Department of
Information Engineering
and Computer Science DISI

UNIVERSITY OF TRENTO - Italy

DISI - Via Sommarive 14 - 38123 Povo - Trento (Italy)
http://www.disi.unitn.it

IMPLEMENTATION REQUIREMENTS AND SPECIFICATION OF THE POLICY CHECKER COMPONENT

Olga Gadyatskaya, Fabio Massacci and Anton Philippov

February 2011

Technical Report # DISI-11-455
Version 1.0

Technical report: Implementation Requirements and Specification of the Policy Checker Component *

Olga Gadyatskaya, Fabio Massacci and Anton Philippov

May 23, 2011

Abstract

Multi-application smart cards running on the Java Card technology can become an open environment for applications coming from different providers and collaborating in order to enable more sophisticated and user-oriented smart card solutions. While this is a dream of a lot of smart card vendors, in reality we do not see such cards around. One of the reasons for the absence of these cards is a necessity for a flexible security enforcement solution which will ensure that each provider's security policy for application interactions is satisfied on the card. We have proposed the Security-by-Contract approach for loading time application certification which enables such security mechanism on a smart card.

Security-by-Contract framework for Java smart cards consists of two components: the Claim Checker and the Policy Checker. Each application arrives on the smart card with its contract, which is a model of the application's security-related behavior. At the loading time the Claim Checker ensures that the contract is compliant with the application code, while the Policy Checker verifies that the contract matches the security policy of the platform.

This technical report proposes a representation of the contract and the security policy objects on Java Card. Related algorithms for the Policy Checker and the maintenance of these objects are also

*Work partially supported by the EU under grant EU-FP7-FET-IP-Secure Change

presented. The approach allows significant time and space savings, which are very important for such resource-constrained environment as a smart card.

1 Introduction

As a case study we consider an open multi-application smart card running on the Java Card technology [3]. These cards can host multiple applications from different vendors written in a subset of the Java language. Applications can be loaded on the card asynchronously and without common agreement among all the stakeholders. Ideas for this type of cards and possible scenarios of its usage and benefits have been discussed by the research community for quite some time now [12, 1, 11, 6].

An important aspect of these cards is collaborations between different applications. The added value of on-card application interactions is alluring. Loyalty cards, transportation cards are typical and widespread examples of an open multi-application platform which can bring benefits to the card users.

1.1 Why Internal Data Structures Are Needed?

The main objective of the current report is to define representation of the contract and the security policy objects, and to provide necessary updates of the PolicyChecker algorithms taking into account the representation defined. The definitions of application contract and security policy in [5, 4] did not detail how these objects could be provided on Java Card. Moreover, different properties are expected from the contract object delivered with application code and contract/policy objects implemented on the card. Indeed, application providers would like contracts to be easy to write and they probably would not want to study a new language for writing contracts. Also application providers do not know in advance which other applications can be in the system (or will arrive later). Thus contract provided on the card needs to be specified worldwide uniquely. That means, in terms of application identifiers (AIDs) and method tokens from export files (with corresponding AIDs). Tokens are discussed further in Section 2.3.

On the other hand, the contract and the policy objects on the card do not need to be worldwide unique. But it is important that they require as few memory as possible. One AID needs up to 16 bytes (by ISO standards). Thus we propose to explore bit vectors for representing applications and their services instead of using full AID and method tokens for the internal policy object. This internal object also speeds-up computations performed by the PolicyChecker on the card. The drawback of the suggested approach is extra time and memory required for maintenance of the mapping tables (that will

allow transformations from external contract objects to internal and back).

The remainder of the report is structured as follows. The Java Card technology is briefly outlined in Section 2. The Security-by-Contract framework is introduced in Section 3. Preliminary details about application contract and our running example are provided in Section 3.2. As the Contract is supplied with the application code we then proceed with a discussion in Section 4 how the actual contract can look like when it is received. Then Section 5 introduces our ideas for on-card objects and the mapping table which stores correspondence between received contract object and internal one. Section 6 presents the objects which can represent the contract and the security policy objects on the card. The modified on-card algorithms for the PolicyChecker are described in Section 7. We then discuss operations necessary for the policy update in Section 8. We analyze the complexity of given algorithms and discuss scaling of the model in Section 9.

