 2000 */
/* */
/*############################################################*/

#GRAPH Basics
#POS -389,770
BASIC TAnswer
PURPOSE "Definition of constants for answers generated by the card."

answer =
    /* Answer to the modify command: */
    aModified |
    /* .. to a successfully completed command (OK response): */
    aSuccess |
    /* .. to a successfully completed authentication PIN command (OK response): */
    aAuthSuccess |
    /* .. to an Expiration event: */
    aAuthExpired |
    /* .. to a successfully completed authentication PUK command (OK response): */
    aAuthPUKSuccess |
    /* .. to a failed command: */
    aAuthIfdSuccess |
    /* .. to a failed command: */
    aFailure |
    /* .. to a failed authentication command: */
    aAuthFailure |
```
/* .. to a fatally failed command: */
aFatalFailure |
/* .. to a command not allowed in the current state */
aDenied |
/* .. to a power up reset */
aReset |
/* .. after closing the signature application */
aClosed |
/* .. after opening the signature application */
aOpened |
/* .. to an unsuccessful user authentication because of 
a disabled card */
aDisabled |
/* .. to security violations */
iSecurityViolation

BASICEND

#POS 250,770
BASIC TBucks

PURPOSE " Theory of control values to be assigned to a 
shared variable in order to coordinate the 
components of ICC_FUNCTION. "

/* One constants of type bucks for each component of ICC_FUNCTION 
is defined */
bucks =
bckSecChannelIn |
bckObjects |
bckAutomaton |
bckSecChannelOut |
bckObjectReset |
bckObjectReset8 |
bckObjectReset9 |
bckO3Unblock |
bckSecChannelOutReady

BASICEND

#POS 100,770
BASIC TCommand

PURPOSE "Definition of constants for commands generated by the IFD."

USING TInformation

ifdCommand =
/* Commands sent by the IFD */
/* Command to sign the information given as 
parameter \1 */
Sign(signDocument: information) WITH isSign |
/* Command to do User Authentication with PIN, 
\1 is the PIN */
Auth(authPin: information) WITH isAuth |
/* Command to do User Authentication with PUK, 
\1 is the PIN, \2 the new PIN */
PukAuth(pukAuthPuk: information, pukAuthPin: information) WITH isPukAuth |
/* Command to do User Authentication and change 
the password, \1 is the old PIN, \2 the new

12.9.2000
one */
AuthChange(authChangeOld: information, authChangeNew: information) WITH isAuthChange |
/* Command to change the display message */
MessageChange(MessageChangeMsg: information) WITH isMessageChange |
/* Command to authenticate the ICC */
IntAuth(intAuthChallenge: counter) WITH isIntAuth |
/* Command to authenticate the IFD */
ExtAuth(extAuthCert: information) WITH isExtAuth |
/* Command to exit (close) the SigG application */
Exit WITH isExit |
/* Command to start (open) the SigG application */
Start WITH isStart |
/* Command to do signature verification
with key reference \1 on signed message \2 */
Verify(verifyRef: information, verifyDocument: information) WITH isVerify |
/* Commands to modify objects */
ModifyObj2(modifyObj2Document: information) WITH isModifyObj2 |
ModifyObj3(modifyObj3Document: information) WITH isModifyObj3 |
ModifyObj4(modifyObj4Document: information) WITH isModifyObj4 |
ModifyObj5(modifyObj5Document: information) WITH isModifyObj5 |
ModifyObj6(modifyObj6Document: information) WITH isModifyObj6 |
ModifyObj7(modifyObj7Document: information) WITH isModifyObj7 |
ModifyObj8(modifyObj8Document: information) WITH isModifyObj8 |
ModifyObj9(modifyObj9Document: information) WITH isModifyObj9 |
ModifyObj10(modifyObj10Document: information) WITH isModifyObj10 |
ModifyObj11(modifyObj11Document: information) WITH isModifyObj11 |
ModifyObj12(modifyObj12Document: information) WITH isModifyObj12 |
/* Commands to read Objects */
ReadObj2 WITH isReadObj2 |
ReadObj3 WITH isReadObj3 |
ReadObj4 WITH isReadObj4 |
ReadObj5 WITH isReadObj5 |
ReadObj6 WITH isReadObj6 |
ReadObj7 WITH isReadObj7 |
ReadObj8 WITH isReadObj8 |
ReadObj9 WITH isReadObj9 |
ReadObj10 WITH isReadObj10 |
ReadObj11 WITH isReadObj11 |
ReadObj12 WITH isReadObj12

BASICEND

#POS -320,600
THEORY TElement
PURPOSE " Theory for the PARAMS slot of TMaybe "
TYPES element
THEORYEND

#POS 49,770
BASIC TEvent
PURPOSE " Theory of security related events "
/* The security related events as defined in the security functions/claims. */
events =
sre1 | sre2 | sre3a | sre3b | sre4 | sre5 | sre6 | sre7 |
sre8 | sre9 | sre10 | sre11 | sre12

BASICEND

#POS -390,680
BASIC TList
PARAMS TElement
List = nil WITH isEmpty |
cons(first : element,rest : list) WITH isNotEmpty
BASICEND

#POS -250,680
BASIC TMaybe
PURPOSE "Generic specification which adds a special constant
for undefined values to a given theory"
PARAMS TElement
m = undef WITH isUndef |
def(value: element) WITH isDef
BASICEND

#POS -100,680
BASIC TObjectIds
PURPOSE "Theory of ObjectIds"
objectIds = obj1 | obj2 | obj3 | obj4 |
obj5 | obj6 | obj7 | obj8 | obj9 |
obj10 | obj11 | obj12
BASICEND

#POS 50,680
BASIC TStates
PURPOSE "Theory of states"
/* States as defined in the security claims/functions */
states = cas1 | cas2 | cas3 |
cas4 | cas5 | cas6 | cas7 | cas8
BASICEND

#GRAPH Main
#POS 75,260
TLSPEC Automaton
PURPOSE
" Specification of the Automaton component.
This component represents the controlflow. "
USING
natural;
TAutomaton
DATA
/* ------------------------------- */
/* INPUT
------------------------------- */
IN
/*:-------------------------------------------:*/
Generic Formal Model of Security Policy
for Signature Creation SmartCards

/* Inputs from SecChannelIn (Secure messaging) */
/* Informations arriving as inputs to the ICC after decoding
  according to secure messaging. */
channelInDecoded: info.m

IN
/* Inputs from objects */
/* from O3: failure counter */
authChFailures: nat

IN
/* Inputs from outside of the ICC */
/* These inputs are signals that require spontaneous reactions. */
/* Security violations that are implementation dependent are reflected by the boolean variable securityViolation.
  These violations can include those detected by the operating system of the card or the hardware.
  securityViolation becomes t if a security violation has been detected. */
securityViolation: bool;

/* The state of power supply.
  It is t, iff the card is being supplied with power. */
power: bool;

/* Reset signal.
  It is t, iff the terminal raises a reset. */
reset: bool

/* ----------------------------- */
/* OUTPUT */
/* ----------------------------- */

OUT
/* The current authentication state as defined by security claims/functions. */
state: states

/* ----------------------------- */
/* SHARED */
/* ----------------------------- */

SHARED INOUT
/* The variable which is used to establish the control flow among the components. Its value determines which
of the components are enabled. */
buck: bucks;

/* Messages from the automaton to the terminal
(To inform it about security related state changes)
This messages are sent to secChannelOut which is responsible for all outgoing messages.

channelOutRaw: info.m

/* History variable for the above variable */
channelOutRawH: list

SHARED INOUT

/* From objects */
/* from O3: Result of cardholder authentication */
authChResult: event.m;
/* from O4: Result of PUK authentication */
authPUKResult: event.m;
/* from O9: Result of IFD authentication */
authIfdResult: event.m

/* ---------------------------------------------------------- */
/* INTERNAL */
/* ---------------------------------------------------------- */
INTERNAL
/* Current security related events as defined by security claims/functions */
event: event.m;
/* Program counter */
pc: nat

VARS dat,i,j,k : info.m;
c: counter

ACTIONS

/* ---------------------------------------------------------- */
/* raiseException */
/* ---------------------------------------------------------- */
/* Raise an exception: 
- disable other components by setting buck 
- jump out of the ordinary program loop by setting PC */
raiseException ::= pc' = 10 AND buck' = bckAutomaton AND
UNCHANGED(channelOutRaw, state, channelOutRawH )

/* ---------------------------------------------------------- */
/* getUrgentEvent */
/* ---------------------------------------------------------- */
/* Reaction on urgent events: security Violation, reset and power failure. 
This action gets enabled if an event occurs that is signalled by one of the boolean input variables. It assigns the corresponding event code to the variable event, which is read by the action stateChange to compute the current...
authentication state.
This action also performs raiseException in order to
stop the normal loop of this component’s main action
normalExectuion and disable all other components
by setting buck. */

getUrgentEvent ::= pc < 10 AND /* disable while an event is being handled */
(power = f AND
  event' = def(sre2) AND
  buck' = bckAutomaton AND
  raiseException
OR
  reset = t AND securityViolation = f AND
  event' = def(sre1) AND
  buck' = bckAutomaton AND
  raiseException
OR
  securityViolation = t AND power = t AND
  event' = def(sre10) AND
  raiseException)
AND
authChResult' = noEvent AND
authPUKResult' = noEvent AND
authIfdResult' = noEvent
/* ---------------------------------------------------------- */
/* noUrgentEvent */
/* ---------------------------------------------------------- */
/* Absence of an urgent event.
This action is used as a (pre)condition in other
actions (it contains no primed variables).
It describes states in which no urgent event occurs. */
/* noUrgentEvent ::= power = t AND reset = f AND
/* Security violation is detected only once. If in
state cas6 detection has already taken place */
(securityViolation = f OR state = cas6)
/* ---------------------------------------------------------- */
/* handlePc */
/* ---------------------------------------------------------- */
/* Check the program counter and increment it. */
handlePc(pCntr: IN nat) ::= pc = pCntr AND pc' = pc + 1
/* ---------------------------------------------------------- */
/* handlePcNormal */
/* ---------------------------------------------------------- */
/* Check the program counter and increment it
provided no urgent event has occurred */
handlePcNormal(pCntr: IN nat) ::= noUrgentEvent AND handlePc(pCntr)
/* ---------------------------------------------------------- */
/* call */
/* ---------------------------------------------------------- */
(Certain actions of) other components need to be synchronised with the automaton and with each other. Their actions are enabled only if the variable ‘buck’ shared among them has a specific value. This action models the transfer of control to another component by setting buck. The automaton only may do so if it currently has the control. */
call(bckNew: IN bucks) ::= 
buck = bckAutomaton AND 
buck’ = bckNew AND 
UNCHANGED(channelOutRaw, state, event, authChResult, 
authPUKResult, authIfdResult, channelOutRawH)

/* GetEvent */
/* ---------------------------------------------------------- */
/* Compute the current event code. 
This action blocks (is disabled) until it can detect an event. 
The event code is then assigned to the internal variable event, which will be used by the action stateChange to compute the next authentication state. 
This action only is responsible for events that do not need immediate reactions. The events which do are detected by the action getUrgentEvent. 

NOTE (USAGE HINT): sre8 does not occur in this action, because it is related to authentication expiration. Its conditions may be defined by the implementor in detail. 
This would lead to new clauses in this action having event’ = def(sre8) in the then part. 
For now the event sre8 does not happen. Despite this the authentication may expire if sre1, sre2, sre4, sre4, sre6 or sre7 are triggered. */
getEvent ::= 
buck = bckAutomaton AND 
buck’ = bckAutomaton AND 
/* Expiration of CH authentication. */ 
IF channelInDecoded = ifdDoExit THEN 
  event’ = def(sre4) /* Closing of the signature application */
ELSE 
  /* A command for another application arrives. */ 
  IF channelInDecoded /= noInfo AND NOT isSigGCommand(channelInDecoded) THEN 
    event’ = def(sre4) /* non signature activity -> close appl. */
  ELSE 
    IF channelInDecoded = ifdDoStart THEN 
      IF AuthChFailures > 0 THEN 
        event’ = def(sre3a) /* Opening of the application with user auth. allowed */
      ELSE 
        event’ = def(sre3b) /* Opening of the application with user auth. blocked */
      FI 
    ELSE 
      /* Cardholder authentication:
The Object O3 returns the event as a result or noEvent */
IF authChResult /= noEvent THEN
  event' = authChResult /* sre6, sre7 */
ELSE
 /* User authentication with PUK:
The Object O4 returns the event as a result or noEvent */
IF authPUKResult /= noEvent THEN
  event' = authPUKResult /* sre1, sre2, sre3 */
ELSE
 /* IFD Authentication:
The Object O9 returns the event as a result */
IF NOT authIfdResult = noEvent THEN
  event' = authIfdResult /* sre9 */
ELSE
  event' = noEvent
FI FI FI FI FI FI AND authChResult' = noEvent AND authPUKResult' = noEvent AND authIfdResult' = noEvent AND UNCHANGED(state, channelOutRaw, channelOutRawH)

/* ---------------------------------------------------------- */
/* stateChange */
/* stateChange ::= */
stateChange ::=
  event' = noEvent AND
  state' = nextstate(state, event) AND
  buck = bckAutomaton AND
  buck' = bckAutomaton AND
  IF eventAnswer(event) /= noinfo THEN
    channelOutRaw' = eventAnswer(event)
  ELSE
    UNCHANGED(channelOutRaw)
  FI AND
  IF isDef(channelOutRaw') THEN
    channelOutRawH' = cons(value(channelOutRaw'), channelOutRawH)
  ELSE
    UNCHANGED(channelOutRawH)
  FI AND
  UNCHANGED(authChResult, authPUKResult, authIfdResult)

/* ---------------------------------------------------------- */
/* normalExectuion */
/* normalExectuion ::= */
/* normalExectuion ::=
  this action represents a main loop of subactions which are executed by the automaton as long as no exceptional events occur (security violation, reset and power events) and a sequence of subaction performed after the normal has been interrupt by an exceptional event.
The main loop could be approximately described like this:

```plaintext
WHILE true DO
BEGIN
    call(bckSecChannelIn);
    call(bckObjects);
    getEvent;
    stateChange;
    call(bckObjectReset);
    call(bckSecChannelOut)
END
```

After an exceptional event the following sequence is executed:

```plaintext
BEGIN
    stateChange;
    call(bckObjectReset);
    call(bckSecChannelOut)
END
```

We use a program counter PC, which enables the respective subactions and is incremented each time a subaction finishes. At the end of the loop (and of the exceptional sequence) it is set back to resume execution at the beginning of the main loop.

Note that handlePcNormal is enabled only if no exceptional event is currently occurring. Otherwise getUrgentEvent becomes the only enabled action which will set the program counter to 10. This enables the exceptional sequence.

```plaintext
/*

normalExecution ::= 
handlePcNormal(0) AND call(bckSecChannelIn) OR
handlePcNormal(1) AND call(bckObjects) OR
handlePcNormal(2) AND getEvent OR
handlePcNormal(3) AND call(bckSecChannelOutReady) OR
handlePcNormal(4) AND stateChange OR
handlePcNormal(5) AND call(bckObjectReset) OR
handlePcNormal(6) AND call(bckSecChannelOut) OR
pc = 7 AND pc' = 0 AND UNCHANGED(channelOutRaw, state, event, buck,
    authChResult, authPUKResult, authIfdResult,
    channelOutRawH)
OR
handlePc(10) AND call(bckSecChannelOutReady) OR
handlePc(11) AND stateChange OR
handlePc(12) AND call(bckObjectReset) OR
handlePc(13) AND call(bckSecChannelOut) OR
pc = 14 AND pc' = 0 AND UNCHANGED(channelOutRaw, state, event, buck,
    authChResult, authPUKResult, authIfdResult,
    dResult,
    channelOutRawH)

