Verification of a Formal Security Model for Multiapplicative Smart Cards*

Gerhard Schellhorn[1], Wolfgang Reif[1], Axel Schairer[2],
Paul Karger[3], Vernon Austel[3], and David Toll[3]

[1] Universität Augsburg, Lehrstuhl Softwaretechnik und Programmiersprachen, D-86135 Augsburg
[2] DFKI GmbH, Stuhlsatzenhausweg 3, D-66123 Saarbrücken
[3] IBM T.J. Watson Research Center, 30 Saw Mill River Rd., Hawthorne, NY 10532

Abstract. We present a generic formal security model for operating systems of multiapplicative smart cards. The model formalizes the main security aspects of secrecy, integrity, secure communication between applications and secure downloading of new applications. The model satisfies a security policy consisting of authentication and intransitive noninterference. The model extends the classical security models of Bell/ LaPadula and Biba, but avoids the need for *trusted processes*, which are not subject to the security policy by incorporating such processes directly in the model itself. The correctness of the security policy has been formally proven with the VSE II system.

1 Introduction

Smart cards are becoming more and more popular. Compared to magnetic stripe cards they have considerable advantages. They may not only store data, that can be read and changed from a terminal, but they can also store executable programs. Therefore, anything that can be done with an ordinary computer can be done with a smart card. Their usefulness is limited only by available memory and computational power.

Currently, smart cards used in electronic commerce are single application smart cards: they store applications (usually only one) developed by a *single* provider. The scenario we envision for the future is that of multiapplicative smart cards, where several independent providers, maybe even competitors, have applications (i.e. collections of programs and data files to achieve a certain task) on a single smart card.

As an example, consider three applications: An airline A, which manages electronic flight tickets with the smart card, and two hotel chains H and I which use the smart card as an electronic door opener. A customer would carry the smart card around and show it whenever he visits one of H, I or flies with A. Of course none of the application providers would like to trust the others,

* Augsburg and DFKI research sponsored by the German Information Security Agency (BSI)

F. Cuppens et al. (Eds.): ESORICS 2000, LNCS 1895, pp. 17–36, 2000.

especially H would not trust his competitor I. Therefore the applications should be completely separate: none of the data H stores for opening doors should be visible or modifiable by I or A.

If two application providers agree, communication should also be possible: Airline A could have a loyalty scheme with H (see Fig. 1), or even with both H and I. Staying in a hotel of H earns a customer loyalty points, which reduces the price to fly with A, but that information must not be available to I. Establishing new communication channels and adding new applications should be possible dynamically: e.g. visiting his bank B, the card holder should be able to add an electronic wallet.

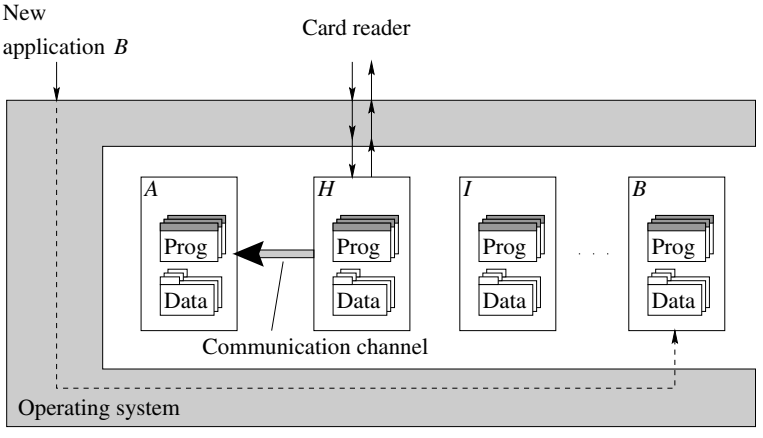

Fig. 1. An example scenario for a multiapplicative smart card

Of course such a scenario raises considerable security issues: How can applications be isolated from each other (e.g. H and its competitor I)? If applications want to communicate, how can communication be allowed without having unwanted information flow (e.g. I should not be able to see loyalty points moving from H to A)? How can it be guaranteed that a dynamically loaded new application does not corrupt existing ones?

In this paper, we present a formal security model that solves these questions for smart cards, as well as cell phones, PDAs, or larger systems. We will model an operating system which executes system calls ("commands"). These system calls are made by applications programs running in user mode on the smart card to an operating system running in supervisor mode on the smart card. They are fundamentally different from the commands defined in ISO/IEC 7816-4 [6] that are commands sent from the outside world to the smart card. The security conditions attached to the commands obey a security policy, which is suitable to solve the problems discussed above. The commands are chosen to be as abstract as possible: There is a command to register authentication information (e.g. a public key) for a new application (one application might have several keys to be

able to structure its data into several levels of secrecy and integrity), commands to load and to delete an application program, and file access commands to create, read, write and delete data files. Finally a command to change the secrecy and integrity of a file (according to the security policy) is provided.

The design of the formal security model was influenced by the informal security model [9] developed as part of IBM Research Division's on-going development of a high assurance smart card operating system for the Philips SmartXA chip [15]. The SmartXA is the first smart card chip to have hardware support for supervisor/user modes and a memory management unit. Some more information on potential applications of multiapplicative smart cards with this security model are given in [10]. Our security model was designed to be a generic abstraction from the IBM model that should be useful for other Smart Card providers too. It is compliant with the requirements for an evaluation of operating systems according to the ITSEC evaluation criteria [8] E4 or higher (and comparable Common Criteria [7] EAL5 or higher). The IBM system is designed for even higher assurance levels – ITSEC E5 or E6 or Common Criteria EAL6 or EAL7.

This paper is structured as follows: Sect. 2 describes the security objectives. Sect. 3 introduces the concepts to be implemented on the smart card which are used to achieve the security objectives. We informally define a mandatory security policy based on intransitive noninterference and authentication. Sect. 4 sketches the theory of intransitive noninterference as defined in [16] and explains some extensions. The data structures and commands of the system model are defined in Sect. 5. Sect. 6 discusses how to use the commands of the model for the example above. Sect. 7 discusses the formal verification of the security policy for the model. Sect. 8 compares some of the features of the informal IBM model [9] and the formal model. Finally, Sect. 9 concludes the paper.