2 The Java Card Technology

Java Card is a popular middleware for multi-application smart cards, which allows post-issuance installation and deletion of applications. Moreover, Java Card brings conveniences of the Java language to such small devices as smart cards. Application providers develop applets (Java Card applications) in a subset of Java language. This subset is object-oriented, but misses some traditional data types (for example, `float` or `long`), even `int` is supported optionally. Also such features as dynamic class loading or finalization are not supported by Java Cards. Full description of the Java Card language and the Java Card virtual machine (JCVM) is provided in the official specifications by Sun (now Oracle) [7, 8, 10].

Currently smart cards in the field run on Java Card version 2.2.2. Also a new specification for Java Card 3.0.3 is available. Third generation of Java Card supports Classic and Connected editions. Classic edition is, essentially, the same version as the second generation Java Cards. Connected edition supports new types of applications and new protocols for accessing them, but is not yet welcomed by the smart card vendors due to, among all, security concerns. Thus we concentrate on Java Card 2.2.2 version. However, the approach we propose in the future can be ported also for the third generation of Java Cards.

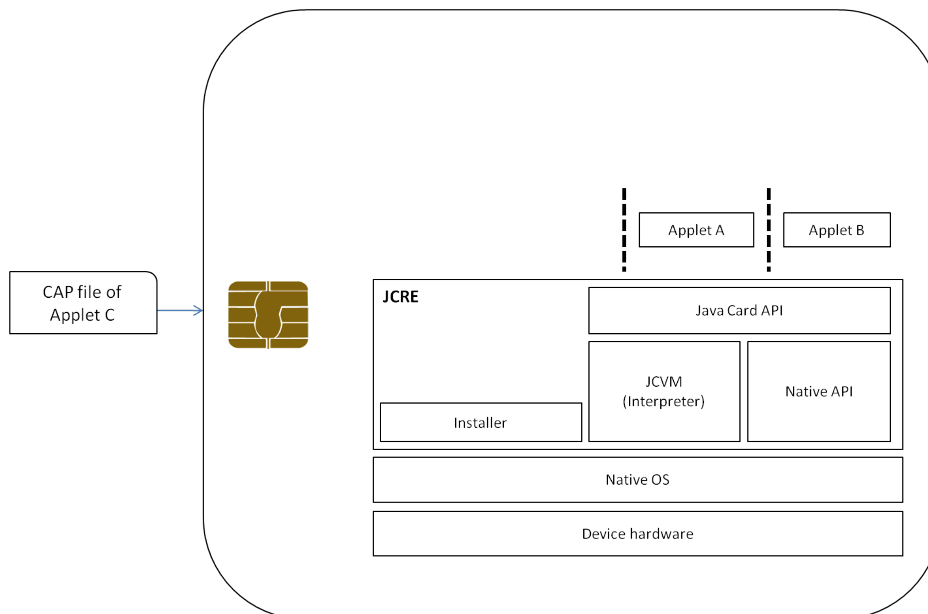

Figure 1: Java Card Architecture

2.1 Java Card Platform Architecture

The architecture of the Java Card platform is depicted on the Fig. 1. The architecture comprises several layers which include device hardware, an embedded operating system (native OS), the Java Card run-time environment (JCRE) and the applications installed on top of it [3]. Important parts of JCRE are Java Card virtual machine (JCVM) (its Interpreter part) and an Installer, which is an entity responsible for loading and installation of applications. The Installer is an optional component, but without it post-issuance application loading is impossible. Applications on Java Card are separated by a firewall.

Applets are supplied on the card in packages. The source code of a package is converted by the application providers into class files and then (using a Converter which is actually an off-card part of the JCVM) into a CAP file. The CAP file is transmitted onto a smart card, where it is processed, linked and transformed into a platform-specific executable format (defined by the platform developer). Application providers do not need to consider different on-card executable formats, as they are just required to

supply a correct (compliant with the Java Card specifications) CAP file. Then, upon finalization of linking process, the applet instances are installed.

A CAP file consists of several components, which typically arrive on a card in a known order [10, Chap. 6]. The information in each of the component is organized in a way which allows any implementation of the Installer and the JCVM to process the received CAP file.