/* Dummy action in order to create a mappable parameter variable. This action can be ignored. Its only purpose is to circumvent a bug in the specification tool. It is not used anywhere in this specification */
*/
```
dummy(sOut: OUT subject) ::= true

SPEC
INITIAL
  event = noEvent AND
  state = cas1 AND
  channelOutRaw = noInfo AND
  channelOutRawH = nil AND
  buck = bckAutomaton AND
  pc = 0
TRANSITIONS [ normalExecution, getUrgentEvent ] (state, event, pc, buck, channelOutRaw,
  authChResult, authPUKResult, authIfdResult,
  channelOutRawH }
FAIRNESS
  WF(normalExecution) {state, event, pc, buck, channelOutRaw,
  authChResult, authPUKResult, authIfdResult, channelOutRawH},
  WF(getUrgentEvent) {state, event, pc, buck, channelOutRaw,
  authChResult, authPUKResult, authIf-
  dResult, channelOutRawH}
HIDE event, pc

TLSPECEND

#POS 70,345

TLSPEC DataLink
  PURPOSE "Defines data connections between
  the various components of ICC_Policy"

USING TICC_Policy; TTransitions

DATA
/* ************************************************** */
/* SHARED */
/* ************************************************** */

/* The Output of the TOE before preprocessing for secure
  messaging.

  This variable is modified by automaton and object
  components */

SHARED INOUT
channelOutRaw: info.m

/* ************************************************** */
/* ICC_Policy */
/* ************************************************** */
IN
securityViolation: bool

/* ************************************************** */
/* Automaton */
/* ************************************************** */

IN
AutomatonState: states

OUT
AutomatonChannelInDecoded: info.m;
AutomatonSecurityViolation: bool

/* ********************************************************** */
/* SecChannelIn */
/* ********************************************************** */
IN
SecChannelInChannelInDecoded: info.m

OUT
SecChannelInSecureMsgKeyIn,
SecChannelInSecureMsgKeySigIn: info.m

/* ********************************************************** */
/* SecChannelOut */
/* ********************************************************** */
OUT
SecChannelOutChannelOutRaw,
SecChannelOutSecureMsgKeyOut,
SecChannelOutSecureMsgKeySigOut: info.m;
SecChannelOutSecurityViolation: bool

/* ********************************************************** */
/* Objects */
/* ********************************************************** */
IN
O8SecureMsgKeyIn, O8SecureMsgKeySigOut,
O9SecureMsgKeyOut, O9SecureMsgKeySigIn: info.m

OUT
O2state,
O3state,
O4state,
O5state,
O6state,
O7state,
O8state,
O9state,
O10state,
O11state,
O12state: states;
O2Command,
O3Command,
O4Command,
O5Command,
O6Command,
O7Command,
O8Command,
O9Command,
O10Command,
O11Command,
O12Command: info.m

SPEC
[] ChannelOutRaw = SecChannelOutChannelOutRaw;
Generic Formal Model of Security Policy for Signature Creation SmartCards

[[] AutomatonState = O2state;
[[] AutomatonState = O3state;
[[] AutomatonState = O4state;
[[] AutomatonState = O5state;
[[] AutomatonState = O6state;
[[] AutomatonState = O7state;
[[] AutomatonState = O8state;
[[] AutomatonState = O9state;
[[] AutomatonState = O10state;
[[] AutomatonState = O11state;
[[] AutomatonState = O12state;
[[] AutomatonSecurityViolation = securityViolation;

[[] SecChannelInChannelInDecoded = AutomatonChannelInDecoded;
[[] SecChannelInChannelInDecoded = O2Command;
[[] SecChannelInChannelInDecoded = O3Command;
[[] SecChannelInChannelInDecoded = O4Command;
[[] SecChannelInChannelInDecoded = O5Command;
[[] SecChannelInChannelInDecoded = O6Command;
[[] SecChannelInChannelInDecoded = O7Command;
[[] SecChannelInChannelInDecoded = O8Command;
[[] SecChannelInChannelInDecoded = O9Command;
[[] SecChannelInChannelInDecoded = O10Command;
[[] SecChannelInChannelInDecoded = O11Command;
[[] SecChannelInChannelInDecoded = O12Command;
[[] SecChannelOutSecurityViolation = securityViolation;

[[] O8SecureMsgKeyIn = SecChannelInSecureMsgKeyIn;
[[] O8SecureMsgKeySigOut = SecChannelOutSecureMsgKeySigOut;
[[] O9SecureMsgKeyOut = SecChannelOutSecureMsgKeyOut;
[[] O9SecureMsgKeySigIn = SecChannelInSecureMsgKeySigIn

#POS -430,440
TLSPEC ICC_Assumption
PURPOSE
" Assumptions on the input to the TOE. "
USING TICC
DATA
OUT
channelIn: info.m

VARS i,j: information;
k: kindSecret

SPEC
/* Assumptions about the input of the card. */
1.
We have to assume that the input of the card does not contain any secret informations (p.e. the key for the cardholder’s signature) in order to be able to show that no such information can be extracted from the card. If for example the card got the command to sign a document which contained the key, the answer would be a message from which the key could easily be inferred.
There are two exceptions:
- the cardholders PIN for authentication
2. If new PINs or display messages are sent, they must obey to the conditions for these kinds of secrets. (In real life this means, the cardholder has to choose good PINs and display messages, that cannot be guessed).

```plaintext
/*
[] isSecretInferable(k,channelIn) -> k = kindPin OR k = kindDisplayMessage;
[] isSecretInferable(kindPIN,channelIn) ->
    EX i,j: (channelIn = ifdDoAuth(i) OR channelIn = ifdDoAuthChange(i,j));
[] isSecretInferable(kindDisplayMessage,channelIn) ->
    EX i: channelIn = ifdDoModifyObj7(i);
/* We also have to assume that PINs or display messages are good in the sense that they are secret. */
[] secureMsgDecode(channelIn,k1,k2) = ifdDoAuthChange(i,j) -> isSecret(kindPIN,def(j));
[] secureMsgDecode(channelIn,k1,k2) = ifdDoModifyObj11(i) -> isSecret(kindDisplayMessage,def(i))
```

**TLSPECEND**

**TLSPEC ICC_Environment**

**PURPOSE** "Combination of ICC_Policy with ICC_Assumption in order to get an assumption about the input of the TOE."

**USING** TICC; boolean

/* Interface: The same as ICC_Policy except for channelIn. */

**DATA**

**OUT**
- channelOut: info.m;
- channelOutRawH: list

**IN**
- securityViolation: bool;
- power, reset: bool

**COMBINE** ICC_Policy

- ICC_Policy.channelOut -> ICC_Environment.channelOut,
- ICC_Policy.channelOutRawH -> ICC_Environment.channelOutRawH,
- ICC_Policy.securityViolation <- ICC_Environment.securityViolation,
- ICC_Policy.channelIn <- ICC_Assumption.channelIn,
- ICC_Policy.power <- ICC_Environment.power,
- ICC_Policy.reset <- ICC_Environment.reset

**TLSPRECEND**

**TLSPEC ICC_Objectives**

**PURPOSE**

"Specification of the ICC derived from the security objectives."
It differs from ICC in that it is expressed in terms of the states from ICC_Policy

```
USING TAutomaton

DATA
/* OUTPUT */
/* OUTPUT */
/* OUTPUT */

OUT
/* Informations leaving the ICC: Every time the ICC sends a message i, the value of channelOut becomes i. */
channelOut : info.m;

/* The history variable for the internal output variable */
channelOutRawH: list

/* INPUT */
/* INPUT */
/* INPUT */

IN
/* Informations arriving as inputs to the ICC (from an IFD if the ICC is used as supposed to). Any input i of the ICC becomes the value of channelIn. */
channelIn: info.m;

/* Additional Security Violations: Security violations that are implementation dependent are reflected by the boolean variable securityViolation. These violations can include those detected by the operating system of the card or the hardware. securityViolation becomes t if a security violation has been detected. This variable seems to be internal in the sense that it occurs 'inside the real ICC'. But with respect to the TOE it is an input variable, because it represents a value coming from the outside rather than being computed by the TOE. */
securityViolation: bool;

/* The state of power supply. It is t, iff the card is being supplied with power. */
power: bool;

/* Reset signal. It is t, iff the terminal raises a reset. */
reset: bool

/* internal Data of the ICC */
/* internal Data of the ICC */
/* internal Data of the ICC */

INTERNAL
/* Preprocessed Input of ICC: If secure messaging is in effect, channelInDecoded is the decoded content of channelIn or invalid, if channelIn

82 / 153 12. 9. 2000
was not properly encoded/signed. Otherwise channelInDecoded will be assigned the value of channelIn. */
channelInDecoded: info.m;

/* Combined authentication status */
state: states

/* """ is not used and can thus be deleted from specification without semantically modifying it as soon as the bug has been eliminated. */

ACTIONS
/* Dummy action in order to create a mappable parameter variable. */
dummy(sOut: OUT subject) ::= true

SPEC
/* """ */
/* INITIAL */
/* """ */
/* Values that have to be initialised during 
   personalization */
skCh = skeygen(initialSKeyCounter);
DisplayMessage = initialDisplayMessage;
pinCh = initialPin;
/* The card starts without any active secure messaging */
secureMsgKeyOut = noKey;
secureMsgKeySigOut = noKey;
secureMsgKeyIn = noKey;
secureMsgKeySigIn = noKey;
/* History variables are empty */
channelOutRawH = nil;

/* ********************************************************** */
/* HISTORY VARIABLE */
/* ********************************************************** */
/* The history variable is a representation of all outputs 
generated by the ICC up to the current point of time.
Its purpose is to express the condition about the maximal 
number of user authentication failures. This variable is 
needed to express SO2.4 */

/* Entries in the history variable never get lost. */
[] channelOutRawH' = channelOutRawH OR
channelOutRawH' = cons(first(channelOutRawH'),channelOutRawH);

/* ********************************************************** */
/* SECURITY OBJECTIVES */
/* ********************************************************** */
/* The first paragraph of the comment for each SO states the 
informal SO verbatim */

/* ----- */
/* SO1 */
/* ----- */
/* The TOE ensures the confidentiality and the integrity of the 
SigG private signature key of the cardholder stored in the TOE 
with two aspects: */

/* ----- */
/* SO1.1 */
/* ----- */
/* The TOE shall prevent any kind of extraction of the cardholder’s 
secret key from the ICC. */

/* Note: 
   This condition is stronger than the one for SO8. */
[] NOT mInferable(channelOut,def(skCh));

/* ----- */
/* SO1.2 */
/* ----- */
/* The TOE shall prevent any kind of modification of the 
cardholder’s secret key */
[] skCh = skCh';
/* ----- */
/* S02 */
/* ----- */
/* The TOE shall allow the use of the digital signature function
only to the cardholder. This security objective has the following
aspects: */

/* ----- */
/* S02.1 */
/* ----- */
/* The TOE shall allow the use of the digital signature function
only to the cardholder after successful authentication by
knowledge. */

/* Authentication:
The user may only be considered as the cardholder after the
IFD has sent his PIN and the authentication has not expired
yet because of closing the application or resetting the card.
This can be checked by inspecting the history of outputs. */

[] ALL i:
    mInferableWithout(channelOut,mSig(i,def(skCh)),def(skCh)) ->
    histAuthUser(channelOutRawH);

/* ----- */
/* S02.2 */
/* ----- */
/* Successive authentication failures will be interpreted as an attempted
unauthorised access by the TOE and will disable the signature function. */

/* The history variable reveals the amount of successive failures.
The predicate maxFailuresExceeded becomes true if this amount
exceeds the maximum. */

[] first(channelOutRawH) = value(answerAuthSuccess) ->
    NOT maxFailuresExceeded(rest(channelOutRawH));

/* ----- */
/* S02.3 */
/* ----- */
/* The authentication data is stored in the TOE and shall not be dis-
closed. */

[] NOT mInferable(channelOut,def(pinCh));
[] NOT mInferable(channelOut,def(PUK));

/* S03 -- S05 are for public IFDs only */
/* ----- */
/* S03 */
/* ----- */
/* ----- */
/* S03.1 */
/* ----- */
/* The TOE shall be able to verify the SigG accreditation of
the IFD. */
/* This security objective doesn’t impose any direct requirement
to the TOE except that the functionality to verify SigG
accreditation of the actual IFD should be implemented.
On the abstraction level of this part of specification this
objective could only be expressed by stating that a certain
output is present on the output channel sometimes or - if a time out
occurs - not at all, which is a tautology.
On the other hand is the responsibility of the terminal to check the
presence and validity of a certificate produced by the TOE.
Nevertheless it can easily be seen that device authentication is
provided by object component O8 of the functional level. */

/* ----- */
/* SO3.2 */
/* ----- */
/* If the TOE verified successfully the SigG accreditation of a
IFD then the TOE grants access to the display message stored in
the TOE. */

[] mInferable(channelOut,def(DisplayMessage))
  -> histAuthIfd(channelOutRawH) OR
    histAuthUser(channelOutRawH);

/* --- */
/* SO4 */
/* --- */
/* The TOE shall be able to demonstrate its SigG accreditation
to the cardholder using a public IFD. */
/* This security objective doesn’t impose any direct requirement
to the TOE except that the functionality to verify SigG
accreditation of the actual IFD should be implemented.
On the abstraction level of this part of specification this
objective could only be expressed by stating that a certain
output is present on the output channel sometimes or - if a time out
occurs - not at all, which is a tautology.
On the other hand is the responsibility of the user to check the
presence and validity of the display message produced by the TOE.
Nevertheless it can easily be seen that device authentication is
provided by object component O11 of the functional level. */

/* ----- */
/* SO5 */
/* ----- */
/* ----- */
/* SO5.1 */
/* ----- */
/* The TOE shall enforce secure messaging between the ICC and the
successfully authenticated SigG accredited public IFD preventing
the disclosure of sensitive data. The sensitive data to be
protected are at least (i) the verification data of the
cardholder, (ii) the reference data of the cardholder transmitted
from the IFD to the ICC and (iii) the display message transmitted from the ICC to the IFD. */

/* Sensitive informations may leave the card only on secure channels if an authenticated terminal is used. Then they must not be sent in clear. Otherwise the user must have been authenticated. This is evidence that the user approved the terminal as a private one. */

[] ALL sk, i:
    NOT channelOut' = channelOut AND
    histAuthIfd(channelOutRawH) AND
    validSecureMsgPair(sk, secureMsgKeyOut) AND
    isIccSensitive(i) ->
    NOT mInferableWithout(channelOut’, i, sk);

/* ----- */
/* SO5.2 */
/* ----- */

/* The TOE shall enforce secure messaging between the ICC and the successfully authenticated SigG accredited public IFD for all commands of the IFD and responses of the ICC preventing any unnoticed modification of the data in transfer. */

[] NOT channelOut’ = noInfo AND
    NOT channelOut’ = channelOut AND
    NOT secureMsgKeyOut = noKey AND
    NOT secureMsgKeySigOut = noKey
    -> EX i: channelOut’ =
        secureMsgEncode(i, secureMsgKeyOut, secureMsgKeySigOut);

/* --- */
/* SO6 */
/* --- */

/* Any key material used by the TOE shall bear a strong cryptographic quality. The cryptographic quality is characterised as follows: */
/* The FMSP describes the behaviour of the ICC in operational use only and does not deal with the process of personalisation. This is in accordance with the underlying security policy described in the GST [Sec. 4] since key generation is not considered in the specification of the state machine [Sec. 4.1]. Moreover the access type generation to O2 is completely denied by the TOE [Sec. 4.2]. Therefore, the only requirement to be met during operational use is that skCh must never change. This already is stated and formalised in SO1.2 */

/* ----- */
/* SO6.1 */
/* ----- */
/* If secret keys are generated either in the personalisation phase or in operational use by the TOE this process must be performed in a confidential way. */

/* ----- */
/* SO6.2 */
/* ----- */
/* The secret keys generated must be unique with a very high probability and cryptographically strong. */

/* ----- */
/* S06.3 */
/* ----- */
/* It shall be impossible to calculate the secret key from the public key. */

/* ----- */
/* S07.1 */
/* ----- */

/* The TOE provides a function to generate a SigG digital signature using the SigG private signature key of the cardholder stored in the TOE. */

[] validpair(isk,ipk) AND 
   NOT channelOut' = channelOut AND 
   minferableWithout(channelOut',def(sig(j,isk)),def(isk)) 
   -> channel1nDecoded = ifdDoSign(j) AND 
      skCh = isk

/* ----- */
/* S07.2 */
/* ----- */
/* The function to generate a SigG digital signature works in a manner that other individuals not possessing the SigG private signature key of the cardholder cannot generate the signature. */

/* ----- */
/* S08 */
/* ----- */

/* S08.1 */
/* ----- */

/* The TOE shall react to potential security violations which are (i) recognised and signalled to the TOE by the underlying hardware or (ii) identified by the TOE itself. The detection of a security violation does not have a direct impact on output variables. Therefore S08.1 does not correspond to any formally specified requirements in ICC_Objectives. */

/* ----- */
/* S08.2 */
/* ----- */
/* If a potential security violation is detected then 
(1) the TOE reaches a secure blocking state disabling all application of the ICC and 
(2) the blocking state is made apparent to the user. */
/* Due to the generic nature of the security target this security objective is incompletely specified. The sponsor is demanded in [2.6.8] to precisely state how the secure blocking state of the TOE is characterised and how this state is made..."
apparent to the user. Therefore, S08.2 doesn’t correspond to any
formally specified requirements in ICC_Objectives. */

HIDE channelInDecoded,
state,
skCh,
DisplayMessage,
pinCh,
PUK,
secureMsgKeyOut,
secureMsgKeyIn,
secureMsgKeySigOut,
secureMsgKeySigIn

#POS -121,312
TLSPEC ICC_Policy
PURPOSE
" Specification of the ICC derived from the security objectives "
USING TTICC_Policy
DATA
/* ---------------------------------------------------------- */
/* OUTPUT */
/* ---------------------------------------------------------- */
OUT
/* Informations leaving the ICC:
Every time the ICC sends a message i, the value of channelOut
becomes i. */
channelOut : info.m;

/* The history variable for the internal output variable */
channelOutRawH: list

/* ---------------------------------------------------------- */
/* INPUT */
/* ---------------------------------------------------------- */
IN
/* Informations arriving as inputs to the ICC
(from an IFD if the ICC is used as supposed to).
Any input i of the ICC becomes the value of channelIn. */
channelIn : info.m

/* Additional Security Violations:
Security violations that are implementation dependent are
reflected by the boolean variable securityViolation.
These violations can include those detected by the operating
system of the card or the hardware.
securityViolation becomes t if a security violation has
been detected. */
securityViolation : bool;
/* The state of power supply. */
power : bool;
/* Reset signal. */
It is t, iff the terminal raises a reset. /*
reset : bool

VARS i, j : info.m;
k : kindSecret

COMBINE
DataLink
[AutomatonState <- Automaton.state,
DataLink.securityViolation <- ICC_Policy.securityViolation,
SecChannelIn.channelInDecoded <- SecChannelIn.channelInDecoded,
08SecureMsgKeyIn <- 08.secureMsgKeyIn,
08SecureMsgKeySigOut <- 08.secureMsgKeySigOut,
09SecureMsgKeyOut <- 09.secureMsgKeyOut,
09SecureMsgKeySigIn <- 09.secureMsgKeySigIn]