2 Security Objectives

The security of a smart card is threatened by a variety of attacks, ranging from physical analysis of the card and manipulated card readers to programs circumventing the OS's security functions, or the OS security functions themselves revealing or changing information that is not intended (covert channels). In the model described in this paper we will be concerned with the question whether the operating system's functionality on the level of (abstract) system calls can guarantee the security requirements. We will therefore assume that operating system calls are atomic actions (this should be supported by hardware, e.g. a supervisor mode of the processor). We will not address physical threats to the card itself or to the card readers. Also, threats on the level of, e.g. memory reuse or register usage in the machine code will not be considered.

Our model addresses the following security objectives.

O1: Secrecy/integrity between programs of the same or different applications.
O2: Secure communication between applications.
O3: Secure downloading of code.

Application providers need to have guarantees that the data and programs they store on the card and pass to and receive from card readers are handled such that their secrecy and integrity is guaranteed: their code and data neither be observable nor alterable by other programs. This guarantees that secret data produced by one application's program cannot be leaked to another application, and that programs of one application can not be crashed by other application's corrupt data. Some application providers will additionally want to classify their programs and data into different levels of security and integrity. The OS should also guarantee that data are handled in a way that respects these different security and integrity levels. This is useful, e.g., in case a small number of programs operate on highly sensitive data and the rest only operate on insensitive data. In this case only a small fraction of the code has to be checked to handle the sensitive data correctly, because the bulk of the programs are guaranteed by the OS not to be able to access the sensitive data at all.

Some application providers will want their programs to exchange data with other applications in a controlled way, i.e. only in a way that has been mutually agreed on by the providers. The objective of secure communication asserts that data can only be communicated between applications if both application providers agree with the communication and with the form in which the data is communicated. It also implies that the communication cannot be observed or manipulated by other programs. We only consider communication through storage channels, timing channels are not in the scope of the model.

Our model does not impose restrictions on transitive communications: if application A sends some information to B (which assumes that they both have agreed), it no longer has any influence on what B does with the information. B could send it to any application C, even one hostile to A, if it has agreed with C to do so. Since it is not clear that this problem can be addressed technically we assume that providers who have explicitly agreed to exchange data have set up contracts to prevent such potential fraud by non-technical means.

Secure communication of application programs with card readers should be supported by the OS. However, this involves security considerations for devices outside the card and is not in the scope of the model described in this paper: we assume that a reliable communication between programs and card readers is implemented by the programs but will not consider the OS services needed to achieve this. Using the terminology of [19] we define all applications on the card to be within the security perimeter (to be in the "controlled application set"), while all devices outside the card are untrusted subjects.

All the objectives described above should not be threatened by programs that are dynamically loaded onto the card.

3 Security Concepts

In this section we define the concepts for security, which should be implemented on the smart card as an infrastructure. We define security claims over these

concepts, which ensure that the 3 security objectives given in the previous section are met. The security claims will be proven formally for our model.

Secrecy and integrity (objective O1) are common objectives of security models. Usually variants of the well-known mandatory security models of Bell/ LaPadula [2] and Biba [3] are used for this purpose. We assume the reader is familiar with them and their use of access classes (consisting of an access level and a set of access categories) to define the secrecy and integrity classification of files (objects) as well as the clearance of subjects. In our case applications will act as subjects. They will get disjoint sets of access categories (in the simplest case, application names and access categories coincide). This results in separated applications, where communication is completely prohibited.

One problem with this classical approach is that adding communication channels (objective O2) in such a Bell/LaPadula model will violate the security policy (simple security and *-property). Of course it is possible to add "trusted processes" (like it was done in the Multics-instance of the Bell/LaPadula [2] or in [12]), which are considered to be outside the model (i.e. properties of the security policy are proved ignoring them). But one of our main security objectives is to include such secure communication in the verified model.

Our solution to this problem consists of two steps. The first part is to use the following idea from the IBM operating system [9] (similar ideas are also given in [17] and [12]): Instead of giving a subject two access classes (*icl*,*scl*) as clearance (one for integrity and one for secrecy), we define the clearance of a subject to be four access classes (*ircl*,*srcl*,*iwcl*,*swcl*): The first two are used in reading operations, the other two in writing operations.

Usual application programs will have the same access classes for reading and writing ($ircl = iwcl$ and $srcl = swcl$). A communication channel from application A to application B is realized by a special program, called a *channel program* with two different pairs of access classes: the pair used for reading will have the clearance of A, while the one used for writing will have the clearance of B. This will allow the channel to read the content of a file from A and to write it into a file, which can then be read by B.

The second part consists in defining a new security policy, which generalizes the one of the Bell/LaPadula and Biba model. We will show, that the model satisfies the following security policy:

> A subject A with clearance (ircl_A,iwcl_A,srcl_A,swcl_A) can transfer information to a subject B with clearance (ircl_B,iwcl_B,srcl_B,swcl_B) if and only if $\mathrm{iwcl}_A \geq \mathrm{ircl}_B$ and $\mathrm{swcl}_A \leq \mathrm{srcl}_B$

Formally, we will prove that our security model is an instance of an intransitive noninterference model. Corollaries from this fact are that without communication channels, the model is an instance of the Bell/LaPadula as well as of the Biba model (objective O1) and that if a channel is set up as described above, it exactly allows communication from A to B (objective O2). The proof also implies that our model is free of covert storage channels. This is in contrast to pure Bell/LaPadula-like models, which require an extra analysis for covert storage channels (see [13]).

To accommodate secure downloading of applications (and of channel programs; objective O3), we have to add a second concept to our model: authentication. We will base authentication on a predefined function *check* for digital signatures. Loaded data d will have to be signed with a signature s, such that calling *check(k,s,d)* with a key k stored on the card yields true. Since issues of cryptography are outside the scope of a formal model, we do not specify the types of s and k (one possible interpretation of k is a public key of RSA cryptography, and that s is a signature for d which can only be given using the corresponding private key). Instead we only make the following basic assumption: From a successful check it can be deduced that the person who stored k previously on the card has signed d, and therefore agreed to loading d.