2.2 Interactions between Applets

The Java Card Virtual Machine is a subset of the standard Java Virtual Machine (JVM) [10]. It also imposes some restrictions on the method invocations. Applications from one package belong to the same context. If two applets belong to different packages, their contexts are different. The Java Card firewall confines applet's actions to its designated context [8]. Thus, normally, an applet can reach only objects belonging to its own context. The only applet's objects accessible through the firewall are methods of specific shareable interfaces, also called *services*. A shareable interface is either the `javacard.framework.Shareable` interface or an interface that extends the `javacard.framework.Shareable`.

If an application A implements some services, it is called a *server*. An application B that tries to call any of these services is called a *client*. A typical scenario of a service usage starts with a client's request to the JCRE for a reference to A's object (that is implementing shareable interface). The firewall passes this request to application A, which decides if the reference can be granted or not. If the decision is positive, the reference is passed through the firewall and is stored by the client for further usage. The client can now invoke any method declared in the shareable interface which is implemented by the referenced object. During invocation of a service a context switch will occur, thus allowing invocation of a method of the application A from a method of the application B. A call to any other method, not belonging to the shareable interface, will be stopped by the Java Card firewall [3, 8].

In order to realize this scenario the client has necessarily to import the shareable interface of the server. As well the client needs to obtain specific export file of the server, which lists shared interfaces and services and contains their tokens. The server's export file is necessary for conversion of the client's package into a CAP file. In a CAP file all methods are referred to by their tokens, thus during conversion from class files into a CAP file the client needs to know correct tokens for services it invokes from other applications.

As shareable interfaces and export files do not contain any implementation, it is safe to distribute them.

As all applet interactions inside one package are not controlled by the firewall and due to the fact that a package is loaded in one pass (thus it is not possible to load a malicious applet in one package with an honest one), we consider that one package contains only one applet. Further we will say that some applet is loaded on the card meaning that the package containing this applet is loaded.

2.3 Tokens and Token-based Linking

Tokens are used by the JCRE for linking on the card in the same fashion as Unicode strings are used for linking in standard Java class files. For externally visible elements, such as shareable interfaces and their methods, tokens are declared in the export file of the package. If applet A wants to provide some services, it has to make its export file available for all potential clients. Applet B in its source code refers to services by their Unicode string names, but when it is converted into CAP file these names are replaced with tokens from A's export file [10]. Thus it is possible to identify provided and called services in terms of tokens correctly and uniquely.

A service s is generally defined as a tuple $\langle A, I, t\rangle$, where A is unique application identifier (AID) of an application that provides the service s, I is a token for a shareable interface where the service is defined and t is a token for the service s in the interface I. Further we will sometimes omit an AID A from the discussion and will refer to a service as a tuple $\langle I, t\rangle$.

3 Security-by-Contract for Java Cards

The Security-by-Contract framework for smart cards provides an extension of the Java Card architecture with two components: the ClaimChecker and the PolicyChecker. The loading time verification process is performed by these components, which separate the duties. The proposed architecture is depicted on the Fig. 2, the additions to the JCRE are in long dashed blue rectangles.

In the Security-by-Contract (S×C) approach every applet arrives on the card equipped with its contract. The contract contains information regarding provided and called services and an applet's policy. On the card the