DATA
[Datamass channelOutRaw <- automaton.channelOutRaw];

Automaton
[Automaton.securityViolation <- DataLink.AutomatonSecurityViolation,
Automaton.power <- ICC_Policy.power,
Automaton.reset <- ICC_Policy.reset,
Automaton.channelInDecoded <- DataLink.AutomatonChannelInDecoded,
Automaton.AuthChFailures <- 03.AuthChFailures,
Automaton.channelOutRawH <- ICC_Policy.channelOutRawH]

SHARE
[Automaton.channelOutRaw <- automaton.channelOutRaw,
Automaton.AuthChResult <- 03.event,
Automaton.AuthPUKResult <- 04.event,
Automaton.authIfdResult <- 09.event];

SecChannelOut
[SecChannelOut.securityViolation <- DataLink.SecChannelOutSecurityViolation,
SecChannelOut.channelOut -> ICC_Policy.channelOut,
SecChannelOut.channelOutRaw <- DataLink.SecChannelOutChannelOutRaw,
SecChannelOut.secureMsgKeyOut <- DataLink.SecChannelOutSecureMsgKeyOut,
SecChannelOut.secureMsgKeySigOut <- DataLink.SecChannelOutSecureMsgKeySigOut]

SHARE
[SecChannelOut.buck <- automaton.buck];

SecChannelIn
[SecChannelIn.channelIn <- ICC_Policy.channelIn,
SecChannelIn.secureMsgKeySigIn <- DataLink.SecChannelInSecureMsgKeySigIn,
SecChannelIn.secureMsgKeyIn <- DataLink.SecChannelInSecureMsgKeyIn]

SHARE
[SecChannelIn.buck <- automaton.buck];

02
[02.command <- DataLink.02Command, 02.state <- DataLink.02state]
SHARE [02.buck <- automaton.buck, 02.valueOut <- automaton.channelOutRaw];

03
[03.command <- DataLink.03Command, 03.state <- DataLink.03state]
SHARE [03.buck <- automaton.buck, 03.valueOut <- automaton.channelOutRaw];

04
[04.command <- DataLink.04Command, 04.state <- DataLink.04state]
SHARE [04.buck <- automaton.buck, 04.valueOut <- automaton.channelOutRaw];

05
[05.command <- DataLink.05Command, 05.state <- DataLink.05state]
SHARE [05.buck <- automaton.buck, 05.valueOut <- automaton.channelOutRaw];

06
[06.command <- DataLink.06Command, 06.state <- DataLink.06state]
SHARE [06.buck <- automaton.buck, 06.valueOut <- automaton.channelOutRaw];

07
[07.command <- DataLink.07Command, 07.state <- DataLink.07state]
SHARED [O7.buck <- automaton.buck, O7.valueOut <- automaton.channelOutRaw];
O8
[O8.command <- DataLink.O8Command, O8.state <- DataLink.O8state]
SHARED [O8.buck <- automaton.buck, O8.valueOut <- automaton.channelOutRaw];
O9
O10
[O10.command <- DataLink.O10Command, O10.state <- DataLink.O10state]
SHARED [O10.buck <- automaton.buck, O10.valueOut <- automaton.channelOutRaw];
O11
[O11.command <- DataLink.O11Command, O11.state <- DataLink.O11state]
SHARED [O11.buck <- automaton.buck, O11.valueOut <- automaton.channelOutRaw];
O12
[O12.command <- DataLink.O12Command, O12.state <- DataLink.O12state]

/* The main part of the overall behaviour is specified as the combination
 of the components.
The only purpose of the following behaviour description is to specify
that the internal variables buck and channelOutRaw do not change while the
complete system is inactive (stuttering) and that always one of the
canonical components is active (i.e. DataLink which is the
only noncanonic component never is active.) Otherwise buck could
change in an undefined manner while DataLink would be active.
DataLink does not need to become active because the formulas
describing its behaviour (SPEC slot) do not depend on the
is_active(DataLink) condition. */

SPEC
[[] is_active(icc_policy) OR automaton.buck' = automaton.buck;
[[] is_active(icc_policy) OR automaton.channelOutRaw' = automaton.channelOutRaw;
[[] is_active(automaton) OR is_active(o3) OR o3.event = o3.event';
[[] is_active(automaton) OR is_active(o4) OR o4.event = o4.event';
[[] is_active(automaton) OR is_active(o9) OR o9.event = o9.event';
[[] NOT is_active(DataLink) OR (automaton.buck' = automaton.buck AND
automaton.channelOutRaw' = automaton.channelOutRaw);
[[] NOT is_active(secchannelin) OR automaton.channelOutRaw' = automaton.channelOutRaw;
[[] NOT is_active(secchannelout) OR automaton.channelOutRaw' = automaton.channelOutRaw

SATISFIES ICC_Objectives
TLSPECEND

#POS 82,-33
TLSpec O10
PURPOSE
* Object O10 as described in the security target.
Device authentication credentials of the TOE. *

USING TObject

DATA
/* ---------------------------------------------------------- */
/* INPUT */
/* ---------------------------------------------------------- */
IN
/* The command input to the TOE after preprocessing by
secure messaging */
command: info.m;

/* The current authentication state. Conditions that enable the actions of this component depend on this variable. */
state: states

/* ---------------------------------------------------------- */
/* SHARED */
/* ---------------------------------------------------------- */
SHARED INOUT
/* The output of the object component */
valueOut: info.m;

/* The variable which is used to establish the control flow */
buck: bucks

/* ---------------------------------------------------------- */
/* INTERNAL */
/* ---------------------------------------------------------- */
INTERNAL
/* The value of the object, which is protected against unprivileged access. */
value: info.m

VARS newvalue: information;
i: information

ACTIONS
/* ********************************************************** */
/* BUCK */
/* ********************************************************** */

/* ---------------------------------------------------------- */
/* handleBuck */
/* ---------------------------------------------------------- */
/* Action for coordination with other components. It is supposed to be conjunctively added to any other action of this component. Thus each action will be enabled only if the control variable has the appropriate value. In turn this value will be set accordingly to enable the next component. (In this case this is the Automaton component). */

handleBuck ::= buck = bckObjects AND buck' = bckAutomaton

/* ********************************************************** */
/* Access Operations */
/* ********************************************************** */
/* ---------------------------------------------------------- */
/* read */
/* Read the protected internal value of the object provided this operation is permitted in the current authentication state. Otherwise give the response 'answerDenied'. */

read ::= handleBuck AND
command = ifdDoReadobj10 AND
IF allowed(obj10,state,ARRead) THEN
  valueOut' = value
ELSE
  valueOut' = answerDenied
FI AND
UNCHANGED(value)

/* modifyInternal */
/* Modify the protected internal value of this object. This operation is intended of internal use within this TLSPEC only. */
modifyInternal(newval: IN information)
::= value' = def(newval)

/* modify */
/* Modify the protected internal value of this object provided this operation is permitted in the current authentication state. Otherwise give the response 'answerDenied'. */
modify ::= handleBuck AND
(EX newvalue: command = ifdDoModifyobj10(newvalue) AND
IF allowed(obj10,state,ARModify) THEN
  modifyInternal(newvalue) AND
  valueOut' = answerModified
ELSE
  UNCHANGED(value) AND
  valueOut' = answerDenied
FI)

SPEC
INITIAL
  value = def(initialIccAuthCred)
TRANSITIONS
  [ read, modify ]
  { value, valueOut, buck } 
FAIRNESS
  WF(read){value, valueOut, buck},
  WF(modify){value, valueOut, buck}
HIDE value
TLSPECEND

#POS 147,-33
TLSpec O11

PURPOSE

"Objetc O11 as described in the security functions. Card display message."

USING TObject

DATA

/* ---------------------------------------------------------- */
/* INPUT */
/* ---------------------------------------------------------- */
IN
/* The command input to the TOE after preprocessing by secure messaging */
command: info.m;

/* The current authentication state. Conditions that enable the actions of this component depend on this variable. */
state: states

/* ---------------------------------------------------------- */
/* SHARED */
/* ---------------------------------------------------------- */
SHARED INOUT
/* The output of the object component */
valueOut: info.m;

/* The variable which is used to establish the control flow */
buck: bucks

/* ---------------------------------------------------------- */
/* INTERNAL */
/* ---------------------------------------------------------- */
INTERNAL
/* The value of the object, which is protected against unprivileged access. */
value: info.m

VARS newvalue: information;
i: information

ACTIONS

/* ********************************************************** */
/* BUCK */
/* ********************************************************** */

/* handleBuck */
/* Action for coordination with other components. It is supposed to be conjunctively added to any other components */
action of this component. Thus each action will be enabled only if the control variable has the appropriate value. In turn this value will be set accordingly to enable the next component. (In this case this is the Automaton component). */

handleBuck ::=  
  buck = bckObjects AND  
  buck' = bckAutomaton

/* ********************************************************** */
/* Access Operations */
/* ********************************************************** */
/* ---------------------------------------------------------- */
/* read */
/* ---------------------------------------------------------- */
/* Read the protected internal value of the object provided this operation is permitted in the current authentication state. Otherwise give the response 'answerdenied'. */
read ::= handleBuck AND  
  command = ifdDoReadobj11 AND  
  IF allowed(obj11,state,ARRead) THEN  
    valueOut' = value  
  ELSE  
    valueOut' = answerDenied  
  FI AND  
  UNCHANGED(value)

/* ---------------------------------------------------------- */
/* modifyInternal */
/* ---------------------------------------------------------- */
/* Modify the protected internal value of this object. This operation is intended of internal use within this TLSPEC only. */
modifyInternal(newval: IN information) ::= 'value' = def(newval)

/* ---------------------------------------------------------- */
/* modify */
/* ---------------------------------------------------------- */
/* Modify the protected internal value of this object provided this operation is permitted in the current authentication state. Otherwise give the response 'answerdenied'. */
modify ::=  
  handleBuck AND  
  (EX newvalue: command = ifdDoModifyobj11(newvalue) AND  
    IF allowed(obj11,state,ARModify) THEN  
      modifyInternal(newvalue) AND  
      valueOut' = answerModified  
    ELSE  
      UNCHANGED(value) AND  
      valueOut' = answerDenied  
    FI)
SPEC
INITIAL
  value = def(initialDisplayMessage) AND
  valueOut = noInfo
TRANSITIONS
  [ read, modify ]
  { value, valueOut, buck }
FAIRNESS
  WF(read){value, valueOut, buck},
  WF(modify){value, valueOut, buck}
HIDE value
TLSPECEND

#POS 212,-33
TLSpec O12
PURPOSE
"Object O12 as described in the security target.
SigG public key of the cardholder."

USING TObject

DATA
/* ---------------------------------------------------------- */
/* INPUT */
/* ---------------------------------------------------------- */
IN
/* The command input to the TOE after preprocessing by */
  secure messaging */
command: info.m;

/* The current authentication state. */
  Conditions that enable the actions of this component */
  depend on this variable. */
state: states

/* ---------------------------------------------------------- */
/* SHARED */
/* ---------------------------------------------------------- */
SHARED INOUT
/* The output of the object component */
valueOut: info.m;

/* The variable which is used to establish the control */
  flow */
buck: bucks

/* ---------------------------------------------------------- */
/* INTERNAL */
/* ---------------------------------------------------------- */
INTERNAL
/* The value of the object, which is protected against */
  unprivileged access. */
value: info.m
VARS newvalue: information;

i,j: information

ACTIONS
/* ********************************************************** */
/* BUCK */
/* ********************************************************** */
/* ---------------------------------------------------------- */
/* handleBuck */
/* ---------------------------------------------------------- */
/* Action for coordination with other components. 
   It is supposed to be conjunctively added to any other 
   action of this component. 
   Thus each action will be enabled only if the control variable 
   has the appropriate value. In turn this value will be 
   set accordingly to enable the next component. 
   (In this case this is the Automaton component). */

handleBuck ::= 
buck = bckObjects AND 
buck' = bckAutomaton
/* ********************************************************** */
/* Access Operations */
/* ********************************************************** */
/* ---------------------------------------------------------- */
/* read */
/* ---------------------------------------------------------- */
/* Read the protected internal value of the object provided 
   provided this operation is permitted in the current 
   authentication state. Otherwise give the response 
   'answerdenied'. */

read ::= handleBuck AND 
command = ifdDoReadobj12 AND 
IF allowed(obj12,state,ARRead) THEN 
valueOut' = value 
ELSE 
valueOut' = answerDenied 
FI AND 
UNCHANGED(value)
/* ---------------------------------------------------------- */
/* modifyInternal */
/* ---------------------------------------------------------- */
/* Modify the protected internal value of this object. 
   This operation is intended of internal use within this 
   TLSPEC only. */
modifyInternal(newval: IN information) ::= value' = def(newval)
/* ---------------------------------------------------------- */
/* modify */
/* ---------------------------------------------------------- */
/* Modify the protected internal value of this object provided
this operation is permitted in the current
authentication state. Otherwise give the response
‘answerdenied’. */
modify ::= handleBuck AND
(EX newValue: command = ifdDoModifyobj12(newValue) AND
IF allowed(obj12,state,ARModify) THEN
modifyInternal(newValue) AND
valueOut’ = answerModified
ELSE
UNCHANGED(value) AND
valueOut’ = answerDenied
FI)
/* ---------------------------------------------------------- */
/* use */
/* ---------------------------------------------------------- */
/* Use the value of the object for signature verification.
This is action is left underspecified because it is not
relevant for the security objectives. */
use ::= handleBuck AND
(EX i,j: (command = ifdDoVerify(i,j) AND
IF allowed(obj12,state,ARUse) THEN
valueOut’ = answerSuccess OR
valueOut’ = answerFailure
ELSE
valueOut’ = answerDenied
FI) AND
UNCHANGED(value)
SPEC
INITIAL
value = info.def(pkeygen(initialSKeyCounter)) AND
valueOut = noInfo
TRANSITIONS
[ use, read, modify ]
{ value, valueOut, buck }
FAIRNESS
WF(use)(value, valueOut, buck),
WF(read)(value, valueOut, buck),
WF(modify)(value, valueOut, buck)
HIDE value
TLSPECEND

#POS -438,-33
TLSpec O2
PURPOSE
" Object O2 as described in the security target.
Sig0 private signature key of the cardholder."
USING TObject

DATA
   /* ----------------------------------------------- */
   /* INPUT */
   /* ----------------------------------------------- */
   IN
   /* The command input to the TOE after preprocessing by secure messaging */
   command: info.m;

   /* The current authentication state. Conditions that enable the actions of this component depend on this variable. */
   state: states

   /* ----------------------------------------------- */
   /* SHARED */
   /* ----------------------------------------------- */
   SHARED INOUT
   /* The output of the object component */
   valueOut: info.m;

   /* The variable which is used to establish the control flow */
   buck: bucks

   /* ----------------------------------------------- */
   /* INTERNAL */
   /* ----------------------------------------------- */
   INTERNAL
   /* The value of the object, which is protected against unprivileged access. */
   value: info.m

VARS newvalue: information;
i: information

ACTIONS
   /* *********************************************** */
   /* BUCK */
   /* *********************************************** */

   /* *********************************************** */
   /* handleBuck */
   /* *********************************************** */

   /* Action for coordination with other components. It is supposed to be conjunctively added to any other action of this component. Thus each action will be enabled only if the control variable has the appropriate value. In turn this value will be set accordingly to enable the next component. (In this case this is the Automaton component). */
handleBuck ::= 
  buck = bckObjects AND 
  buck' = bckAutomaton
/* ********************************************************** */
/* Access Operations */
/* ********************************************************** */
/* ---------------------------------------------------------- */
/* read */
/* ---------------------------------------------------------- */
/* Read the protected internal value of the object provided 
  provided this operation is permitted in the current 
  authentication state. Otherwise give the response 
  'answerdenied'. */
read ::= handleBuck AND 
  command = ifdDoReadobj2 AND 
  IF allowed(obj2,state,ARRead) THEN 
    valueOut' = value 
  ELSE 
    valueOut' = answerDenied 
  FI AND 
  UNCHANGED(value)
/* ---------------------------------------------------------- */
/* modifyInternal */
/* ---------------------------------------------------------- */
/* Modify the protected internal value of this object. 
  This operation is intended of internal use within this 
  TLSPEC only. */
modifyInternal(newval: IN information) 
  ::= value' = def(newval)
/* ---------------------------------------------------------- */
/* modify */
/* ---------------------------------------------------------- */
/* Modify the protected internal value of this object provided 
  this operation is permitted in the current 
  authentication state. Otherwise give the response 
  'answerdenied'. */
modify ::= 
  handleBuck AND 
  (EX newvalue: command = ifdDoModifyobj2{newvalue} AND 
    IF allowed(obj2,state,ARRmodify) THEN 
      modifyInternal(newvalue) AND 
      valueOut' = answerModified 
    ELSE 
      valueOut' = answerDenied 
    FI)
/* ---------------------------------------------------------- */
/* USE */
/* ---------------------------------------------------------- */
/* ---------------------------------------------------------- */
/* Use the object for signature generation provided this 
operation is permitted in the current 
authentication state. Otherwise give the response 
‘answerDenied’. */
use ::= handleBuck AND
EX i: (command = ifdDoSign(i) AND
    IF allowed(obj2,state,ARUse) THEN
      valueOut' = mSig(def(i),value)
    ELSE
      valueOut' = answerDenied
    FI) AND
UNCHANGED(value)