Under the basic assumption, our authentication scheme will guarantee the following two properties for downloading applications:

- The card issuer can control which applications are loaded onto the card.
- The owner of an application has agreed to loading each of his programs. All other programs, which he has not agreed to being loaded, can not interfere with the application.

In particular, it is guaranteed that if the application owner does not want any communication with other applications, the application will be completely isolated. Also, the second property implies that any channel program between two applications A and B must have been authenticated by both A and B.

4 Noninterference

This section first repeats the main definitions of the generic noninterference model as defined by Rushby [16]. Following Rushby, we will sketch that a simple Bell/LaPadula model, where the system state consists of a set of subjects with an access class as clearance and a set of objects with an access class as classification, is an instance of the model. To define our smart card security model as an instance of noninterference, we had to make small modifications to the generic model. They resulted in a generalization of Rushby's main theorem, which is given at the end of the section.

The system model of noninterference is based on the concept of a state machine, which starts in a fixed initial state *init* and sequentially executes commands (here: OS commands, i.e. system calls). Execution of a command may alter the system state and produces some output. The model does not make any assumptions on the structure of the system or on the set of available commands. The model is specified algebraically using functions *exec*, *out* and *execl*; for a system state *sys* and a command *co*, *exec*(*sys*, *co*) is the new system state and *out*(*sys*, *co*) is the generated output. *execl*(*sys*, *cl*) (recursively defined using *exec*) returns the final state of executing a list *cl* of commands.

To define security it is assumed that each command *co* is executed by a subject with a certain *clearance*[1] D which is computable as $D = dom(co)$. The general model of noninterference makes no assumptions about the structure of clearances. They are just an abstract notion for the rights of a subject executing a command. Also note, that subjects are not defined explicitly in the generic model, since only their clearance matters for security.

A security policy is defined to be an arbitrary relation $\rightsquigarrow$ on clearances. $A \rightsquigarrow B$ intuitively means that a subject with clearance A is allowed to pass information to a subject with clearance B ("A interferes with B"), whereas $A \not\rightsquigarrow B$ means that commands executed by A will have no effect on B.

For the Bell/LaPadula instance of the model, the clearance of a subject is defined as usual as an access class, and the $\rightsquigarrow$-relation coincides with the less-or-equal relation on access classes (a subject with lower clearance can pass information to one with higher clearance, but not vice versa). The $\rightsquigarrow$-relation is therefore *transitive* in this case. The big advantage of a noninterference model over a Bell/LaPadula model is that it is possible to define interference relations, which are *not* transitive[2]. This is what we need for the smart card security model, to model communication: we want an application A to be able to pass information to another application B via a channel program C, i.e. we want $A \rightsquigarrow C$ and $C \rightsquigarrow B$. But we do not want information to be passed from A directly to B, i.e. we want $A \not\rightsquigarrow B$.

Informally, security of a noninterference model is defined as the requirement that the outcome of executing a command *co* does not depend on commands that were previously executed by subjects which may not interfere with the subject of *co*, i.e. $dom(co)$.

To formalize this, a function *purge* is defined. *purge(cl,B)* removes all commands "irrelevant for B" from the commandlist *cl*. The output to a command *co* then must be the same, whether *cl* or *purge(cl,dom(co))* are executed before it. Formally, a system is defined to be secure, if and only if for all commandlists *cl* and all commands *co*

$$out(execl(init,cl),co) = out(execl(init,purge(cl,dom(co))),co) \quad (1)$$

holds. For a transitive interference relation the definition of *purge* is simple: a command *co* can be purged if and only if $dom(co) \not\rightsquigarrow B$. For the simple Bell/LaPadula instance, Rushby[16] shows that this definition of security is equivalent to simple security and the $\star$-property. Therefore the simple Bell/LaPadula model is an instance of transitive noninterference.

The definition of security for an intransitive noninterference model (i.e. a noninterference model with an intransitive interference relation) also requires to prove property (1), but the definition of commands, which must be purged

[1] The clearance of a subject is called *security domain* in [16]. We avoid this term since it is also used with a different meaning in the context of Java security.

[2] an intransitive interference relation is also possible in domain and type enforcement models [4], [1], but these models do not have a uniform, provable definition of security, which rules out covert channels.

is more complicated: Consider the case mentioned above, where we have two applications A, B and a channel program C with $A \rightsquigarrow C$ and $C \rightsquigarrow B$, but $A \not\rightsquigarrow B$. Now according to the original definition of *purge*, first executing three commands $[co_1, co_2, co_3]$ with $dom(co_1) = A$, $dom(co_2) = C$ and $dom(co_3) = A$, and then looking at the output for a fourth command co executed by B should give the same result as looking at the output to co after only executing co_2: *purge* will remove both co_1 and co_3 since their clearance (in both cases A) does not interfere with B. But removing co_1 is wrong, since command co_1 could make some information of A available for C (since $A \rightsquigarrow C$), and the subsequent command co_2 could pass just this information to B (since $C \rightsquigarrow B$). Finally co could just read this information and present it as output.

Therefore co_1 may affect the output of co and should not be purged. In contrast, co_3 should be purged, since no subsequent commands can pass information to B (the domain of co). The definition of *purge* must be modified, such that its result is $[co_1, co_2]$. The question whether a command is allowed to have a visible effect on some subject after some more commands have been executed now becomes dependent on these subsequently executed commands. Therefore a set of clearances *sources(cl,B)*, which may pass information to B during the execution of a list of commands cl is defined. The first command, co, of a commandlist $[co|cl]$ then does not interfere with clearance B directly or indirectly (and may therefore be purged) if and only if it is not in $sources(cl, B)$.

We will give extended versions of *sources* and *purge* for our variant of the model below, which has Rushby's definitions as special cases. Defining a variant of the noninterference model was necessary to make our smart card security model an instance. Two modifications were necessary.

The first is a technical one: the system states we will consider in the smart card security model will have invariant properties, that will hold for all system states reachable from the initial state. Therefore, instead of showing proof obligations for *all* system states, it is sufficient to show them for system states with the invariant property only.