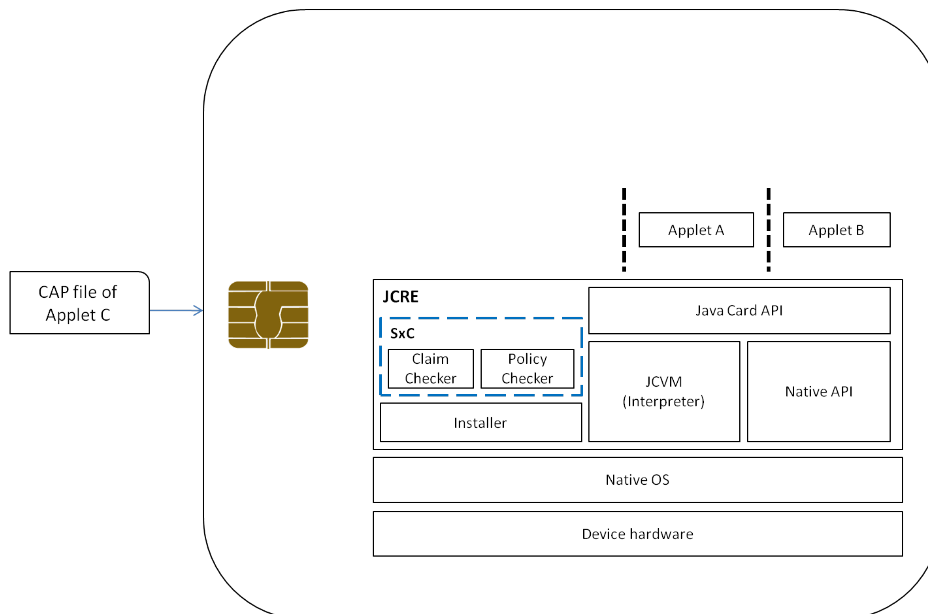

Figure 2: Security-by-Contract Extended Architecture

ClaimChecker verifies during installation process that the contract is compliant with the applet's bytecode. Then the PolicyChecker ensures that the contract is compliant with the card's security policy, which is provided jointly by all the stakeholders by combining their own policies related to the service usage. If one of the components finds inconsistences, the applet's installation process is stopped and it is rejected from the card. Consequently, only honest applications with compatible contracts can be installed on the card.

3.1 The S×C Framework Description

We will denote contract object delivered on the card as ContractExt and internal contract and policy objects as ContractInt and Policy correspondingly.

The following workflow will be performed by the Security-by-Contract (S×C) framework in case of loading of a new application:

1. A package A is sent to the card together with its contract $\mathsf{ContractExt}_{\mathsf{A}}$.
2. The ClaimChecker ensures that the $\mathsf{ContractExt}_{\mathsf{A}}$ is compliant with the actual code (specification is provided in [4]).

3. The PolicyChecker transforms the object $\mathsf{ContractExt_A}$ into internal object $\mathsf{ContractInt_A}$ using techniques we specify in this work.

4. The PolicyChecker runs verification algorithms to make sure that the contract ContractInt is compliant with the platform policy Policy.

5. If the PolicyChecker returned TRUE, the application A is installed on the platform and the security policy is updated correspondingly (to include $\mathsf{ContractInt_A}$.)

6. If the PolicyChecker returned FALSE, the installation process is stopped. The card returns to the previous state (it is possible due to Java Card transaction possibilities [9]).

In this technical report we present the main ideas behind the Security-by-Contract (S×C) framework for smart cards focusing mainly on the Policy Checker component. The detailed specifications of the formal models and the components can be found in the deliverables of the Secure Change project [5], [4].

3.2 Contracts

The main goal of the S×C framework is to verify at loading time absence of non-authorized service invocations. Service for us is a method of an interface which extends `Shareable`. Applications from different packages can access services of each other, but cannot access other methods. In the proposed solution each application declares in its Contract for each service which applications are authorized to call it. During installation of an applet B it can turn out that B may call a service $A.s$ of some applet A (which is present on the card). In this case, if B is not authorized by A to call s, then B's loading process is stopped and B is rejected.

We use the following notations in the sequel.

- $\mathcal{A}$ is a set of applications installed on the smart card platform, $A \in \mathcal{A}$ is an application;

- $\mathcal{S}$ is a set of services provided together by all the applications $\mathcal{A}$ on the card, $s \in \mathcal{S}$ is a service, $A.s$ is a service of application A.

- wp() is the Weierstrass symbol.

The $\mathsf{Contract_A}$ of an application A in our approach is defined as following:

Definition 3.1 (Contract) *For application A its $\mathsf{Contract_A}$ is a tuple $\langle \mathsf{Claim_A}, \mathsf{AppPolicy_A} \rangle$, where*

- $\mathsf{Claim_A} = \langle \mathsf{Provides_A}, \mathsf{Calls_A} \rangle$, *and* $\mathsf{Provides_A}, \mathsf{Calls_A} \in \mathrm{wp}(\mathcal{S})$;
- $\mathsf{AppPolicy_A} = \langle \mathsf{secrules_A}, \mathsf{funcrules_A} \rangle$, *and* $\mathsf{secrules_A} : \mathcal{S} \rightarrow \mathrm{wp}(\mathcal{A})$, $\mathsf{funcrules_A} \in \mathrm{wp}(\mathcal{S})$.

We assume $dom(\mathsf{secrules_A}) \subseteq \mathsf{Provides_A}$ *and* $\mathsf{funcrules_A} \subseteq \mathsf{Calls_A}$.