SPEC
INITIAL
  value = info.def(skeygen(initialSKeyCounter)) AND
  valueOut = noInfo
TRANSITIONS
[ use, read, modify ]
{ value, valueOut, buck }
FAIRNESS
  WF(use){value, valueOut, buck},
  WF(read){value, valueOut, buck},
  WF(modify){value, valueOut, buck}
HIDE value

TLSPECEND

/* Local Variables: */
/* mode: indented-text */
/* End: */

TLSpec O3

PURPOSE
" Object O3 as described in the security target. 
SigG cardholder reference data. 
These data are used to authenticate the cardholder (PIN) "

USING TObject

DATA
/* ---------------------------------------------------------- */
/* INPUT */
/* ---------------------------------------------------------- */
IN
/* The command input to the TOE after preprocessing by 
secure messaging */
command: info.m;

/* The current authentication state. 
Conditions that enable the actions of this component 
depend on this variable. */
state: states

/* ---------------------------------------------------------- */
/* SHARED */
/* ---------------------------------------------------------- */
SHARED INOUT
/* The output of the object component */
valueOut: info.m;

/* The variable which is used to establish the control
flow */
buck: bucks

/* ---------------------------------------------------------- */
/* INTERNAL */
/* ---------------------------------------------------------- */
INTERNAL
/* The value of the object, which is protected against
unprivileged access. */
value: info.m

/* ---------------------------------------------------------- */
/* OUTPUT */
/* ---------------------------------------------------------- */
OUT
/* Authentication may only fail a constant amount of times.
AuthChFailures is a counter for failures. */
AuthChFailures: nat

/* ---------------------------------------------------------- */
/* SHARED */
/* ---------------------------------------------------------- */
SHARED INOUT
/* The result of cardholder authentication as a
security related event. (It is shared to allow
the automaton to reset it after reading.) */
event: event.m

VARS newvalue: information;
i,j: information

ACTIONS
/* ********************************************************** */
/* BUCK */
/* ********************************************************** */
/* handleBuck */
/* ********************************************************** */
/* Action for coordination with other components. */
/* It is supposed to be conjunctively added to any other
action of this component. Thus each action will be enabled only if the control variable
has the appropriate value. In turn this value will be
set accordingly to enable the next component.
(In this case this is the Automaton component). */

handleBuck ::= 
buck = bckObjects AND 
buck' = bckAutomaton

/* ********************************************************** */
/* Access Operations */
/* ********************************************************** */
/* ---------------------------------------------------------- */
/* read */
/* ---------------------------------------------------------- */
/* Read the protected internal value of the object provided */
/* provided this operation is permitted in the current */
/* authentication state. Otherwise give the response */
/* 'answerdenied'. */
read ::= handleBuck AND 
command = ifdDoReadobj3 AND 
IF allowed(obj3,state,ARRead) THEN 
valueOut' = value 
ELSE 
valueOut' = answerDenied 
FI AND 
UNCHANGED(value)

/* ---------------------------------------------------------- */
/* modifyInternal */
/* ---------------------------------------------------------- */
/* Modify the protected internal value of this object. */
/* This operation is intended of internal use within this */
/* TLSPEC only. */
modifyInternal(newval: IN information) ::= value' = def(newval)

/* ---------------------------------------------------------- */
/* modify */
/* ---------------------------------------------------------- */
/* Modify the protected internal value of this object provided */
/* this operation is permitted in the current */
/* authentication state. Otherwise give the response */
/* 'answerdenied'. */
modify ::= handleBuck AND 
(command = ifdDoModifyobj3(newvalue) AND 
IF allowed(obj3,state,ARModify) THEN 
modifyInternal(newvalue) AND 
valueOut' = answerModified 
ELSE 
UNCHANGED(value) AND 
valueOut' = answerDenied 
FI)

12.9.2000 103 / 153
/** ---------------------------------------------------------- */
/** useBlocked */
/** ---------------------------------------------------------- */
/* 'Sub-Action' of USE/USEANDMODIFY:
   Attempt to use the PIN while it is blocked. This causes
   a security related event and an answer to be generated.
   This is a subaction of use / useAndModify. */
useBlocked ::= AuthChFailures = 0 AND
event' = def(sre7) AND
valueOut' = noInfo AND
UNCHANGED(value)

/** ---------------------------------------------------------- */
/** unblock */
/** ---------------------------------------------------------- */
/* Unblock user authentication. Reverse action of block.
   It has to be triggered by the object O4 after
   successful PUK-Verification. O4 sets the value of buck
   accordingly. */
unblock ::= buck = bckG3Unblock AND
buck' = bckAutomaton AND
EX i,j: command = ifdDoPUKAuth(i,j) AND
(IF allowed(obj3,state,ARUnblock) THEN
   AuthChFailures' = initialMaxAuthChFailures AND
   value' = def(j) AND /* Modification possible while modify is not allowed.
   This is not a real problem, because unblock must be
   allowed. */
   valueOut' = noInfo AND
   UNCHANGED(event) /* Event already generated by O4 */
ELSE
   event' = def(sre10) AND /* This should never happen
   (handled as security violation) */
   valueOut' = noInfo AND
   UNCHANGED(value,AuthChFailures)
FI)

/** ---------------------------------------------------------- */
/** useSuccess */
/** ---------------------------------------------------------- */
/* 'Sub-Action' of USE/USEANDMODIFY:
   Handles the case of successful user authentication.
   - send the return code for successful authentication to
     the automaton
   - change the internal state such that the answer for
     the first successful authentication is not generated
     any more (if it is the first time)
   - reset the counter for authentication failures */
useSuccess ::= event' = def(sre5) AND
valueOut' = noInfo AND
AuthChFailures > 0 AND
AuthChFailures' = initialMaxAuthChFailures AND
UNCHANGED(value)
/* ------------------------------ */
/* actionDenied */
/* ------------------------------ */
/* 'Sub-Action' of USE/USEANDMODIFY/BLOCK/UNBLOCK: 
Handles the case when the current state does not allow it: 
- generate an answer to be sent to the terminal */
actionDenied ::=
valueOut' = answerDenied AND 
event' = noEvent AND 
UNCHANGED(value)
/* ------------------------------ */
/* block */
/* ------------------------------ */
/* 'Sub-Action' of USE/USEANDMODIFY: 
Block user authentication. All user authentications will fail until the execution of unblock. 
- send the event code sre7 (repeated auth. failure) to the automaton 
- decrease the failure counter 
This is a subaction of use / useAndModify. */
block ::= 
handleBuck AND 
EX i,j: (command = ifdDoAuth(i) OR 
command = ifdDoAuthChange(i,j) ) 
AND def(i) /= value AND 
IF allowed(obj3,state,ARBlock) THEN 
event' = def(sre7) AND 
valueOut' = noInfo AND 
AuthChFailures' = 0 AND 
UNCHANGED(value) 
ELSE 
actionDenied 
FI
/* ------------------------------ */
/* useFailure */
/* ------------------------------ */
/* 'Sub-Action' of USE/USEANDMODIFY: Handles the case of a parameter not matching the PIN (or other cardholder verification data) stored in VALUE. Authentication then fails. 
- send the event code to the automaton 
- decrease the failure counter (if not yet = 0) */
useFailure ::= 
IF AuthChFailures > 1 THEN 
event' = def(sre6) AND 
AuthChFailures' = AuthChFailures - 1 AND 
valueOut' = noInfo AND 
UNCHANGED(value) 
ELSE 
block 
FI
/* ------------------------------ */
/* readUnchanged */
/* Read the protected internal value and leave everything */
/* except for the output variable unchanged. */
readUnchanged ::= read AND UNCHANGED(event, AuthChFailures)

/* use */
/* USE: use the value of this object for user authentication */
/* (and do not change it). */
use ::= handleBuck AND
EX i: (command = ifdDoAuth(i) AND
   IF allowed(obj3, state, ARUse) THEN
      IF AuthChFailures > 0 THEN
         IF def(i) = value THEN
            /* Use the object for authentication successfully. */
            useSuccess AND UNCHANGED(value)
            ELSE /* def(i) /= value */
               useFailure
         FI
      ELSE /* AuthChFailures = 0 */
         useBlocked
      FI
   ELSE /* NOT allowed(...) */
      actionDenied
   FI)

/* useAndModify */
/* USEANDMODIFY: use the value of this object for user */
/* authentication and change it */
useAndModify ::= handleBuck AND
EX i, newValue: (command = ifdDoAuthChange(i, newValue) AND
   IF allowed(obj3, state, ARUse) AND allowed(obj3, state, ARModify) THEN
      IF AuthChFailures > 0 THEN
         IF def(i) = value THEN
            /* Use the object for authentication successfully and change. */
            useSuccess AND modifyInternal(newValue)
            ELSE /* def(i) /= value */
               useFailure
         FI
      ELSE /* AuthChFailures = 0 */
         useBlocked
      FI
   ELSE /* NOT allowed(...) */
      actionDenied
   FI)

SPEC
INITIAL
   value = def(initialPin) AND
event = noEvent AND
AuthChFailures = initialMaxAuthChFailures
TRANSITIONS
[ use, useAndModify, readUnchanged, unblock ] /* block not
included here because it is a subaction of use/useAndModify */
{ value, event, AuthChFailures,
  valueOut, buck }
FAIRNESS
WF(use){value, event, AuthChFailures,
  valueOut, buck},
WF(useAndModify){value, event, AuthChFailures,
  valueOut, buck},
WF(readUnchanged){value, event, AuthChFailures,
  valueOut, buck}
HIDE value

#POS -308,-33
TLSpec 04
PURPOSE
" Object 04 as described in the security target.
SigG cardholder reference reset code. (PUK)
These data are used to authenticate the cardholder
and reset O3 (after repeated failures)"

USING TObject

DATA
/* ---------------------------------------------------------- */
/* INPUT */
/* ---------------------------------------------------------- */
IN
/* The command input to the TOE after preprocessing by
secure messaging */
command: info.m;

/* The current authentication state.
Conditions that enable the actions of this component
depend on this variable */
state: states

/* ---------------------------------------------------------- */
/* OUTPUT */
/* ---------------------------------------------------------- */
SHARED INOUT
/* The output of the object component */
valueOut: info.m;

/* The variable which is used to establish the control
flow */
buck: bucks

/* ---------------------------------------------------------- */
/* INTERNAL */
/* ---------------------------------------------------------- */
INTERNAL
/* The value of the object, which is protected against
  unprivileged access. */
value: info.m

/* ---------------------------------------------------------- */
/* SHARED */
/* ---------------------------------------------------------- */
SHARED INOUT
/* The result of cardholder authentication as a
  security related event. */
event: event.m

/* ---------------------------------------------------------- */
/* INTERNAL */
/* ---------------------------------------------------------- */
/* Authentication may only fail a constant amount of times.
  chPukFailures is counter for failures. */
INTERNAL
chPukFailures: nat

VARS newvalue: information;
i,j: information

ACTIONS
/* ********************************************************** */
/* BUCK */
/* ********************************************************** */
/* handleBuck */
/* ********************************************************** */
/* Action for coordination with other components.
  It is supposed to be conjunctively added to any other
  action of this component.
  Thus each action will be enabled only if the control variable
  has the appropriate value. In turn this value will be
  set accordingly to enable the next component.
  (In this case this is the Automaton component). */
handleBuck ::= buck = bckObjects AND
              buck’ = bckAutomaton

/* ********************************************************** */
/* Access Operations */
/* ********************************************************** */
/* read */
/* Read the protected internal value of the object provided 
provided this operation is permitted in the current 
authentication state. Otherwise give the response 
'answerdenied'. */

read ::= handleBuck AND
    command = ifdDoReadobj4 AND
    IF allowed(obj4,state,ARRead) THEN
        valueOut' = value
    ELSE
        valueOut' = answerDenied
    FI AND
UNCHANGED(value)

/* ---------------------------------------------------------- */
/* modifyInternal */
/* ---------------------------------------------------------- */
/* Modify the protected internal value of this object. 
This operation is intended of internal use within this 
TLSPEC only. */
modifyInternal(newval: IN information)
  ::= value' = def(newval)

/* ---------------------------------------------------------- */
/* modify */
/* ---------------------------------------------------------- */
/* Modify the protected internal value of this object provided 
this operation is permitted in the current 
authentication state. Otherwise give the response 
'answerdenied'. */
modify ::= handleBuck AND
    (EX newvalue: command = ifdDoModifyobj4(newvalue) AND
    IF allowed(obj4,state,ARModify) THEN
        modifyInternal(newvalue) AND
        valueOut' = answerModified
    ELSE
        UNCHANGED(value) AND
        valueOut' = answerDenied
    FI)

/* ---------------------------------------------------------- */
/* use */
/* ---------------------------------------------------------- */
use ::= EX i,j :
    buck = bckObjects AND
    command = ifdDoPukAuth(i,j) AND
    IF allowed(obj4,state,ARUse) THEN
        IF def(i) = value AND chPukFailures > 0 THEN
            buck' = bckO3Unblock AND
            event' = def(sre11) AND
            valueOut' = noInfo AND
            chPukFailures' = initialMaxAuthPukFailures
        ELSE
            buck' = bckAutomaton AND
            valueOut' = noInfo AND
            chPukFailures' = initialMaxAuthPukFailures
        ENDIF
    ELSE
        buck' = bckAutomaton AND
        valueOut' = noInfo AND
        chPukFailures' = initialMaxAuthPukFailures
    ENDIF
IF chPukFailures > 1 THEN
valueOut' = noInfo AND
chPukFailures' = chPukFailures - 1 AND
event' = def(sre12)
ELSE
valueOut' = noInfo AND
chPukFailures' = 0 AND
event' = def(sre10) /* security violation */
FI
FI
ELSE
valueOut' = answerDenied AND
event' = noEvent AND
UNCHANGED(buck)
FI

*SPEC
INITIAL
value = def(initialPuk) AND
event = noEvent AND
chPukFailures = initialMaxAuthPukFailures AND
valueOut = noInfo

TRANSITIONS
[ readUnchanged, modifyUnchanged, use ]
{ value, event, chPukFailures,
valueOut, buck }

FAIRNESS
WF(use){value, event, chPukFailures, valueOut, buck},
WF(modify){value, event, chPukFailures, valueOut, buck},
WF(readUnchanged){value, event, chPukFailures, valueOut, buck}

HIDE value, chPukFailures

TLSPECEND

#POS -243,-33
TLSpec O5

PURPOSE
"Object O5 as described in the security target.
SigG signature key certificate of the cardholder"

USING TObject
DATA
/* ---------------------------------------------------------- */
/* INPUT */
/* ---------------------------------------------------------- */
IN
/* The command input to the TOE after preprocessing by secure messaging */
command: info.m;

/* The current authentication state. Conditions that enable the actions of this component depend on this variable. */
state: states

/* ---------------------------------------------------------- */
/* SHARED */
/* ---------------------------------------------------------- */
SHARED INOUT
/* The output of the object component */
valueOut: info.m;

/* The variable which is used to establish the control flow */
buck: bucks

/* ---------------------------------------------------------- */
/* INTERNAL */
/* ---------------------------------------------------------- */
INTERNAL
/* The value of the object, which is protected against unprivileged access. */
value: info.m

VARS newvalue: information;

i,j: information

ACTIONS
/* ********************************************************** */
/* BUCK */
/* ********************************************************** */
/* ---------------------------------------------------------- */
/* handleBuck */
/* ---------------------------------------------------------- */
/* Action for coordination with other components. It is supposed to be conjunctively added to any other action of this component. Thus each action will be enabled only if the control variable has the appropriate value. In turn this value will be set accordingly to enable the next component. (In this case this is the Automaton component). */
handleBuck ::= bckObjects AND bckAutomaton

/* ********************************************************** */
/* Access Operations */
/* ********************************************************** */
/* ********************************************************** */
/* read */
/* ********************************************************** */
/* Read the protected internal value of the object provided */
/* provided this operation is permitted in the current */
/* authentication state. Otherwise give the response */
/* 'answerdenied'. */

read ::= handleBuck AND command = ifdDoReadobj5 AND IF allowed(obj5,state,ARRead) THEN valueOut' = value ELSE valueOut' = answerDenied FI AND UNCHANGED(value)

/* ********************************************************** */
/* modifyInternal */
/* ********************************************************** */
/* Modify the protected internal value of this object. */
/* This operation is intended of internal use within this */
/* TLSPEC only. */
modifyInternal(newval: IN information) ::= value' = def(newval)

/* ********************************************************** */
/* modify */
/* ********************************************************** */
/* Modify the protected internal value of this object provided */
/* this operation is permitted in the current */
/* authentication state. Otherwise give the response */
/* 'answerdenied'. */
modify ::= handleBuck AND (EX newvalue: command = ifdDoModifyobj5(newvalue) AND IF allowed(obj5,state,ARModify) THEN modifyInternal(newvalue) AND valueOut' = answerModified ELSE valueOut' = answerDenied FI)

/* ********************************************************** */
/* use */
/* ********************************************************** */
/* Use the value of the object for signature verification. 
 * This action is left underspecified because it is not 
 * relevant for the security objectives. */

use ::= handleBuck AND
(EX i,j: (command = ifdDoVerify(i,j) AND
    IF allowed(obj5,state,ARUse) THEN
        valueOut' = answerSuccess OR
        valueOut' = answerFailure
    ELSE
        valueOut' = answerDenied
    FI))
AND UNCHANGED(value)

SPEC
INITIAL
valueOut = noInfo AND
value = def(initialCHCredentials)
TRANSITIONS
{ use, read, modify }
{ value, valueOut, buck }

FAIRNESS
WF(use){value, valueOut, buck},
WF(read){value, valueOut, buck},
WF(modify){value, valueOut, buck}

HIDE value

TLSPECEND

#POS -178,-33
TLSpec O6
PURPOSE
"Object O6 as described in the security target.
Sig0 public key of the root certification authority."