The second modification is more substantial: We do not assume that the clearance of a subject executing a command can be computed from the command alone, since usually the clearance of a subject is stored in the system state. Therefore we must assume that function *dom(sys,co)* may also depend on the system state. Making the *dom*-function dependent on the system state requires that *sources* and *purge* must also depend on the system state. Our definitions are:

$$sources(sys, [\,], B) = \{B\}$$

$$sources(sys, [co|cl], B) = \begin{cases} \{dom(sys, co)\} \cup sources(exec(sys, co), cl, B) \\ \quad \text{if } dom(sys, co) \rightsquigarrow A \\ \quad \text{for any } A \in sources(exec(sys, co), cl, B) \\ sources(exec(sys, co), cl, B) \quad \text{otherwise} \end{cases}$$

and

$$purge(sys, [\,], B) = [\,]$$

$$purge(sys, [co|cl], B) = \begin{cases} purge(sys, cl, B) \\ \quad \text{if } dom(sys, co) \notin sources(sys, [co|cl], B) \\ [co|purge(exec(sys, co), cl, B)] \quad \text{otherwise} \end{cases}$$

Security is now defined as:

$$\begin{array}{l} out(execl(init,cl),co) = \\ out(execl(init,purge(cl,dom(execl(init,cl),co))),co) \end{array} \tag{2}$$

Rushby's definitions are the special case, where none of the functions *dom*, *sources* and *purge* depends on the system state. It is easy to see that for transitive interference relations the simple definition of *purge* coincides with the definition given above.

For our definition, we proved the following generalization of Rushby's "Unwinding theorem" (Theorem 7 on p. 28 in [16]).

Theorem 1. If a relation $\overset{A}{\sim}$ and a predicate *inv* can be defined, such that the conditions

1. $\overset{A}{\sim}$ is an equivalence relation
2. $inv(sys) \wedge inv(sys') \wedge sys \overset{dom(sys,co)}{\sim} sys' \rightarrow out(sys,co) = out(sys',co)$
   (system is output consistent)
3. $inv(sys) \wedge dom(sys,co) \not\rightsquigarrow A \rightarrow sys \overset{A}{\sim} exec(sys,co)$
   (system locally respects $\rightsquigarrow$)
4. $inv(sys) \wedge inv(sys') \wedge sys \overset{A}{\sim} sys' \wedge sys \overset{dom(sys,co)}{\sim} sys'$
   $\rightarrow exec(sys,co) \overset{A}{\sim} exec(sys',co)$
   (system is weakly step consistent)
5. $sys \overset{A}{\sim} sys' \rightarrow (dom(sys,co) \rightsquigarrow A \leftrightarrow dom(sys,co) \rightsquigarrow A)$
   (commands respect $\rightsquigarrow$)
6. $sys \overset{dom(sys,co)}{\sim} sys' \rightarrow dom(sys,co) = dom(sys',co)$
   (commands respect equivalence $\sim$)
7. $inv(init)$ (initially invariant)
8. $inv(sys) \rightarrow inv(exec(sys,co))$ (invariance step)

are all provable, then the system is secure, i.e. property (2) holds.

The theorem allows to reduce the proof of property (2), which talks globally about all possible commandlists, to eight local properties for every command. It uses an equivalence relation $sys \overset{A}{\sim} sys'$ on system states, which intuitively says, that two system states *sys* and *sys'* "look the same" for a subject with clearance A. In the simple Bell/LaPadula instance of the model this is true if the files readable by A are identical.

5 The Formal Model

This section describes the formal security model in detail. First, we informally describe the data structures that form the system model. Then, we will describe the set of commands (OS calls) and their security conditions. Finally, we will give the formal properties we proved for the system model.

The main data structure used in the model is the system state. It consists of three components: a *card key*, the *authentication store* and the *file system*.

The card key is not modifiable. It represents authentication information that is necessary for any application to be downloaded onto the card. The card key could be the public key of the card issuer, but it could also contain additional information, e.g. the public keys of some certifying bodies, that are allowed to certify the integrity level of subjects (this is another idea used in the IBM system [9]), or it could contain the key of the card user. We assume that the card key is fixed, before the operation system is started (either already when the card is manufactured, or when the card is personalized).

The second component is the authentication store. It stores authentication information for every access category, for which there are files on the card. Usually we will have one authentication information per application, but it is also possible to allocate several access categories for one application (presumed the card issuer agrees).

The third, main component is the file system. An important decision we have taken in modeling the file system is to abstract from the structure of directories. Instead we have modeled only the classification of directories. This has the disadvantage that we must assume directories to exist when needed. On the other hand this makes the model more generic, since we do not need to fix a concrete directory structure like a tree or an (acyclic) graph. Note that adding a directory structure would only require to verify that creating and removing directories does not cause covert channels. All other commands and their security conditions (e.g. the compatibility property, see the next section) would remain unchanged.

The file system uniquely addresses files with file identifiers (which could be either file names together with an access path, or physical addresses in memory). Files contain the following five parts of information:

- The classification (secrecy and integrity access class) of the directory, where the file is located.
- The classification (secrecy and integrity access class) of the file itself.
- The file content, which is not specified in detail (usually a sequence of bytes or words).
- An optional security marking (i.e. a classification, consisting of four access classes). Files with a security marking act as subjects and objects (in the sense of Bell/LaPadula). Data files do not carry a security marking. They only have the role of objects.

Access classes consist of an access level (a natural number) and a set of access categories (i.e. unique application names), as usual in Bell/LaPadula-like models.

Access classes are partially ordered, using the conjunction of the less-or-equal ordering on levels, and the subset-ordering on sets of categories. The lowest access class *system-low* consists of level 0 and an empty category set. To have a lattice of access classes we add a special access class *system-high*, which is only used as the integrity level of the top-level directory.

The system starts in an initial state with an empty authentication store and an empty file system. Note that there is no "security officer" (or a "root" using UNIX terminology) who sets up the initial state or maintains the security policy. Such a supervisor is assumed in many security models, but in contrast to a stationary computer there is no one who could fill this role after the smart card has been given to a user.

The system now executes OS commands. The commands are grouped in two classes: *createappl*, *loadappl* and *delappl* are invoked by the OS itself as an answer to external requests, while *read*, *write*, *create*, *remove* and *setintsec* are called by a currently running (application or channel) program.

Our model can be viewed as a simple instance of a domain and type enforcement (DTE) model (see [4], [1]) with two domains "OS" and "application", where the domain interaction table (DIT) is set such that only the OS domain may create or delete subjects and the domain definition table (DDT) for the domain "application" is set according to the interference relation (the domain "OS" can not access files).

The command *createappl* creates a new access category, which acts as the name of a new application. *loadappl* loads the main file of an application (or a channel). The file gets a classification as security marking, and therefore can act as a subject in the model. *delappl* removes such a file. To access files, we use the commands *create* to create a new one, *read* to read its content, *write* to overwrite it, and *remove* to delete the file. Usual operating systems will have more low-level commands (like opening and closing files, or commands to read only the next byte of a file) to support an efficient memory management, but since the security conditions for opening a file would be exactly the same as for our *read* command, we have chosen the more abstract version. Finally, the command *setintsec* modifies the integrity and secrecy classification of a file.

The commands *read*, *write*, *create*, *remove* and *setintsec* are called by a currently running application or channel program (the current subject). To model this current subject, their first argument is a file identifier which points to the currently running program (a file with a security marking). We call a file identifier, which points to a program, a *program identifier*, and denote it as *pid*. The security marking of the file determines the clearance of the current subject.

It is not necessary to model the currently running program as an extra component of the system state, since files with a security marking are stored in directories with secrecy *system-low* and integrity *system-high* (we do not consider "secret subjects"). Therefore, switching between applications has no security conditions, and the additional argument *pid* which is given to each command can be freely chosen.

We will now give a detailed listing of the operating system commands available. For each command we first define its functionality (new system state and output), if all security conditions are fulfilled. Otherwise, all commands return *no* as output and leave the system state unchanged. Second, for each command a precise definition of the security conditions is given. To make the security conditions easily readable, we will use predicates *read-access*(*pid*, *fid*, *sys*), *write-access*(*pid*, *fid*, *sys*), *dir-read-access*(*pid*, *fid*, *sys*) and *dir-write-access*(*pid*, *fid*, *sys*). These describe in which circumstances a subject *pid* is allowed to see, read or write a file *fid* given a system state *sys*. For the predicates to hold, it is required that

- *pid* points to a file in the file system of *sys*, which has a security marking consisting of the four access classes (*ircl*,*iwcl*,*srcl*,*swcl*) for integrity/secrecy read/write. Remember that these markings characterize the clearance of the subject executing the command.
- *fid* points to a file in the file system of *sys* which has access classes *icl* and *scl* for integrity/secrecy, and whose directory has classification *idcl* and *sdcl*.
- For *read-access fid* must be readable by *pid*, i.e. $ircl \leq icl$ and $scl \leq srcl$.
- For *write-access fid* must be writable by *pid*, i.e. $icl \leq iwcl$ and $swcl \leq scl$.
- For *dir-read-access* the directory of *fid* must be readable by *pid*, i.e. $ircl \leq idcl$ and $sdcl \leq srcl$.
- For *dir-write-access* the directory of *fid* must be writable by *pid*, i.e. $idcl \leq iwcl$ and $swcl \leq sdcl$.

Note that *dir-read-access* determines whether a file *fid* is visible to the currently running application *pid* (i.e. whether its existence is known), while *read-access* gives access to the contents of a file.

create(pid,iac,sac). Subject *pid* creates a new file with empty content and no security marking in a directory with classification *iac* and *sac* for integrity and secrecy. The classifications of the new file are set to the read classifications of *pid*. The new file name is returned as output.

Security conditions: pid must point to a file with marking $(ircl, iwcl, srcl, swcl)$ and a directory that has classification (*iac*,*sac*) must be readable and writable by *pid*, i.e. $ircl \leq iac$, $sac \leq srcl$, $iac \leq iwcl$ and $swcl \leq sac$ must hold.

remove(pid,fid). Program *pid* deletes the file named by *fid* from the file system. The resulting output is *yes* on success, *no* on failure.

Security conditions: dir-read-access(*pid*, *fid*, *sys*) and *dir-write-access*(*pid*, *fid*, *sys*) must hold. Note that *dir-write-access* implies, that *fid* has no secrecy marking, since such files are stored in a directory with integrity = *system-high*.

setintsec(pid,fid,iac,sac). Program *pid* sets the classification of file *fid* to be *iac* for integrity and *sac* for secrecy. The command returns *yes* as output.

Security conditions:

1. *dir-read-access*(*pid*, *fid*, *sys*).

2. *dir-write-access*(*pid*, *fid*, *sys*).
3. *write-access*(*pid*, *fid*, *sys*).
4. Either one of the following two conditions holds:
   - The new integrity access class *iac* is not higher than the old integrity access class of the file, and the new secrecy class *sac* is not lower than the old secrecy class of *fid* (downgrading integrity and upgrading secrecy is allowed).
   - *fid* is readable by *pid*, i.e. *read-access*(*pid*, *fid*, *sys*) holds, the new integrity class is not higher than the integrity class of its directory and the new secrecy class is not lower than the secrecy class of its directory (upgrading integrity and downgrading secrecy is allowed for for *readable* files, as long as compatibility is not violated. Note that *dir-write-access* together with compatibility assures, that *pid*'s new integrity/secrecy will not be higher/lower than the write integrity/secrecy).

write(pid,fid,c). Program *pid* overwrites the file content of file *fid* to be *c*. The command returns *yes* as output.

Security conditions: fid must point to a file with no security marking. The conditions *dir-read-access*(*pid*, *fid*, *sys*) and *write-access*(*pid*, *fid*, *sys*) must hold.

read(pid,fid). Program *pid* reads the contents of file *fid*, which are returned as output. The system state is unchanged.

Security conditions: dir-read-access(*pid*, *fid*, *sys*) and *read-access*(*pid*, *fid*, *sys*) is required.

createappl(au, au'). A new application name (an access category) *ap* (relative to the ones that exist in the authentication store) with associated authentication information *au* is created, stored in the authentication store. *ap* is returned as output.