USING TObject

DATA
/* ---------------------------------------------------------- */
/* INPUT *
/* ---------------------------------------------------------- */
IN
/* The command input to the TOE after preprocessing by 
 * secure messaging */
command: info.m;

/* The current authentication state. 
 * Conditions that enable the actions of this component 
 * depend on this variable. */
state: states

/* ---------------------------------------------------------- */
/* SHARED *
/* ---------------------------------------------------------- */
SHARED INOUT
/* The output of the object component */
valueOut: info.m;

/* The variable which is used to establish the control
flow */
buck: bucks

/* ----------------------------------------------- */
/* INTERNAL */
/* ----------------------------------------------- */

INTERNAL
/* The value of the object, which is protected against
unprivileged access. */
value: info.m

VARS newvalue: information;
i,j: information

ACTIONS
/* ----------------------------------------------- */
/* BUCK */
/* ----------------------------------------------- */

/* ----------------------------------------------- */
/* handleBuck */
/* ----------------------------------------------- */
/* Action for coordination with other components.
   It is supposed to be conjunctively added to any other
   action of this component.
   Thus each action will be enabled only if the control variable
   has the appropriate value. In turn this value will be
   set accordingly to enable the next component.
   (In this case this is the Automaton component). */

handleBuck ::= 
buck = bckObjects AND
buck' = bckAutomaton

/* ----------------------------------------------- */
/* Access Operations */
/* ----------------------------------------------- */

/* ----------------------------------------------- */
/* read */
/* ----------------------------------------------- */
/* Read the protected internal value of the object provided
   this operation is permitted in the current
   authentication state. Otherwise give the response
   ‘answerdenied’. */

read ::= handleBuck AND
   command = ifdDoReadobj6 AND
   IF allowed(obj6,state,ARRead) THEN
      valueOut' = value
   ELSE

valueOut' = answerDenied
FI AND
UNCHANGED(value)

modifyInternal

/* Modify the protected internal value of this object.
This operation is intended of internal use within this
TLSPEC only. */
modifyInternal(newval: IN information)
::= value' = def(newval)

modify

/* Modify the protected internal value of this object provided
this operation is permitted in the current
authentication state. Otherwise give the response
‘answerDenied’. */
modify ::= 
handleBuck AND
(EX newvalue: command = ifdDoModifyobj6(newvalue) AND
IF allowed(obj6,state,ARModify) THEN
modifyInternal(newvalue) AND
valueOut' = answerModified
ELSE
UNCHANGED(value) AND
valueOut' = answerDenied
FI)

use

/* Use the value of the object for signature verification.
This is action is left underspecified because it is not
relevant for the security objectives. */
use ::= 
handleBuck AND
(EX i,j: (command = ifdDoVerify(i,j) AND
IF allowed(obj6,state,ARUse) THEN
valueOut' = answerSuccess OR
valueOut' = answerFailure
ELSE
valueOut' = answerDenied
FI)) AND
UNCHANGED(value)

SPEC
INITIAL
valueOut = noInfo AND
value = def(initialCHCredentialsCA)
TRANSITIONS
[ use, read, modify ]
{ value, valueOut, buck }
DATA

/* INPUT */
/* ---------------------------------------------------------- */
IN
/* The command input to the TOE after preprocessing by
secure messaging */
command: info.m;

/* The current authentication state.
Conditions that enable the actions of this component
depend on this variable. */
state: states

/* ---------------------------------------------------------- */
/* SHARED */
/* ---------------------------------------------------------- */
SHARED INOUT
/* The output of the object component */
valueOut: info.m;

/* The variable which is used to establish the control
flow */
buck: bucks

/* ---------------------------------------------------------- */
/* INTERNAL */
/* ---------------------------------------------------------- */
INTERNAL
/* The value of the object, which is protected against
unprivileged access. */
value: info.m

VARS newvalue: information;

i, j: information

ACTIONS
Generic Formal Model of Security Policy
for Signature Creation SmartCards
Version: 1.1

/* ********************************************************** */
/* BUCK */
/* ********************************************************** */

/* -- ----------------------------------- */
/* handleBuck */
/* -- ----------------------------------- */

/* Action for coordination with other components. */
/* It is supposed to be conjunctively added to any other */
/* action of this component. */
/* Thus each action will be enabled only if the control variable */
/* has the appropriate value. In turn this value will be */
/* set accordingly to enable the next component. */
/* (In this case this is the Automaton component). */

handleBuck ::= 
    buck = bckObjects AND
    buck' = bckAutomaton

/* ********************************************************** */
/* Access Operations */
/* ********************************************************** */

/* -- ----------------------------------- */
/* read */
/* -- ----------------------------------- */

/* Read the protected internal value of the object provided */
/* provided this operation is permitted in the current */
/* authentication state. Otherwise give the response */
/* 'answerdenied'. */

read ::= handleBuck AND 
    command = ifdDoReadobj7 AND 
    IF allowed(obj7,state,ARRead) THEN 
    valueOut' = value 
    ELSE 
    valueOut' = answerDenied 
    FI AND 
    UNCHANGED(value)

/* -- ----------------------------------- */
/* modifyInternal */
/* -- ----------------------------------- */

/* Modify the protected internal value of this object. */
/* This operation is intended of internal use within this */
/* TLSPEC only. */

modifyInternal(newval: IN information)
    ::= value’ = def(newval)

/* -- ----------------------------------- */
/* modify */
/* -- ----------------------------------- */

/* Modify the protected internal value of this object provided */
/* this operation is permitted in the current */
/* authentication state. Otherwise give the response */
/* 'answerdenied'. */

modify ::=
handleBuck AND
(EX newvalue: command = ifdDoModify(obj7,newvalue) AND
  IF allowed(obj7,state,ARModify) THEN
    modifyInternal(newvalue) AND
    valueOut’ = answerModified
  ELSE
    UNCHANGED(value) AND
    valueOut’ = answerDenied
  FI)

USE
handleBuck AND
(EX i,j: (command = ifdDoVerify(i,j) AND
  IF allowed(obj7,state,ARUse) THEN
    valueOut’ = answerSuccess OR
    valueOut’ = answerFailure
  ELSE
    valueOut’ = answerDenied
  FI)) AND
UNCHANGED(value)

SPEC
INITIAL
valueOut = noInfo AND
value = def(initialOtherCred)
TRANSITIONS
{ use, read, modify }
{ value, valueOut, buck }

FAIRNESS
WF(use)(value, valueOut, buck),
WF(read)(value, valueOut, buck),
WF(modify)(value, valueOut, buck)

HIDE value

TLSPECEND

#POS -48,-33
TLSpec 08
PURPOSE
" Object O8 as described in the security target.
  Private device authentication key of the TOE. "

USING TObject

DATA

IN
/* The command input to the TOE after preprocessing by secure messaging */
command: info.m;

/* The current authentication state. Conditions that enable the actions of this component depend on this variable. */
state: states

/* ----------------------------------------------------------
/* SHARED
/* ----------------------------------------------------------

SHARED INOUT
/* The output of the object component */
valueOut: info.m;

/* The variable which is used to establish the control flow */
buck: bucks

/* ----------------------------------------------------------
/* INTERNAL
/* ----------------------------------------------------------

INTERNAL
/* The value of the object, which is protected against unprivileged access. */
value: info.m

OUT
/* The secret keys for secure messaging */
secureMsgKeyIn, secureMsgKeySigOut: info.m

VARS newvalue: information;
i: info.m;
challenge: counter

ACTIONS
/* **********************************************************
/* BUCK
/* **********************************************************

/* ----------------------------------------------------------
/* handleBuck
/* ----------------------------------------------------------

/* ------------------------------ *********
/* handleBuck ::= 
buck = bckObjects AND
buck' = bckAutomaton
/* ********************************************************** */
/* Access Operations */
/* ********************************************************** */

/* ---------------------------------------------------------- */
/* read */
/* ---------------------------------------------------------- */
/* Read the protected internal value of the object provided */
/* provided this operation is permitted in the current */
/* authentication state. Otherwise give the response */
/* 'answerdenied' */
read ::= handleBuck AND
        command = ifdDoReadobj8 AND
        IF allowed(obj8,state,ARRead) THEN
            valueOut' = value
        ELSE
            valueOut' = answerDenied
        FI AND
UNCHANGED(value)
/* ---------------------------------------------------------- */

/* modifyInternal */
/* ---------------------------------------------------------- */
/* Modify the protected internal value of this object. */
/* This operation is intended of internal use within this */
/* TLSPEC only */
modifyInternal(newval: IN information)
        ::= value' = def(newval)
/* ---------------------------------------------------------- */

/* modify */
/* ---------------------------------------------------------- */
/* Modify the protected internal value of this object provided */
/* this operation is permitted in the current */
/* authentication state. Otherwise give the response */
/* 'answerdenied' */
modify ::= handleBuck AND
        (EX newvalue: command = ifdDoModifyobj8(newvalue) AND
        IF allowed(obj8,state,ARModify) THEN
            modifyInternal(newvalue) AND
            valueOut' = answerModified
        ELSE
            UNCHANGED(value) AND
            valueOut' = answerDenied
        FI)
/* ---------------------------------------------------------- */

/* use */
/* ---------------------------------------------------------- */
/* USE: */
/* Generate a certificate and generate key pairs for secure */
/* messaging. Public keys are attached to the certificate,
use ::=
handleBuck AND
EX challenge: command = ifdDoIntAuth(challenge) AND IF allowed(obj8,state,ARUse) THEN
    valueOut' = certIccKey(value(value),challenge) AND
    validSecureMsgPair(secureMsgKeyIn',
        extractSecureMsgKey(valueOut')) AND
    validSecureMsgPairSig(secureMsgKeySigOut',
        extractSecureMsgKeySig(valueOut'))
ELSE
    valueOut' = answerDenied AND
    UNCHANGED(secureMsgKeyIn,secureMsgKeySigOut)
FI AND
UNCHANGED(value)

/* ---------------------------------------------------------- */
/* readUnchanged */
/* ---------------------------------------------------------- */
/* Read the protected internal value and leave the output */
/* variables for secure messaging keys unchanged. */
readUnchanged ::= read AND UNCHANGED(secureMsgKeyIn, secureMsgKeySigOut)

/* ---------------------------------------------------------- */
/* modifyUnchanged */
/* ---------------------------------------------------------- */
/* Modify the protected internal value provided this is */
/* allowed in the current state and leave the output */
/* variables unchanged. */
modifyUnchanged ::= modify AND UNCHANGED(secureMsgKeyIn, secureMsgKeySigOut)

/* ---------------------------------------------------------- */
/* reset */
/* ---------------------------------------------------------- */
/* Reset the object after the keys for secure messaging have */
/* become invalid. This is the case after one of the states */
/* cas4, cas5 or cas8 have been left caused by exiting */
/* the SigG application either explicitely or by another */
/* event like security violation or reset. */
reset ::= ((buck = bckObjectReset AND buck' = bckObjectReset9) OR
    (buck = bckObjectReset8 AND buck' = bckAutomaton) ) AND
    ((state /= cas4 AND state /= cas5 AND state /= cas8) AND
        secureMsgKeyIn' = noKey AND
        secureMsgKeySigOut' = noKey
    ) OR
    (state = cas4 OR state = cas5 OR state = cas8) AND
    UNCHANGED(secureMsgKeyIn,secureMsgKeySigOut)) AND
    UNCHANGED(value,valueOut)

SPEC
INITIAL
value = def(initialIccAuthKey) AND
secureMsgKeyIn = noKey AND
secureMsgKeySigOut = noKey

TRANSITIONS
[ use, reset, readUnchanged, modifyUnchanged ]
{ value, secureMsgKeyIn, secureMsgKeySigOut, valueOut, buck }

FAIRNESS
WF(use){value, secureMsgKeyIn, secureMsgKeySigOut, valueOut, buck},
WF(reset){value, secureMsgKeyIn, secureMsgKeySigOut, valueOut, buck},
WF(readUnchanged){value, secureMsgKeyIn, secureMsgKeySigOut, valueOut, buck},
WF(modifyUnchanged){value, secureMsgKeyIn, secureMsgKeySigOut, valueOut, buck}

HIDE value

#POS 17, -33
TLSpec O9

PURPOSE
" Object O9 as described in the security target.
Public key of the root certification authority for
device authentication."

USING TObject

DATA
/* ---------------------------------------------------------- */
/* INPUT */
/* ---------------------------------------------------------- */
IN
/* The command input to the TOE after preprocessing by
secure messaging */
command: info.m;

/* The current authentication state. 
Conditions that enable the actions of this component 
depend on this variable. */
state: states

/* ---------------------------------------------------------- */
/* SHARED */
/* ---------------------------------------------------------- */
SHARED INOUT
/* The output of the object component */
valueOut: info.m;

/* The variable which is used to establish the control 
flow */
buck: bucks

/* ---------------------------------------------------------- */
/* INTERNAL */
/* ---------------------------------------------------------- */
INTERNAL
/* The value of the object, which is protected against 
unprivileged access */
value: info.m
The secret keys for secure messaging:
secureMsgKeySigIn, secureMsgKeyOut: info.m

The result of verification is a security relevant event.
It is reset by the automaton component each time the event has been handled.

event: event.m

VARS newvalue: information;
i: information

handleBuck ::= 
buck = bckObjects AND 
buck' = bckAutomaton

Access Operations

Access Operations

Read the protected internal value of the object provided provided this operation is permitted in the current authentication state. Otherwise give the response 'answerdenied'.
read ::= handleBuck AND
command = ifdDoReadobj9 AND
IF allowed(obj9,state,ARRead) THEN
  valueOut' = value
ELSE
  valueOut' = answerDenied
FI AND
UNCHANGED(value)

modifyInternal

modify

use

EX i: command = ifdDoExtAuth(i) AND
IF allowed(obj9,state,ARUse) THEN
  IF isCertIfdKey(value,def(i)) THEN
    secureMsgKeyOut' = extractSecureMsgKey(command) AND
    secureMsgKeySigIn' = extractSecureMsgKeySig(command) AND
    event' = def(sre9) AND
    valueOut' = noInfo
  ELSE
    UNCHANGED(secureMsgKeyOut, secureMsgKeySigIn)
  ENDIF
ELSE
  UNCHANGED(value)
  valueOut' = answerDenied
FI

/* ---------------------------------------------------------- */
/* modifyInternal */
/* ---------------------------------------------------------- */
/* Modify the protected internal value of this object. */
/* This operation is intended of internal use within this */
/* TLSPEC only. */
modifyInternal(newval: IN information)
  ::= 'value' = def(newval)

/* ---------------------------------------------------------- */
/* modify */
/* ---------------------------------------------------------- */
/* Modify the protected internal value of this object provided */
/* this operation is permitted in the current */
/* authentication state. Otherwise give the response */
/* 'answerDenied'. */
modify ::= handleBuck AND
(EX newValue: command = ifdDoModifyobj9(newValue) AND
 IF allowed(obj9,state,ARModify) THEN
   modifyInternal(newValue) AND
   valueOut' = answerModified
 ELSE
   UNCHANGED(value) AND
   valueOut' = answerDenied
 FI)

/* ---------------------------------------------------------- */
/* use */
/* ---------------------------------------------------------- */
/* USE: */
/* Verify the certificate and challenge response from the IFD */
/* and extract the session keys on success. The value */
/* representing no key (noKey) is assigned in case of failure. */
/* The event is set according to the success of the */
/* verification. */
use ::= handleBuck AND
EX i: command = ifdDoExtAuth(i) AND
IF allowed(obj9,state,ARUse) THEN
  IF isCertIfdKey(value,def(i)) THEN
    secureMsgKeyOut' = extractSecureMsgKey(command) AND
    secureMsgKeySigIn' = extractSecureMsgKeySig(command) AND
    event' = def(sre9) AND
    valueOut' = noInfo
  ELSE
    UNCHANGED(secureMsgKeyOut, secureMsgKeySigIn)
  ENDIF
ELSE
UNCHANGED(value)
valueOut' = answerDenied AND
  event' = noEvent
FI
ELSE
valueOut' = answerDenied AND
  event' = noEvent AND
  UNCHANGED(secureMsgKeyOut, secureMsgKeySigIn)
FI AND
  UNCHANGED(value)
/* ---------------------------------------------------------- */
/* readUnchanged */
/* ---------------------------------------------------------- */
/* Read the protected internal value and leave the output */
/* variables for secure messaging keys unchanged. */
readUnchanged ::= read AND UNCHANGED(event, secureMsgKeySigIn, secureMsgKeyOut)
/* ---------------------------------------------------------- */
/* modifyUnchanged */
/* ---------------------------------------------------------- */
/* Modify the protected internal value provided this is */
/* allowed in the current state and leave the output */
/* variables unchanged. */
modifyUnchanged ::= modify AND UNCHANGED(event, secureMsgKeySigIn, secureMsgKeyOut)
/* ---------------------------------------------------------- */
/* reset */
/* ---------------------------------------------------------- */
/* Reset the object after the keys for secure messaging have */
/* become invalid. This is the case after one of the states */
/* cas4, cas5 or cas8 have been left caused by exiting */
/* the SigG application either explicitly or by another */
/* event like security violation or reset. */
reset ::= ((buck = bckObjectReset AND buck' = bckObjectReset8) OR
  (buck = bckObjectReset9 AND buck' = bckAutomaton)) AND
  ((
    (state /= cas4 AND state /= cas5 AND state /= cas8) AND
    secureMsgKeyOut' = noKey AND
    secureMsgKeySigIn' = noKey
  ) OR
  (state = cas4 OR state = cas5 OR state = cas8) AND
  UNCHANGED(secureMsgKeyOut, secureMsgKeySigIn)) AND
  UNCHANGED(value, valueOut, event)
SPEC
INITIAL
  value = def(initialIfdAuthCred);
  secureMsgKeySigIn = noKey;
  secureMsgKeyOut = noKey;
  event = noEvent
TRANSITIONS
  [ use, readUnchanged, modifyUnchanged, reset ]
  { value, event, secureMsgKeySigIn, secureMsgKeyOut, valueOut, buck }
FAIRNESS
  WF(use){value, event, secureMsgKeySigIn, secureMsgKeyOut, valueOut, buck},