Security conditions: It is checked, whether the card issuer allows a new application with authentication information *au*. This is done with *check(ck,au',au)* using the additionally given key *au'* (a digital signature for *au* given by the card issuer) and the key *ck* of the card issuer that is stored on the card.

loadappl(au,st,d,c,iac,sac). A new program with clearance *d* and content *c* is loaded (added to the file system). Its security classes become *iac* and *sac*. The integrity/secrecy classification of the files directory is set to *system-high* and *system-low*. The new file identifier is returned.

Security conditions:

First the authorization of the card issuer for downloading the application is checked using the digital signature *au* by calling *check*(*ck*, *au*, (*d*, *c*, *iac*, *sac*)) (note that the full information (*d*,*c*,*iac*,*sac*) to be stored on the card must be signed). Then a check is done for every access category *an* (= application name) that is contained in any of the four access classes of the clearance *d*. For each such name *st* must contain an appropriate digital signature *au'* and calling *check*(*au"*, *au'*, (*d*, *c*, *iac*, *sac*)) must yield true, where *au"* is the key for *an* stored in the authentication store of the card. These checks make sure that any application which may be interfered, agrees to the downloading of the application.

delappl(au,st,fid). The file, to which *fid* points, is deleted. The command returns *yes* as output.

Security conditions: fid must have a security marking *d*. Otherwise, the security conditions are the same as for the loadappl command, except that the argument (d,c,iac,sac) for the *check* function is computed from the file *fid*.

6 How to Use the Model

In this section, we revisit the example from the introduction. We discuss how the commands of the previous section are used to establish the scenario of an airline A, which exchanges loyalty points with hotel chains H and I, and how the security policy is used.

We start with an empty smart card, that only stores a card key, which we assume to encode authentication information for the card issuer and the card holder. As a first action, we use *createappl* to store authentication information (a public key) for each of the three applications. Since we do not want the application to be structured internally, one call to *createappl* for each application is sufficient. Each returns one (new) access class for the application. The call to *createappl* is checked against the card key, so it is made sure that card issuer and card holder agree to creating the access classes, which we call A, H and I in the following.

To load an application program for each application, *loadappl* is called (there may be several programs for each application). The loaded application programs are checked to be signed by A, H and I respectively. Loading a new version of an application program can be done by calling *delappl* and then *loadappl* at any time.

After the application programs have been loaded, they can now be called freely (calling an application program is not security relevant, so the security model does not contain a special command). Each can freely create and modify files, using the *create*, *read*, *write* and *delete* commands.

The security policy ensures that the three applications will be completely separate, i.e. reading or writing files of another application, even by accident, is impossible.[3] No communication between the applications is possible, even if new applications are loaded, since we assume that no one else can guess the signature of A, H and I.

If H wants to transfer loyalty points to A, both A and H have to agree to load a channel program[4]. The channel program will have a read access class (both for integrity and secrecy) of H, and a write access class of A. Therefore to load it, both A and H must sign it. Of course this channel program should be checked carefully by both parties to do the following: When called, it should

[3] It is impossible, because the three secrecy access classes for A, H, and I are all disjoint. Bell/LaPadula also permits an access class to dominate another. For example, if access class X > Y, then information could flow from Y to X.

[4] Channel programs are called guard programs in [9]. They have also be called downgrading programs or sanitizers in various military applications.

read a file given by H to ensure that it has a suitable format, containing only information about loyalty points. This is possible, since the channel program has read access. Then it should call *setintsec* to change both the integrity and the secrecy access class of the file from H to A. Thereby the file is moved from the set of files accessible for H to those accessible by A, and application A can subsequently read the loyalty points.

The security policy ensures that the channel program can only transfer information from H to A. No other communication will be possible as long as no other pair of applications agrees to loading another channel program. The actions of the channel program will be completely invisible to the other hotel I.

Finally, to establish the scenario of the introduction, I and A will load another channel program, and bank B will create another application with *createappl* and *loadappl*. Maybe the bank as application provider will need two access classes, to run two applications (maybe one for the electronic wallet and one for online banking). Then it will call *createappl* twice. The bank might also use various secrecy and integrity levels for its files. Then the security policy will guarantee that the applications of the bank will respect Bell/LaPadula secrecy and Biba integrity internally.

Additional examples of using the model can be found in [10], where the model is shown in an electronic purse example and in several possible messaging scenarios.

7 Verification

The smart card security model described in Sect. 5 and the modified generic model of noninterference were formally specified using the VSE II system [5].

The noninterference model as described in Sect. 4 consists of about 200 lines of algebraic specification, and Theorem 1 was proved similarly to [16].

The smart card security model of Sect. 5 was also specified algebraically with 800 lines. A full specification of both models can be found in [11]. The following three main security claims were verified:

- The card issuer controls which applications are loaded onto the card, i.e. any application loaded on the card was signed by the card issuer.
- The owner of each application has signed each of his programs, when it was loaded. No program, which he has not agreed to loading, can interfere with the application.
- The smart card model is an instance of the noninterference model.

For the first proof we basically have to show, that each authentication information stored in the authentication store has been checked for agreement of the card issuer. This is done by induction on the number of executed commands, since the authentication store is modified only by adding new entries in *createappl*.

For the proof of the second property, note that an application, which first allocates an access category A as its application name with *createappl*, and then loads an application file with the set of access categories in all four access classes

set to {A} can be interfered only by other applications, which have A in their category set of integrity write clearance (provided the third property holds!). Therefore it is sufficient to check, that any file with a security marking that contains an access category A, has been checked to be signed by A when it was loaded. This can again be proved by induction on the number of executed commands, similar to the first property.