12.9.2000


**Generic Formal Model of Security Policy for Signature Creation SmartCards**

**WF(readUnchanged){value, event, secureMsgKeySigIn, secureMsgKeyOut, valueOut, buck},**

**WF(modifyUnchanged){value, event, secureMsgKeySigIn, secureMsgKeyOut, valueOut, buck},**

**WF(reset){value, event, secureMsgKeySigIn, secureMsgKeyOut, valueOut, buck}**

HIDE value

TLSPECEND

/* Local Variables: */
/* mode: indented-text */
/* End: */

#POS -315,345

TLSPEC SecChannelIn

**PURPOSE**

* Secure Channel for Input *

**USING** TICCPolicy; natural

**DATA**

/* ------------------------------- */
/* OUTPUT */
/* ------------------------------- */

OUT

/* Input of the ICC after decoding and signature verification according to secure messaging. */

channelInDecoded: info.m

/* ------------------------------- */
/* INPUT */
/* ------------------------------- */

IN

/* Informations arriving as inputs to the ICC. It is encoded and signed according to secure messaging. */

channelIn: info.m

IN

/* The keys for secure messaging. They are computed by device authentication */

secureMsgKeyIn: info.m;
secureMsgKeySigIn: info.m

/* ------------------------------- */
/* SHARED */
/* ------------------------------- */

SHARED INOUT

/* The variable which is used to establish the control flow */

buck: bucks

VARS dat,i,j,k : info.m;

c : counter

**ACTIONS**

/* ------------------------------- */
/* prepareInput */
/* ------------------------------- */
/* Decode the input from channelIn according to secure messaging and forward it along the variable channelInDecoded to the other components. */
prepareInput ::= 
buck = bckSecChannelIn AND 
buck’ = bckAutomaton AND 
/* Precondition: is an input from the terminal */ 
channelIn /= noInfo AND 
/* Postcondition: Forward the decoded input to the other components */
channelInDecoded’ = secureMsgDecode(secureMsgKeyIn,secureMsgKeySigIn, channelIn)

/* ---------------------------------------------------------- */
/* waitForNoInfo */
/* ---------------------------------------------------------- */
/* Wait for the the input channel to be noInfo. In our abstract protocol each real input (command) to the card is preceded by the value noInfo. This way we prevent commands from being handled twice or more often only because they stay in the input variable long enough */
waitForNoInfo ::= 
channelIn = noInfo AND 
UNCHANGED(channelInDecoded, buck)

SPEC
INITIAL
channelInDecoded = noInfo
TRANSITIONS
BEGIN
WHILE true DO
BEGIN
waitForNoInfo;
prepareInput
/* Now this component stutters to allow the other components to work on the input and produce output. */
END OD
END

{ channelInDecoded, buck }

TLSPECEND

#POS -320,260

TLSPEC SecChannelOut

PURPOSE
" Secure Channel Output "
USING TICC_Policy; natural
DATA
/* OUTPUT */
/* OUTPUT */
OUT
/* Output of the ICC prepared according to secure messaging */
channelOut: info.m

/* INPUT */
/* INPUT */
/* INPUT */
IN

12.9.2000
/* The output of the ICC in clear text before any secure messaging transformations have been applied. One variable for each component that produces output is supplied. */
channelOutRaw: info.m

IN

/* The keys for secure messaging. They are computed by device authentication */
secureMsgKeyOut: info.m;
secureMsgKeySigOut: info.m

IN

/* The security violation indicator. If a security violation has occurred outputs must not be generated any more. */
securityViolation: bool

/* ---------------------------------------------------------- */
/* SHARED */
/* ---------------------------------------------------------- */
SHARED INOUT
/* The variable which is used to establish the control flow */
buck: bucks

VARS dat,i,j,k : info.m;
c: counter

ACTIONS

/* ---------------------------------------------------------- */
/* prepareOutput */
/* ---------------------------------------------------------- */
/* Encode the output from the parameter channelOutRaw according to secure messaging and send it to the output channelOut. This is only possible if no information (noInfo) is on the input line. */
prepareOutput ::= buck = bckSecChannelOut AND buck’ = bckAutomaton AND channelOut’ = secureMsgEncode(secureMsgKeyOut,secureMsgKeySigOut, channelOutRaw)

/* ---------------------------------------------------------- */
/* reset */
/* ---------------------------------------------------------- */
/* Set the output back to noInfo according to the protocol which requires each value sent to be followed by the noInformation constant. */
reset ::= channelOut’ = noInfo AND UNCHANGED(buck)
/* ---------------------------------------------------------- */
/* ready */
/* ---------------------------------------------------------- */

/* Mutual Synchronisation with Automaton. */
The Automaton has to wait before every state change
until the output is reset. This action is to be placed after
reset. Then it returns the buck back to Automaton only

after reset. */
ready ::= 
buck = bckSecChannelOutReady AND buck’ = bckAutomaton AND
UNCHANGED(channelOut)

SPEC
INITIAL
channelOut = noInfo
TRANSITIONS
BEGIN
WHILE true DO
BEGIN
prepareOutput;
reset;
ready
END OD
END
{channelOut,buck }

#GRAPH Theories
#POS 857,462
THEORY SSignature
PURPOSE " Formalisation of SO7.2 "

USING TSignature

VARS sk,pk,i: information

AXIOMS
/* ----- */
/* SO7.2 */
/* ----- */

/* The function to generate a SigG digital signature works in a
manner that other individuals not possessing the SigG private
signature key of the cardholder cannot generate the signature. */

/* This SOS can be circumscribed as:
If the secret key can be derived from a signed message, it can already
be derived from the unsigned message. */

validpair(sk,pk) AND inferable(sig(i,sk),sk) -> inferable(i,sk)

THEORYEND

#POS 897,748
THEORY TAccessRights
PURPOSE "Theory of accessrights"
USING TStates; TObjectIds; boolean
TYPES
/* Access rights defined in the security claims for security functions. We don’t add infer and refer because these rights are forbidden and thus mentioning them here is useless. */
rights = FREELY GENERATED BY AROpen | ARClose | ARRead | ARModify | ARUse | ARSupplement | ARBlock | ARUnblock ;

/* Tuple of boolean entries */
boolRec = FREELY GENERATED BY
mkBoolRec(getCas1: bool,
getCas2: bool,
getCas3: bool,
getCas4: bool,
getCas5: bool,
getCas6: bool,
getCas7: bool,
getCas8: bool)

FUNCTIONS
/* Access rights function: It returns a tuple r which contains t for exactly those authentication states in which object \1 may grant access right of kind \2 */
allowedRec : objectIds, rights -> boolRec;
/* A constant for a tuple with all entries = f */
fRec: boolRec

PREDICATES
/* Access rights predicate: It is valid, iff object \1 may grant access right of kind \2 in authentication state \3. */
allowed: objectIds, states, rights

VARS
s: states;
r: rights;
obj: objectIds

AXIOMS
/* ALLOWED is defined in terms of the function allowedRec */
FOR allowed :
DEFPRED allowed(obj,s,r) <->
SWITCH s IN
CASE cas1: getCas1(allowedRec(obj,r)) = t
CASE cas2: getCas2(allowedRec(obj,r)) = t
CASE cas3: getCas3(allowedRec(obj,r)) = t
CASE cas4: getCas4(allowedRec(obj,r)) = t
CASE cas5: getCas5(allowedRec(obj,r)) = t
CASE cas6: getCas6(allowedRec(obj,r)) = t
CASE cas7: getCas7(allowedRec(obj,r)) = t
CASE cas8: getCas8(allowedRec(obj,r)) = t
NI
FOR fRec:
DEFFUNC fRec = mkBoolRec(f,f,f,f,f,f,f)

130 / 153 12. 9. 2000
/* The following axioms are a representation of Table 13 */

```plaintext
allowedRec(obj2, AROpen) = fRec;
allowedRec(obj2, ARClose) = fRec;
allowedRec(obj2, ARRead) = fRec;
allowedRec(obj2, ARModify) = fRec;
allowedRec(obj2, ARUse) = mkBoolRec(f, t, f, t, f, t, f, f);
allowedRec(obj2, ARSupplement) = fRec;
allowedRec(obj2, ARBlock) = fRec;
allowedRec(obj2, ARUnblock) = fRec;

allowedRec(obj3, AROpen) = fRec;
allowedRec(obj3, ARClose) = fRec;
allowedRec(obj3, ARRead) = fRec;
allowedRec(obj3, ARModify) = mkBoolRec(f, t, f, t, f, t, f, f);
allowedRec(obj3, ARUse) = mkBoolRec(f, t, t, t, f, f, t, f);
allowedRec(obj3, ARSupplement) = fRec;
allowedRec(obj3, ARBlock) = mkBoolRec(f, t, t, t, f, t, f, t);
allowedRec(obj3, ARUnblock) = mkBoolRec(f, t, t, f, t, f, t, t);

allowedRec(obj4, AROpen) = fRec;
allowedRec(obj4, ARClose) = fRec;
allowedRec(obj4, ARRead) = fRec;
allowedRec(obj4, ARModify) = mkBoolRec(f, t, t, f, t, f, t, t);
allowedRec(obj4, ARUse) = mkBoolRec(f, t, t, f, t, f, t, t);
allowedRec(obj4, ARSupplement) = fRec;
allowedRec(obj4, ARBlock) = fRec;
allowedRec(obj4, ARUnblock) = fRec;

allowedRec(obj5, AROpen) = fRec;
allowedRec(obj5, ARClose) = fRec;
allowedRec(obj5, ARRead) = fRec;
allowedRec(obj5, ARModify) = mkBoolRec(f, t, t, f, t, f, t, t);
allowedRec(obj5, ARUse) = mkBoolRec(f, t, t, f, t, t, f, t);
allowedRec(obj5, ARSupplement) = fRec;
allowedRec(obj5, ARBlock) = fRec;
allowedRec(obj5, ARUnblock) = fRec;

allowedRec(obj6, AROpen) = fRec;
allowedRec(obj6, ARClose) = fRec;
allowedRec(obj6, ARRead) = mkBoolRec(f, t, t, t, f, t, t);
allowedRec(obj6, ARModify) = fRec;
allowedRec(obj6, ARUse) = mkBoolRec(f, t, t, t, t, f, t, t);
allowedRec(obj6, ARSupplement) = fRec;
allowedRec(obj6, ARBlock) = fRec;
allowedRec(obj6, ARUnblock) = fRec;

allowedRec(obj7, AROpen) = fRec;
allowedRec(obj7, ARClose) = fRec;
allowedRec(obj7, ARRead) = mkBoolRec(f, t, t, t, f, t, t);
allowedRec(obj7, ARModify) = fRec;
allowedRec(obj7, ARUse) = mkBoolRec(f, t, t, t, f, t, t);
allowedRec(obj7, ARSupplement) = fRec;
allowedRec(obj7, ARBlock) = fRec;
allowedRec(obj7, ARUnblock) = fRec;

allowedRec(obj8, AROpen) = fRec;
```

allowedRec(obj8,ARClose) = fRec;
allowedRec(obj8,ARRead) = fRec;
allowedRec(obj8,ARModify) = fRec;
allowedRec(obj8,ARUse) = mkBoolRec(f,t,t,t,t,f,t,t);
allowedRec(obj8,ARSupplement) = fRec;
allowedRec(obj8,ARBlock) = fRec;
allowedRec(obj8,ARUnblock) = fRec;

allowedRec(obj9,AROpen) = fRec;
allowedRec(obj9,ARClose) = fRec;
allowedRec(obj9,ARRead) = fRec;
allowedRec(obj9,ARModify) = fRec;
allowedRec(obj9,ARUse) = mkBoolRec(f,t,t,t,t,f,t,t);
allowedRec(obj9,ARSupplement) = fRec;
allowedRec(obj9,ARBlock) = fRec;
allowedRec(obj9,ARUnblock) = fRec;

allowedRec(obj10,AROpen) = fRec;
allowedRec(obj10,ARClose) = fRec;
allowedRec(obj10,ARRead) = mkBoolRec(f,t,t,t,t,f,t,t);
allowedRec(obj10,ARModify) = fRec;
allowedRec(obj10,ARUse) = fRec;
allowedRec(obj10,ARSupplement) = fRec;
allowedRec(obj10,ARBlock) = fRec;
allowedRec(obj10,ARUnblock) = fRec;

allowedRec(obj11,AROpen) = fRec;
allowedRec(obj11,ARClose) = fRec;
allowedRec(obj11,ARRead) = mkBoolRec(f,t,t,t,t,f,f,f);
allowedRec(obj11,ARModify) = mkBoolRec(f,f,t,t,t,f,f,f);
allowedRec(obj11,ARUse) = fRec;
allowedRec(obj11,ARSupplement) = fRec;
allowedRec(obj11,ARBlock) = fRec;
allowedRec(obj11,ARUnblock) = fRec;

allowedRec(obj12,AROpen) = fRec;
allowedRec(obj12,ARClose) = fRec;
allowedRec(obj12,ARRead) = mkBoolRec(f,t,t,t,t,f,t,t);
allowedRec(obj12,ARModify) = fRec;
allowedRec(obj12,ARUse) = mkBoolRec(f,t,t,t,t,f,t,t);
allowedRec(obj12,ARSupplement) = fRec;
allowedRec(obj12,ARBlock) = fRec;
allowedRec(obj12,ARUnblock) = fRec;

THEORYEND

#POS 600,822
THEORY TAutomaton
USING
TICC_Policy;
TTransitions
THEORYEND

#POS 635,394
THEORY TCard
PURPOSE
" Definition of predicates and functions that are related to ICCs "
USING TAnswer; TInitials
FUNCTIONS
/* Certificate Function for the TOE.
   This function produces a certificate combined with a valid response to a challenge.
   \1 contains the secret key needed to produce the response
   \2 is the challenge
*/
certIccKey: information, counter -> info.m;
/* Certificates generated with the initial key */
certIcc: counter -> info.m;

/* Answers to commands */
answerModified: info.m;
/* .. to a successfully completed command (OK response): */
answerSuccess: info.m;
/* .. to a successfully completed PIN authentication command (OK response): */
answerAuthSuccess: info.m;
/* .. to an Expiration event */
answerAuthExpired: info.m;
/* .. to a successfully completed PUK authentication command (OK response): */
answerAuthPUKSuccess: info.m;
/* .. to a failed command: */
answerAuthFailure: info.m;
/* .. to a failed non authentication command: */
answerAuthIfdSuccess: info.m;
/* .. to a failed authentication command: */
answerAuthFailure: info.m;
/* .. to a fatally failed command: */
answerFatalFailure: info.m;
/* .. to a command not allowed in the current state */
answerDenied: info.m;
/* .. to a power up reset */
answerReset: info.m;
/* .. after closing the signature application */
answerClosed: info.m;
/* .. after opening the signature application */
answerOpened: info.m;
/* .. to an unsuccessful user authentication because of a disabled card */
answerDisabled: info.m;

/* Message for the user: "Security Violation occurred" */
infoSecurityViolation: info.m;

/* Convert answer constant to information type */
answerInfo: answer -> info.m;
theAnswer: info.m -> answer

PREDICATES
/* Valid, iff \1 is an answer of the ICC to be sent to the terminal. */
isAnswer: info.m

VARS
c,d: counter;
s: subject;
i: info.m;
j,k: information;
a: answer

AXIOMS

FOR certIcc:
DEFFUNC certIcc(c) = certIccKey(initialIccAuthKey,c)

FOR isAnswer:
DEFPRED isAnswer(i) <-> answerInfo(theAnswer(i)) = i

/* Abbreviations */

FOR answerModified:
DEFFUNC answerModified = answerInfo(aModified)

FOR answerSuccess:
DEFFUNC answerSuccess = answerInfo(aSuccess)

FOR answerAuthSuccess:
DEFFUNC answerAuthSuccess = answerInfo(aAuthSuccess)

FOR answerAuthExpired:
DEFFUNC answerAuthExpired = answerInfo(aAuthExpired)

FOR answerAuthFailure:
DEFFUNC answerAuthFailure = answerInfo(aAuthFailure)

FOR answerAuthPUKSuccess:
DEFFUNC answerAuthPUKSuccess = answerInfo(aAuthPUKSuccess)

FOR answerAuthIfdSuccess:
DEFFUNC answerAuthIfdSuccess = answerInfo(aAuthIfdSuccess)

FOR answerFailure:
DEFFUNC answerFailure = answerInfo(aFailure)

FOR answerFatalFailure:
DEFFUNC answerFatalFailure = answerInfo(aFatalFailure)

FOR answerDenied:
DEFFUNC answerDenied = answerInfo(aDenied)

FOR answerReset:
DEFFUNC answerReset = answerInfo(aReset)

FOR answerClosed:
DEFFUNC answerClosed = answerInfo(aClosed)

FOR answerOpened:
DEFFUNC answerOpened = answerInfo(aOpened)

FOR answerDisabled:
DEFFUNC answerDisabled = answerInfo(aDisabled)

FOR infoSecurityViolation:
DEFFUNC infoSecurityViolation = answerInfo(iSecurityViolation)
OTHERS
/* Axioms about answers from the card */
theAnswer(answerInfo(a)) = a;
NOT isAnswer(noInfo);

/* All answers are classified as simple
information. Thus no secret informations
like the cardholder’s secret key can be
inferred from them. */
isSimple(answerInfo(a));

/* Certificate-responses of the ICC SigG application */
/* All certificate responses can be generated if the
key is known*/
knows(s,k) -> mKnows(s,certIccKey(k,c));

/* Certificates-responses cannot be inferred from each other */
mInferable(certIccKey(k,c), certIccKey(k,d)) -> c = d

THEORYEND

THEORY TClassification
PURPOSE "Classification of Information"
USING TInstInformation

TYPES
/* A type containing one element for each kind of secret used
   in the context of the TOE. */
kindSecret = FREELY GENERATED BY
/* Secret key to generate the cardholder’s
     signature */
kindSkCh |
/* Secret data to produce valid IFD certificates */
kindCertIfd |
/* Secret data to produce valid ICC certificates */
kindCertIcc |
/* the pin (or other user authentication data) */
kindPIN |
/* keys for secure messaging */
kindSecureMsg |
/* display Message */
kindDisplayMessage

PREDICATES
/* Classification of informations */
/* Commands valid for the SigG application */
isSigGCommand: info.m;
/* This predicate is valid for informations that need to be protected
    on their way between ICC and a public IFD. */
isIccSensitive: info.m;

/* isSecret is true, iff \2 is a secret of kind \1.
   This predicate is used to describe the mutual independence
   of the various secrets in a concise way. */
isSecret: kindSecret, info.m;
/* isPublic is true, iff \1 is an information that may
   be distributed publicly. */
This predicate is used to describe the uninferability of secret informations from public ones in a concise way. */
isPublic: info.m;
/* isSimple is true, iff \l is an information so 'small' that it can be expected to be known to virtually every subject. It is a subclass of public informations. Possible members of this class of information are short responses from the card like answerDenied. 'isSimple' is not valid for Signatures or signed messages, though it is not excluded that they are public. */
isSimple: info.m;
/* isSecretInferable is true, iff \l contains a secret of kind \k in the sense that such a secret is inferable from \l */
isSecretInferable: kindSecret, info.m

VARS
k,k1,k2: kindSecret;
i,j: info.m

AXIOMS
/* Classification of noInfo */
NOT isSigGCommand(noInfo);
NOT isIccSensitive(noInfo);
NOT isSecret(k,noInfo);
isSimple(noInfo);
/* Simple informations are public */
isSimple(i) -> isPublic(i);
/* Different secrets are independent of each other, i.e. a secret does not give a hint which allows to infer another secret. */
isSecret(k1,i) AND isSecret(k2,j) AND NOT i=j -> NOT mInferable(i,j);
/*Secrets cannot be inferred from public information */
isSecret(k,i) AND isPublic(j) -> NOT mInferable(j,i);
/* Secrets of different kinds are different. */
isSecret(k1,i) AND isSecret(k2,i) -> k1 = k2;
/* Definition of isSecretInferable */
isSecretInferable(k,i) <->
EX j: isSecret(k,j) AND mInferable(i,j);
/* Sensitiveness of secret informations. All informations exchanged between IFD and ICC are sensitive if they contain secrets. */
isSecretInferable(k,i) -> isIccSensitive(i)
THEORY THistory
USING
TInstEvent;
TCard;
TList[information]

FUNCTIONS
/* The answer belonging to the security related event \1 */
eventAnswer: event.m -> info.m;
/* The function allowedFailures takes a list of answers from an ICC
and returns the numbers of user authentication failures still allowed.
The number is set to the maximal value each time an authentication
succeeds. */
allowedFailures: list -> nat

PREDICATES
/* User is authenticated for internal output history */
histAuthUser: list;
/* IFD is authenticated for internal output history */
histAuthIfd: list;
/* Valid for all lists of outputs that correspond to answers
of an ICC which experienced more than the allowed amount
of failed authentication attempts. */
maxFailuresExceeded: list

VARS l: list

AXIOMS
FOR histAuthUser:
DEFPRED histAuthUser(l) <->
SWITCH 1 IN
CASE nil: FALSE
CASE cons:
IF def(first(l)) = answerAuthFailure THEN FALSE
ELSE IF def(first(l)) = answerReset THEN FALSE
ELSE IF def(first(l)) = answerClosed THEN FALSE
ELSE IF def(first(l)) = infoSecurityViolation THEN FALSE
ELSE IF def(first(l)) = answerAuthSuccess THEN TRUE
ELSE IF def(first(l)) = answerAuthPUKSuc-
cess THEN TRUE
ELSE histAuthUser(rest(l))
FI FI FI FI FI FI NI

FOR histAuthIfd:
DEFPRED histAuthIfd(l) <->
SWITCH 1 IN
CASE nil: FALSE
CASE cons:
IF def(first(l)) = answerReset THEN FALSE
ELSE IF def(first(l)) = answerClosed THEN FALSE
ELSE IF def(first(l)) = infoSecurityViolation THEN FALSE
ELSE IF def(first(l)) = answerAuthIfdSuccess THEN TRUE
ELSE IF def(first(l)) = answerAuthPUKSuc-
cess THEN TRUE
ELSE histAuthIfd(rest(l))
FI FI FI FI FI NI

FOR allowedFailures:
DEFFUNC allowedFailures(l) =
SWITCH 1 IN
CASE nil: initialMaxAuthChFailures
CASE cons:
  IF first(l) = value(answerAuthPUKSuccess) THEN initialMaxAuthChFailures
  ELSE IF first(l) = value(answerAuthSuccess) THEN initialMaxAuthChFailures
  ELSE IF first(l) = value(answerFailure) THEN pred(allowedFailures(rest(l)))
  ELSE IF first(l) = value(answerFatalFailure) THEN 0
  ELSE /* answers not relevant for authentication */
  allowedFailures(rest(l))
FI FI FI FI

NI

FOR maxFailuresExceeded:
  DEFPRED maxFailuresExceeded(l) <-> allowedFailures(l) <= 0

OTHERS

eventAnswer(noevent) = noinfo;
eventAnswer(def(sre1)) = answerReset;
eventAnswer(def(sre2)) = answerReset;
eventAnswer(def(sre3a)) = answerOpened;
eventAnswer(def(sre3b)) = answerOpened;
eventAnswer(def(sre4)) = answerClosed;
eventAnswer(def(sre5)) = answerAuthSuccess;
eventAnswer(def(sre6)) = answerAuthFailure;
eventAnswer(def(sre7)) = answerFatalFailure;
eventAnswer(def(sre8)) = answerAuthExpired;
eventAnswer(def(sre9)) = answerAuthIfdSuccess;
eventAnswer(def(sre10)) = infoSecurityViolation;
eventAnswer(def(sre11)) = answerAuthPUKSuccess;
eventAnswer(def(sre12)) = answerAuthFailure
THEORYEND

THEORY TICC
USING
TTerminal;
TCard;
TSecureConnection;
TSignature;
THistory
THEORYEND

THEORY TICC_Policy
USING
TICC;
TBucks;
boolean
THEORYEND

THEORY TInformation
PURPOSE
" The concept of information *
USING TSubject
TYPES
  /* A simple counter
  (We don’t use the predefined natural numbers
   because we don’t need addition etc.) */
  counter = FREELY GENERATED BY cNull | cSucc(cPred : counter);
Generic Formal Model of Security Policy
for Signature Creation SmartCards
Version: 1.1

/* Pieces of information (messages):
   This is the main type of this model.
   Any information exchanged between the subjects of this
   model has the type information (or a type derived from
   type information). This can be Documents to be signed,
   keys, commands, certificates etc.
*/

information

FUNCTIONS

/*::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::*/
/* Basic Functions of the theory */
/*::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::*/
/* \1 enhances its knowledge with \2 */
learns: subject, information -> subject;
/* The knowledge of \1 is enhanced with the
knowledge of \2 */
learnsall: subject, subject -> subject;
/* A subject that knows nothing */
ignorant: subject

PREDICATES

/* \1 is a place that holds (knows or has in its store)
   information \2 */
knows : subject, information ;
/* \1 and \2 are the same person/thing,
   but they may be different in what they know and
   their internal state but have the same identity.
   \1 and \2 can be regarded as the same subject
   watched at different points of time. */
identical: subject, subject;
/* subject \2 knows everything subject \1 knows */
_ << _ : subject, subject;
/* subject \1 and \2's knowledge is exactly the same */
_ == _ : subject, subject;
/* Information \1 can be used to infer \2,
   more precisely: there is a subject that
   knows \2 after learning \1 even though
   it did not know \2 before. */
inferable: information, information;
/* Information \1 can be used to infer \2
   even if \3 is unknown,
   more precisely: there is a subject that
   knows \2 after learning \1 even though
   it did not know \2 AND \3 before. */
inferableWithout: information, information, information

VARS
c,d: counter;
i, j, k: information;
s, t, u: subject

AXIOMS

/* Identity is an equivalence relation */
identical(s, s);
identical(s, t) and identical(t, u) -> identical(s, u);
identical(s, t) -> identical(t, s);

/* The Ignorant knows nothing */
not knows(ignorant, i);

/* Learning has no effects on the Identity */
identical(s, learns(s, i));
identical(s, learnsall(s, t));

/* Definition of « >> */
s << t <-> ALL i: knows(s, i) -> knows(t, i);

/* Description of how the knowledge changes if a subject learns something: The new knowledge is the ’smallest’ knowledge containing the old knowledge of s and the new piece of information i. */
s << t and knows(t, i) <-> learns(s, i) << t;

/* Definition of learnsall: */
s << learnsall(s, t);
t << learnsall(s, t);
s << u and t << u -> learnsall(s, t) << u;

/* Definition of identical knowledge */
s == t <-> s << t AND t << s;

/* Definition of inferable */
/* j is inferable from i, i.e. there is someone who doesn’t know j and can deduce j, if he learns i */
inferable(i, j) <->
EX s: NOT knows(s, j) AND knows(learns(s, i), j);

/* Definition of inferableWithout */
inferableWithout(i, j, k) <->
EX s: NOT knows(s, j) AND NOT knows(s, k) AND knows(learns(s, i), j);

/* No subject can know everything */
ALL s: EX i: NOT knows(s, i)

THEORYEND

#POS 635,217
THEORY TInitials
PURPOSE " Declaration and classification of initial values "

USING TClassification;natural

FUNCTIONS

/*Initial Values used for personalisation of the ICC */
/* The initial PIN */
initialPin : information;
/* The initial PUK */
initialPuk: information;
/* The initial cardholder’s credentials non CA part */
initialCHCredentials: information;
/* The initial cardholder’s credentials CA part */
initialCHCredentialsCA: information;
/* The initial authentication key of the ICC */
initialIccAuthKey: information;
/* The initial authentication credentials of the ICC */
initialIccAuthCred: information;
/* The initial authentication credentials of IFDs */
initialIfdAuthCred: information;
/* The initial display message */
initialDisplayMessage: information;
/* The initial credentials for other signatures. */
initialOtherCred: information;

/* The initial value for the counter used for cardholders signature key generation */
initialSKeyCounter: counter;

/* Maximum of successive card holder authentication failures by PUK. The SigG functionality of the ICC is disabled thereafter until successful authentication with PUK. */
initialMaxAuthChFailures: nat;
/* Maximum of successive card holder authentication failures by PUK. PUK-Authentication is disabled thereafter. */
initialMaxAuthPukFailures: nat

AXIOMS

/* Secret Constants */
isSecret(kindPIN,def(initialPin));
isSecret(kindPIN,def(initialPuk));
isSecret(kindDisplayMessage,def(initialDisplayMessage));
isSecret(kindCertIcc,def(initialIccAuthKey));

/* Public Constants */
isPublic(def(initialCHCredentials));
isPublic(def(initialCHCredentialsCA));
isPublic(def(initialIfdAuthCred));
isPublic(def(initialIccAuthCred));
isPublic(def(initialOtherCred));

NOT isSimple(def(initialCHCredentials));
NOT isSimple(def(initialCHCredentialsCA));
NOT isSimple(def(initialIfdAuthCred));
NOT isSimple(def(initialIccAuthCred));
NOT isSimple(def(initialOtherCred))

THEORYEND

#POS 487,396
THEORY TInstEvent
PURPOSE "Instantiation of TMaybe with TEvent"

USING TEvent;
  event = TMaybe[events]

FUNCTIONS
  /* Constant used in case there is no security related event. */
  noEvent: event.m

AXIOMS
  FOR noEvent: DEFFUNC noEvent = event.undef
THEORYEND

#POS 634,95
THEORY TInstInformation
  PURPOSE "Instantiation of TMaybe with TInformation: Generalised information"

USING TInformation;
  info = TMaybe[information]

FUNCTIONS
  /* Constants for undefined informations or default values. 
     For example currently unused input/output channels */
  noKey: info.m;
  noInfo: info.m;

  /* Functions carried over from TInformation */
  mLearns: subject, info.m -> subject

PREDICATES
  /* Predicates carried over from TInformation */
  mKnows: subject, info.m;
  mInferable: info.m, info.m;
  mInferableWithout: info.m, info.m, info.m

VARS i,j,k: info.m;
  s: subject

AXIOMS
  FOR noInfo: DEFFUNC noInfo = info.undef
  FOR noKey: DEFFUNC noKey = info.undef

  FOR mKnows: DEFPRED
    mKnows(s,i) <-> SWITCH i IN
      CASE undef: TRUE
      CASE def: knows(s,value(i))
    NI

  FOR mLearns: DEFFUNC
    mLearns(s,i) = SWITCH i IN
      CASE undef: s
      CASE def: learns(s,value(i))
    NI

  FOR mInferable: DEFPRED
    mInferable(i,j) <-> SWITCH i IN
      CASE undef: FALSE
      CASE def:
        SWITCH j IN
          ...
CASE undef: FALSE
CASE def: inferable(value(i), value(j))

FOR mInferableWithout: DEFPRED
mInferableWithout(i, j, k) <->
SWITCH i IN
  CASE undef: FALSE
  CASE def:
    SWITCH j IN
      CASE undef: FALSE
      CASE def:
        SWITCH k IN
          CASE undef: inferable(value(i), value(j))
          CASE def: inferableWithout(value(i), value(j), value(k))

THEORYEND

#POS 743,816
THEORY TObject
USING
  natural;
  TAccessrights;
  TICC_Policy

FUNCTIONS
  /* The name of the Object */
  obj : ObjectIds;
  /* The initial value of the variable named value */
  initialValue : information

THEORYEND

#POS 1020,392
THEORY TSecureConnection
PURPOSE
  "Definitions for secure messaging"

USING
  TClassification

FUNCTIONS
  /* Get the public key for secure message channel encryption
     attached to the certificate \1 */
  extractSecureMsgKey: info.m -> info.m;

  /* Get the public key for secure message signature
     attached to the certificate \1 */
  extractSecureMsgKeySig: info.m -> info.m;

  /* Secure message encryption with key \1 and
     signing with key \2 */
  secureMsgEncode: info.m, info.m, info.m -> info.m;

  /* Secure message decryption with key \1
     and signature checking with key \2 */
If the signature is invalid the result is noInfo. */

secureMsgDecode: info.m, info.m, info.m -> info.m

PREDICATES
/* \1 is a secret key and \2 the matching public
key to \1 for secure message channel encryption. */
validSecureMsgPair: info.m, info.m;

/* \1 is a secret key and \2 the matching public
key to \1 for secure message channel signatures. */
validSecureMsgPairSig: info.m, info.m

VARS
s,t,u: subject;
i,j,k,sk,pk,sksig,pksig: info.m

AXIOMS
/* Secure messages can be decoded */
validSecureMsgPair(sk,pk) AND validSecureMsgPairSig(sksig,pksig) -> secureMsgDecode(sk,pksig,secureMsgEncode(pk,sksig,i)) = i;

/* An invalid signature can be detected */
NOT validSecureMsgPair(sksig,pksig) AND pksig /= noKey -> secureMsgDecode(sk,pksig,secureMsgEncode(pk,sksig,i)) = noInfo;

/* If keys and content are known the secret message is known as well and vice versa */
mKnows(s,i) AND mKnows(s,j) AND mKnows(s,k) -> mKnows(s,secureMsgEncode(i,j,k)) AND mKnows(s,secureMsgDecode(i,j,k));

/* The content of Secure messages cannot be guessed */
validSecureMsgPair(sk,pk) -> NOT mInferableWithout(secureMsgEncode(pk,j,i),i,sk);

/* Secure messages reveal sensitive informations only to possessors of the secret encryption key (if at all) */
validSecureMsgPair(sk,pk) AND mInferableWithout(secureMsgEncode(pk,sksig,i),j,sk) -> NOT isIccSensitive(j);

/* Secure messages do not reveal anything but what is inferable from the the message in clear */
validSecureMsgPair(sk,pk) AND mInferable(secureMsgEncode(pk,sksig,i),j) -> mInferable(i,j);

/* secureMsgEncode is a signature function: */
validSecureMsgPairSig(sk,pk) -> mInferableWithout(i,secureMsgEncode(j,sk,k),sk) -> i = sk OR i = secureMsgEncode(j,sk,k);

validSecureMsgPairSig(sk,pk) -> secureMsgEncode(j,sk,k) /= sk;
/* Decode and encode without keys leave the message unchanged */
secureMsgDecode(noKey, noKey, i) = i;
secureMsgEncode(noKey, noKey, i) = i;