The proof that shows that the smart card model is an instance of noninterference is much more complicated. The main problem here is to find definitions of the noninterference relation $\rightsquigarrow$, the equivalence relation $\overset{A}{\sim}$ and the system invariant *inv* such that Theorem 1 holds. Sect. 3 requires that we define $A \rightsquigarrow B$ as $\text{iwcl}_A \geq \text{ircl}_B$ and $\text{swcl}_A \leq \text{srcl}_B$, but the other two definitions have to be found incrementally by proof attempts. We tried several versions, which lead to unprovable goals. Analyzing such an unprovable goal always resulted in a concrete system state, which our definitions classified as secure and a sequence of commands that lead to an insecure state. We then had to decide, whether the security conditions of one of the involved commands was wrong, or whether our definitions of $\overset{A}{\sim}$ and *inv* were still insufficient.

For the final system invariant we use the conjunction of the two properties:

1. Compatibility property: Each file has an integrity (secrecy) classification that is at most (at least) the integrity (secrecy) classification of its directory
2. Visibility property: All files with a security marking are stored in a directory with integrity/secrecy classification system-high/system-low

The first property is common in mandatory security models (e.g. the Multics instance of Bell/LaPadula uses it too). The second property is necessary since we want to be able to switch freely between applications (see previous section).

The final definition of the equivalence relation $sys \overset{A}{\sim} sys'$, which says, which system states "look the same for a subject with clearance A" consists of the following four properties:

1. The authentication store and the card key must be the same in both system states.
2. The loaded files, which have a security marking, must be the same.
3. The set of files visible for A must be the same, i.e. the set of filenames for which a file exists in a directory which A can read, must be the same. Their classifications and directory classifications must be the same.
4. The files which A can read in both system states must be identical. Not only must they have the same security marking (this follows already from the second property), the same classification and the same directory classification (because of compatibility this is implied by the third property), but also their content must be identical.

With these three definitions of $\rightsquigarrow$, *inv* and $\overset{A}{\sim}$ we were able to verify the eight preconditions of Theorem 1. Each of the proofs split into subproofs for each of the eight commands.

About a month of work was needed to reach a fully verified security policy, most of the time was spent to verify that the model is an instance of noninterference. During this time several specification errors were found. Many of them were typing errors, but a few of them were errors in the security conditions, which had not shown up during a careful informal analysis by several people. There were some minor errors, that were easy to correct, e.g. that the integrity classification of a newly created file has to be set to the read integrity of the caller, not its write integrity.

The most problematic security conditions we found are those of the *setintsec* command, which modifies the classification of a file. Originally there were separate commands to set integrity and secrecy, but this results in the following problem: If we want both to upgrade secrecy and to downgrade integrity, after executing the first command the files' secrecy will be too high to downgrade integrity. Weakening the conditions of *setintegrity* to allow integrity downgrading for files, whose secrecy has been upgraded resulted in covert channels. It was interesting to see, that these covert channels are not possible in a pure Bell/LaPadula and Biba setting, but that they are specific to using subjects (channels), which have different access classes for reading and writing.

8 The IBM Model

The IBM model [9] addresses two issues that are not in the current version of the formal model. The first on assignment of integrity levels is a purely practical issue that cannot really be formalized. The second on execution control is simply not formalized at this time.

The first issue is that the Biba integrity model does not model any real practical system. Unlike the Bell/LaPadula model that developed from existing military security systems, the Biba integrity model developed purely from a mathematical analysis of the security models. However, Biba did not suggest how to actually decide which programs were deserving of a high integrity access class and which were not. This has made practical application of the Biba model very difficult.

In section 3, we required that developers and card issuers digitally sign the applications, using the function *check*. This is much as is done in Java and ActiveX security approaches. However, the IBM informal model goes beyond this. If an application has been independently evaluated and digitally signed by a certifying body, then we can grant it a higher level of integrity, without having to depend on the reputation of the developer or the skills of the card issuer. For example, we could define integrity levels for ITSEC-evaluated [8] applications. The Commercially Licensed Evaluation Facility (CLEF) would evaluate the application and the certifying body would digitally sign the application and its ITSEC E-level. A card issuer (such as a bank) might lay a requirement on vendors who want to download applications onto their cards. The application must have received an ITSEC evaluation at a policy-determined level to be acceptable. Common criteria evaluations [7] would be equally acceptable.

There could be provisions for less formal evaluations than full ITSEC. For example, a commercial security laboratory could check an application for obvious security holes (buffer overflows and the like) and for Trojan horses or trapdoors. While not as formal as an ITSEC evaluation, it might be sufficient for loyalty applications.

There is one problem with using more than one kind of evaluation criteria. If an application has been evaluated under one criteria, and another application has been evaluated under a very different criteria, then if a user wishes to download both of those application onto the same card, it is not clear how to compare the integrity classes. If the two criteria have defined mappings (such as the E levels of the ITSEC and the EAL levels of the Common Criteria), then there is not a problem. However, if the card issuer chose to use some very different and incompatible criteria, then downloading of other applications that were ITSEC evaluated might be difficult.

The second issue is control of execution permissions. The original Biba model prevents high integrity applications from reading low-integrity data, in fear that the application might be compromised in some form. This makes it difficult to describe applications that have been designed with high integrity to specifically process low integrity data input and to rule on its appropriateness. This processing of low integrity data is called sanitization. However, in the process of allowing a high integrity application to sanitize low integrity data, we do NOT want to allow a high integrity application to execute low integrity code, either deliberately or accidentally[5].

As discussed in section 3, we support sanitization for both secrecy and integrity by assigning four access classes to each subject (*ircl*,*srcl*,*iwcl*,*swcl*), the first two for reading and the last two for writing. In the traditional Bell/LaPadula and Biba models, execution is always associated with reading, but that association would allow a high integrity subject that was sanitizing low integrity data to also execute low integrity program code. Therefore, for integrity only, the IBM model associates execute permission with write permission, rather than read permission. Separating the execute permission from the read permission originated in the program integrity model of Shirley and Schell [18] which was in turn based on the protection ring mechanism of Multics [14]. The policy was further developed in the GEMSOS security model [17] that specified a range of levels within which integrity downgrading could occur.

The combined access rules are shown in Fig. 2. Recall that subjects have four access classes, while objects have only two[6]. The execute permission rule specified in the figure is for a normal program to program transfer[7].

[5] Most buffer overflow attacks come from violating this rule.

[6] Subjects can sometimes be treated as objects. Details on this can be found in [9].