/* Key pairs for secure messaging consist of a secret and some public information */
validSecureMsgPair(sk, pk) -> isSecret(kindSecureMsg, sk);
validSecureMsgPairSig(sksig, pksig) -> isSecret(kindSecureMsg, sksig);
validSecureMsgPair(sk, pk) -> isPublic(pk);
validSecureMsgPairSig(sksig, pksig) -> isPublic(pksig)

THEORYEND

#POS 841,392
THEORY TSignature
PURPOSE
"Signatures and keys"
USING TClassification /* includes TInformation */
FUNCTIONS
/* The signature function: The result is \1 signed with key \2 */
sig : information, information -> information;
/* The signature function extended on generalised information: */
mSig: info.m, info.m -> info.m;
/* The key generation function: The result is the \1-th secret key. */
skeygen: counter -> information;
/* The result is the \1-th public key. */
pkeygen: counter -> information
PREDICATES
/* Valid pair of private and secret key */
validpair : information, information;
/* Signature verification: t, iff signature is generated with corresponding secret key */
validSig: information, information;
mValidSig: info.m, info.m
VARS
i, j, k, sk, pk : information;
s : subject;
c : counter;
mSk, mPk: info.m;
mI: info.m

AXIOMS
/* Signatures on generalised informations */
FOR mSig: DEFFUNC
mSig(mSk, mI) = SWITCH mSk IN
CASE undef: undef
CASE def:
  SWITCH mI IN
    CASE undef: undef
      CASE def: mI
    CASE undef: undef
      CASE def: mI
    CASE undef: undef
      CASE def: mI
    CASE undef: undef
      CASE def: mI
    CASE undef: undef
      CASE def: mI
    CASE undef: undef
      CASE def: mI
    CASE undef: undef
      CASE def: mI
    CASE undef: undef
      CASE def: mI
    CASE undef: undef
      CASE def: mI
    CASE undef: unde
CASE def: def(sig(value(mSk),value(mI)))

FOR mValidSig: DEFPRED
   mValidSig(mI,mPk) <->
   IF isDef(mI) AND isDef(mPk) THEN validSig(value(mI),value(mPk))
   ELSE FALSE FI

OTHERS
   /* The following axioms define properties of sig
      that are necessary for SO7.2.
      (See also SSignature) */

   /* If a piece of information has been signed,
      the original information still can be inferred from
      it. */
   validpair(sk,pk) AND knows(s,sig(i,sk)) -> knows(s,i);

   /* If we have a piece of information and key we
      also know the signed piece of information. */
   knows(s,i) and knows(s,sk) -> knows(s,sig(i,sk));

   /* Signed documents are different if the original
      documents are different */
   sig(i,sk) = sig(j,sk) -> i = j;

   /* Signing a document never generates the secret key */
   validpair(sk,pk) -> sig(i,sk) /= sk;

   /* Signtatures must be different from the signed document */
   validpair(sk,pk) -> sig(i,sk) /= i;

   /* The following axiom essentially formalises SO7.3.
      It is circumscribed as follows:
      If a subjects knows a signature after receiving a
      message i, at least one of the following conditions
      must hold:
      - the subject already knew the signature
      - the subject already knew the secret key
      - the message i is the signature (or the signed message)
      - the message i is the secret key
      The axiom also is required for SO7.2.
      Taking into account (see above)
      sig(i,sk) /= sk AND
      knows(s,i) and knows(s,sk) -> knows(s,sig(i,sk))
      one can also derive that the secret key cannot be
      inferred from the result of sig (provided it
      is not already contained in the unsigned message).
      SO7.2 is formulated in SSignature, a theory
      connected to this one by a 'satisfies'-link.
      The proof of the proof obligations stemming from
      this link show that TSignature actually covers
      SO7.2. */
   validpair(sk,pk) AND
   knows(learns(s,i),sig(j,sk)) ->
   knows(s,sk) OR
   knows(s,sig(j,sk)) OR

146 / 153 12. 9. 2000
i = sk OR i = sig(j, sk);

/* The secret key cannot be inferred 
from the public key */
validpair(sk, pk) ->
NOT inferable(pk, sk);

/* validpairs consist of a secret and some public information */
validpair(sk, pk) -> isSecret(kindSkCh, info.def(sk));
validpair(pk, pk) -> isPublic(info.def(pk));

/* The key generation function generates valid keys */
validpair(skeygen(c), pkeygen(c));

/* Signature verification */
validpair(sk, pk) ->
(validSig(i, pk) <=> EX j: i = sig(j, sk));

/* Signatures are not simple */
validpair(sk, pk) -> not isSimple(mSig(mI, def(sk)));

SATISFIES SSsignature
THEORYEND

THEORY TSubject
PURPOSE
" Theory of Subjects "
TYPES subject
THEORYEND

THEORY TTerminal
PURPOSE
" Predicates and functions that are related to IFDs"
USING TCommand; TInitials
FUNCTIONS
/* Commands sent by the IFD converted to type info.m */
commandInfo: ifdCommand -> info.m;
/* Selector for commands 'encoded' in type info.m */
theIfdCommand: info.m -> ifdCommand;

/* Command to sign the information given as 
 parameter \1 */
ifdDoSign: information -> info.m;
/* Command to do User Authentication with PIN, \1 is the PIN */
ifdDoAuth: information -> info.m;
/* Command to do User Authentication with PUK, \1 is the PUK, \2 the new PIN */
ifdDoPUKAuth: information, information -> info.m;
/* Command to do User Authentication and change the password, \1 is the old PIN, \2 the new one */
ifdDoAuthChange: information, information -> info.m;
/* Command to change the display message */
ifdDoMessageChange: information -> info.m;
/* Command to authenticate the ICC */
ifdDoIntAuth: counter -> info.m;
    /* Command to authenticate the IFD */
ifdDoExtAuth: information -> info.m;
    /* Command to exit (close) the SigG application */
ifdDoExit: info.m;
    /* Command to start (open) the SigG application */
ifdDoStart: info.m;
    /* Command to do signature verification
     with key reference \1 on signed message \2 */
ifdDoVerify: information, information -> info.m;
    /* Commands to modify objects */
ifdDoModifyObj1: information -> info.m;
ifdDoModifyObj2: information -> info.m;
ifdDoModifyObj3: information -> info.m;
ifdDoModifyObj4: information -> info.m;
ifdDoModifyObj5: information -> info.m;
ifdDoModifyObj6: information -> info.m;
ifdDoModifyObj7: information -> info.m;
ifdDoModifyObj8: information -> info.m;
ifdDoModifyObj9: information -> info.m;
ifdDoModifyObj10: information -> info.m;
ifdDoModifyObj11: information -> info.m;
ifdDoModifyObj12: information -> info.m;
    /* Commands to read Objects */
ifdDoReadObj1: info.m;
ifdDoReadObj2: info.m;
ifdDoReadObj3: info.m;
ifdDoReadObj4: info.m;
ifdDoReadObj5: info.m;
ifdDoReadObj6: info.m;
ifdDoReadObj7: info.m;
ifdDoReadObj8: info.m;
ifdDoReadObj9: info.m;
ifdDoReadObj10: info.m;
ifdDoReadObj11: info.m;
ifdDoReadObj12: info.m

PREDICATES
    /* IFD commands given to the card */
isIfdCommand: info.m;
    /* The IFD commands belonging to the SigG application */
isSigGCommand: information;
    /* Checking certificates:
The given information is a certificate
for an accredited IFD based on
initialIfdAuthCred. */
isCertIfd: info.m;
    /* The given information \2 is a certificate
for an accredited IFD based on
key \1 */
isCertIfdKey: info.m, info.m

VARS i, i1, i2, j, j1, j2: information;
    mi: info.m;
c, c1, c2: counter;
    cmd: ifdCommand

AXIOMS
FOR isCertIfd:
DEFPRED isCertIfd(mi) <-> isCertIfdKey(def(initialIfdAuthCred),mi)
FOR isSigGCommand:
DEFPRED isSigGCommand(mi) <-> mi = commandInfo(theIfdCommand(mi))

/* Abbreviations */
FOR ifdDoSign:
DEFFUNC ifdDoSign(i) = commandInfo(Sign(i))
FOR ifdDoAuth:
DEFFUNC ifdDoAuth(i) = commandInfo(Auth(i))
FOR ifdDoPUKAuth:
DEFFUNC ifdDoPUKAuth(i,j) = commandInfo(PUKAuth(i,j))
FOR ifdDoAuthChange:
DEFFUNC ifdDoAuthChange(i,j) = commandInfo(AuthChange(i,j))
FOR ifdDoMessageChange:
DEFFUNC ifdDoMessageChange(i) = commandInfo(MessageChange(i))
FOR ifdDoExtAuth:
DEFFUNC ifdDoExtAuth(i) = commandInfo(ExtAuth(i))
FOR ifdDoIntAuth:
DEFFUNC ifdDoIntAuth(c) = commandInfo(IntAuth(c))
FOR ifdDoExit:
DEFFUNC ifdDoExit = commandInfo(Exit)
FOR ifdDoStart:
DEFFUNC ifdDoStart = commandInfo(Start)
FOR ifdDoVerify:
DEFFUNC ifdDoVerify(i,j) = commandInfo(Verify(i,j))

FOR ifdDoModifyObj2:
DEFFUNC ifdDoModifyObj2(i) = commandInfo(ModifyObj2(i))
FOR ifdDoModifyObj3:
DEFFUNC ifdDoModifyObj3(i) = commandInfo(ModifyObj3(i))
FOR ifdDoModifyObj4:
DEFFUNC ifdDoModifyObj4(i) = commandInfo(ModifyObj4(i))
FOR ifdDoModifyObj5:
DEFFUNC ifdDoModifyObj5(i) = commandInfo(ModifyObj5(i))
FOR ifdDoModifyObj6:
DEFFUNC ifdDoModifyObj6(i) = commandInfo(ModifyObj6(i))
FOR ifdDoModifyObj7:
DEFFUNC ifdDoModifyObj7(i) = commandInfo(ModifyObj7(i))
FOR ifdDoModifyObj8:
DEFFUNC ifdDoModifyObj8(i) = commandInfo(ModifyObj8(i))
FOR ifdDoModifyObj9:
DEFFUNC ifdDoModifyObj9(i) = commandInfo(ModifyObj9(i))
FOR ifdDoModifyObj10:
DEFFUNC ifdDoModifyObj10(i) = commandInfo(ModifyObj10(i))
FOR ifdDoModifyObj11:
DEFFUNC ifdDoModifyObj11(i) = commandInfo(ModifyObj11(i))
FOR ifdDoModifyObj12:
DEFFUNC ifdDoModifyObj12(i) = commandInfo(ModifyObj12(i))

FOR ifdDoReadObj2:
DEFFUNC ifdDoReadObj2 = commandInfo(ReadObj2)
FOR ifdDoReadObj3:
DEFFUNC ifdDoReadObj3 = commandInfo(ReadObj3)
FOR ifdDoReadObj4:
DEFFUNC ifdDoReadObj4 = commandInfo(ReadObj4)
FOR ifdDoReadObj5:
DEFFUNC ifdDoReadObj5 = commandInfo(ReadObj5)
FOR ifdDoReadObj6:
DEFFUNC ifdDoReadObj6 = commandInfo(ReadObj6)
FOR ifdDoReadObj7:
  DEFFUNC ifdDoReadObj7 = commandInfo(ReadObj7)
FOR ifdDoReadObj8:
  DEFFUNC ifdDoReadObj8 = commandInfo(ReadObj8)
FOR ifdDoReadObj9:
  DEFFUNC ifdDoReadObj9 = commandInfo(ReadObj9)
FOR ifdDoReadObj10:
  DEFFUNC ifdDoReadObj10 = commandInfo(ReadObj10)
FOR ifdDoReadObj11:
  DEFFUNC ifdDoReadObj11 = commandInfo(ReadObj11)
FOR ifdDoReadObj12:
  DEFFUNC ifdDoReadObj12 = commandInfo(ReadObj12)

OTHERS
/* Certificates */
NOT isCertIfdKey(mi,noKey);
/* IFD Commands */
isIfdCommand(commandInfo(cmd));
theIfdCommand(commandInfo(cmd)) = cmd;

/* IfdCommands are well defined values */
NOT isIfdCommand(noInfo);
/* Sensitivity of certain IFD commands:
The following axioms prevent acception of commands to sign information arriving through insecure channels (in case of a public IFD) */
isIccSensitive(ifdDoSign(i));
/* Authentication must not be done on insecure channels (including setting new PINs) */
isIccSensitive(ifdDoAuth(i));
isIccSensitive(ifdDoPUKAuth(i,j));
isIccSensitive(ifdDoAuthChange(i,j));
isIccSensitive(ifdDoMessageChange(i));

/* The parameters contain the information exchanged with commands: Anything inferable from the parameters can be inferred from the command. */
inferable(i,j) <-> mInferable(ifdDoSign(i),def(j));
inferable(i,j) <-> mInferable(ifdDoAuth(i),def(j));
inferable(i1,j) OR inferable(i2,j) <-> mInferable(ifdDoAuthChange(i1,i2),def(j))
THEORYEND

THEORY TTransitions
PURPOSE "Representation of the state transition table"
USING TStates; TInstEvent

FUNCTIONS
/* The result is the next if event \2 occurs while in state \1 */
nextstate: states, event.m -> states

150 / 153 12. 9. 2000
VARS s: states;
e: event.m

AXIOMS
/* Representation of table 13 in the target. */
FOR nextstate:
  DEFFUNC nextstate(s,e) =
    SWITCH e IN
      CASE undef: s
      CASE def:
        SWITCH s IN
          CASE cas1:
            SWITCH value(e) IN
              CASE sre1: CAS1
              CASE sre2: CAS1
              CASE sre3a: CAS2
              CASE sre3b: CAS7
              CASE sre4: CAS1
              CASE sre5: CAS1
              CASE sre6: CAS1
              CASE sre7: CAS1
              CASE sre8: CAS1
              CASE sre9: CAS1
              CASE sre10: CAS6
              CASE sre11: CAS1
              CASE sre12: CAS1
              NI
          CASE cas2:
            SWITCH value(e) IN
              CASE sre1: CAS1
              CASE sre2: CAS1
              CASE sre3a: CAS1
              CASE sre3b: CAS1
              CASE sre4: CAS1
              CASE sre5: CAS3
              CASE sre6: CAS2
              CASE sre7: CAS7
              CASE sre8: CAS1
              CASE sre9: CAS4
              CASE sre10: CAS6
              CASE sre11: CAS3
              CASE sre12: CAS2
              NI
          CASE cas3:
            SWITCH value(e) IN
              CASE sre1: CAS1
              CASE sre2: CAS1
              CASE sre3a: CAS1
              CASE sre3b: CAS1
              CASE sre4: CAS1
              CASE sre5: CAS3
              CASE sre6: CAS2
              CASE sre7: CAS7
              CASE sre8: CAS2
              CASE sre9: CAS1
              CASE sre10: CAS6
              CASE sre11: CAS3
              CASE sre12: CAS2
              NI

12.9.2000
CASE cas4:
  SWITCH value(e) IN
  CASE sre1: CAS1
  CASE sre2: CAS1
  CASE sre3a: CAS1
  CASE sre3b: CAS1
  CASE sre4: CAS1
  CASE sre5: CAS5
  CASE sre6: CAS4
  CASE sre7: CAS8
  CASE sre8: CAS1
  CASE sre9: CAS1
  CASE sre10: CAS6
  CASE sre11: CAS5
  CASE sre12: CAS4
  NI
CASE cas5:
  SWITCH value(e) IN
  CASE sre1: CAS1
  CASE sre2: CAS1
  CASE sre3a: CAS1
  CASE sre3b: CAS1
  CASE sre4: CAS1
  CASE sre5: CAS5
  CASE sre6: CAS4
  CASE sre7: CAS8
  CASE sre8: CAS4
  CASE sre9: CAS1
  CASE sre10: CAS6
  CASE sre11: CAS5
  CASE sre12: CAS4
  NI
CASE cas6:
  SWITCH value(e) IN
  CASE sre1: CAS6
  CASE sre2: CAS6
  CASE sre3a: CAS6
  CASE sre3b: CAS6
  CASE sre4: CAS6
  CASE sre5: CAS6
  CASE sre6: CAS6
  CASE sre7: CAS6
  CASE sre8: CAS6
  CASE sre9: CAS6
  CASE sre10: CAS6
  CASE sre11: CAS6
  CASE sre12: CAS6
  NI
CASE cas7:
  SWITCH value(e) IN
  CASE sre1: CAS1
  CASE sre2: CAS1
  CASE sre3a: CAS1
  CASE sre3b: CAS1
  CASE sre4: CAS1
  CASE sre5: CAS6
  CASE sre6: CAS6
  CASE sre7: CAS7
CASE sre8: CAS1
CASE sre9: CAS8
CASE sre10: CAS6
CASE sre11: CAS3
CASE sre12: CAS7
  NI
  CASE cas8:
    SWITCH value(e) IN
    CASE sre1: CAS1
    CASE sre2: CAS1
    CASE sre3a: CAS1
    CASE sre3b: CAS1
    CASE sre4: CAS1
    CASE sre5: CAS6
    CASE sre6: CAS6
    CASE sre7: CAS8
    CASE sre8: CAS1
    CASE sre9: CAS1
    CASE sre10: CAS6
    CASE sre11: CAS5
    CASE sre12: CAS8
    NI
  NI
  NI
THEORYEND