[7] The IBM operating system also supports another operation, called CHAIN, which is a way to start a separate process executing at some other integrity and secrecy access class. The intended use of CHAIN is to start a guard or sanitization process or for a guard process to start a recipient of sanitized information. Details of CHAIN are omitted here, for reasons of space, but can be found in [9].

Read permission
srcl (subject) $\geq$ *scl* (object) and *ircl* (subject) $\leq$ *icl* (object)

Write permission
swcl (subject) $\leq$ *scl* (object) and *iwcl* (subject) $\geq$ *icl* (object)

Execute permission
srcl (subject) $\geq$ *scl* (object) and *iwcl* (subject) $\leq$ *icl* (object)

The target program of a transfer runs at the integrity level of the caller. A high integrity program cannot call or transfer to lower integrity code.

Fig. 2. Access Control Rules

9 Conclusion

We have defined a generic security model for the operating system of a multiapplicative smart card. The model formalizes the main security aspects of secrecy, integrity, secure communication between applications and secure downloading of new applications. The two main theoretical results are that intransitive noninterference is a suitable framework for such models and that authentication can be integrated in the model.

We found that formal verification was extremely helpful in analyzing the security model. We were able to remove all covert channels from the model, even ones that we had not found during a thorough informal analysis. The six weeks required for formal specification and verification of the model are a reasonable effort to achieve this result.

There is still work to do. One important question we have left open is to formalize the communication of applications on the card with the outside world. This issue would require to extend the security model to include security aspects of the outside world, e.g. the authentication of card readers. We also would have liked to compare our security policy for downloading with the upcoming VISA standards (which were not available to us yet). Finally, extending the model to be applicable to Java Cards will also require further research.

References

1. L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. Practical domain and type enforcement for UNIX. In *1995 IEEE Symposium on Security and Privacy*, pages 66–77, Oakland, CA, May 1995. URL: http://www.tis.com/docs/research/secure/secure_dte_proj2.html.
2. D. E. Bell and L. J. LaPadula. Secure Computer Sytems: Unified Exposition and Multics Interpretation. Technical Report ESD–TR–75–306, The MITRE Corporation, HQ Electronic Systems Division, Hanscom AFB, MA, March 1976. URL: http://csrc.nist.gov/publications/history/bell76.pdf.
3. K. J. Biba. Integrity Considerations for Secure Computer Sytems. Technical Report ESD–TR–76–372, The MITRE Corporation, HQ Electronic Systems Division, Hanscom AFB, MA, April 1977.
4. W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *8th National Computer Security Conf.*, pages 18–27, Gaithersburg, MD, 1985. National Computer Security Center and National Bureau of Standards.

5. D. Hutter, H. Mantel, G. Rock, W. Stephan, A. Wolpers, M. Balser, W. Reif, G. Schellhorn, and K. Stenzel. Vse : Controlling the complexity in formal software developments. In *Applied Formal Methods — FM-Trends 98*. LNCS 1641, 1998.
6. Identification cards - identification cards - interrelated circuit(s) cards with contacts - part 4: Inter-industry commands for interchange. ISO/IEC 7816-4, International Standards Organization, 1995.
7. Information technology - security techniques – evaluation criteria for IT security. ISO/IEC 15408, International Standards Organization, 1999. URL: http://csrc.nist.gov/cc.
8. ITSEC. *Information Technology Security Evaluation Criteria, Version 1.2*. Office for Official Publications of the European Communities, Brussels, Belgium, 1991.
9. P. A. Karger, V. Austel, and D. Toll. A new mandatory security policy combining secrecy and integrity. RC 21717, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY, 15 March 2000. URL: http://domino.watson.ibm.com/library/CyberDig.nsf/home.
10. P. A. Karger, V. Austel, and D. Toll. Using a mandatory secrecy and integrity policy on smart cards and mobile devices. In *(EUROSMART) Security Conference*, pages 134–148, Marseille, France, 13-15 June 2000. RC 21736 available at http://domino.watson.ibm.com/library/CyberDig.nsf/home.
11. F. Koob, M. Ullmann, S. Wittmann, G. Schellhorn, W. Reif, A. Schairer, and W. Stephan. A generic security model for multiapplicative smart cards — final report of the SMaCOS project. to appear as BSI report.
12. T. F. Lunt, P. G. Neumann, D. Denning, R. R. Schell, M. Heckman, and W. R. Shockley. Secure distributed data views – vol.1: Security policy and policy interpretation for a class A1 multilevel secure. Technical Report SRI-CSL-88-8, SRI International, Menlo Park, CA, August 1988.
13. J. McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley & Sons, 1994. URL: http://chacs.nrl.navy.mil/publications/CHACS.
14. Elliott I. Organick. *The Multics System: An Examination of Its Structure*. The MIT Press, Cambridge, MA, 1972.
15. Philips semiconductors and IBM research to co-develop secure smart cards: Highly secure operating system and processor, suitable for multiple applications. URL: http://www.semiconductors.philips.com/news/content/file_384.html, Feb. 1999.
16. J. Rushby. Noninterference, Transitivity, and Channel-Control Security Policies. Technical Report CSL-92-02, SRI International, Menlo Park, CA, 1992. URL: http://www.csl.sri.com/~rushby/reports/csl-92-2.dvi.Z.
17. R. Schell, T. F. Tao, and M. Heckman. Desingning the GEMSOS security kernel for security and performance. In *8th National Computer Security Conference*, pages 108–119, Gaithersburg, MD, 30 September - 3 October 1985. DoD Computer Security Center and National Bureau of Standards.
18. L. J. Shirley and R. R. Schell. Mechanism sufficiency validation by assignment. In *1981 Symposium on Security and Privacy*, pages 26–32, Oakland, CA, 27–29 April 1981. IEEE Computer Society.
19. D. F. Sterne and G. S. Benson. The controlled application set paradigm for trusted systems. In *1995 National Information Systems Security Conference*, Baltimore, Maryland, 1995. National Computer Security Center and National Institute of Standards and Technology. URL: http://www.tis.com/docs/research/secure/secure_dte_proj2.html.