Thus, $\mathsf{Provides_A}$ is a declared set of the services that applet A supplies. $\mathsf{Calls_A}$ is a declared set of services that applet A can invoke. It is a task of the ClaimChecker to ensure that $\mathsf{Provides_A}$ and $\mathsf{Calls_A}$ are declared truthfully.

$\mathsf{secrules_A}$ is a set of authorizations granted by A. If a pair (s, B) is in $\mathsf{secrules_A}$, then B can call s without limitations. $\mathsf{funcrules_A}$ declares functionally necessary services.

The *security policy* in our approach is a collection of the contracts of all applets installed on the card.

Definition 3.2 (Platform Security Policy) *Security policy of the platform $\mathcal{P}$ consists of the contracts of all the applications $\mathcal{A}$ on the platform:* $\mathcal{P}=\{\mathsf{Contract_A}\}_{\forall \mathsf{A} \in \mathcal{A}}$

Thus, when an update happens on the platform (an application is installed, updated or removed) the security policy $\mathcal{P}$ needs to be updated correspondingly.

3.2.1 Received and Internal Objects

The definition 3.1 describes a formal model of an application contract. In the current report we will mainly discuss representation ContractInt of this model explored on the card and transformation from received contract object ContractExt into internal object ContractInt.

We denote fields of the contracts objects as Provides, Calls, secrules and funcrules, as these notations are compatible with the Contract model. Thus we will refer to these fields as, for example, ContractInt.Provides or ContractExt.Secrules. When the object (ContractExt or ContractInt) will be obvious from the context, we will use simpler notations Provides or secrules.

The security policy object on the card will be denoted as Policy. As we will explain further it also has fields Provides, Calls, secrules and funcrules. We will refer to these fields as Policy.Provides or Policy.Secrules. If the context will be self-evident, we may as well refer to them just as Provides or secrules.

We expect that $\mathsf{ContractExt_A}$ of an application A will be received by the card during A's loading process. Thus $\mathsf{ContractExt_A}$ is a Java object packed together with the bytecode of A (CAP file).

For the purposes of this report it is sufficient to discuss contracts delivered on the card as classes of the applets. We reference the contract fields as, for example, AppletName.ContractExt.Calls. Concrete implementation though will require some more elaboration on the contracts.

It is known that all the services are listed in the export file of A. They are represented there as Unicode strings in the UTF-8 format and also their tokens are present. It is expected that if some other application B will have a call to some of these services, then B needs to know the services' names and also to import A's export file. Application identifier (AID) is also listed in the export file. In the CAP file only the tokens for services are present. Thus, as the ClaimChecker has to verify compliance of the application bytecode with its contract, we have chosen to use tokens for service naming in the ContractExt. We recall, that each service can be uniquely identified by a tuple $\langle AID, I, t \rangle$, where AID is the package AID and I and t are the Export file tokens for the interface and the method.

Example 1 *Our running example is a multi-application smart card described in [2]. We assume two stakeholders on the platform:* Bank *and* Transport*. Two applications belong to* Bank*, these are* EMV *and* ePurse*. Application* jTicket *belongs to* Transport*.*

We assign the following identifiers to the packages and applications on the card:

- EMV *package AID:* [0x01, 0x01, 0x01, 0x01, 0x01]*,*
- ePurse *package AID:* [0x01, 0x01, 0x01, 0x01, 0x01, 0x02]*,*
- Transport *package AID:* [0x02, 0x02, 0x02, 0x02, 0x02]*,*

Application EMV *has one service* transaction*,* ePurse *has one service* payment *and* jTicket *has no services. Application* jTicket *provides to the card holder*

certain number of tickets for public transportation. Tickets can be bought using ePurse*, and the act of ticket purchasing requires the invocation of service* ePurse.payment *by* jTicket*.*

The Bank *owner allows data exchanges between applets* EMV *and* ePurse*, and between* ePurse *and* jTicket*, but not between* EMV *and* jTicket*. On the other hand,* Transport *owner not only allows data exchange between* ePurse *and* jTicket*, but she actually needs this exchange, otherwise her application is useless.*

4 ContractExt Object Received with an Applet

Further we provide details of ContractExt Java object (for each ContractExt field), which for now we describe as a subclass of the applet object. We assume that all the tables are of dynamic (not fixed) size.

4.1 ContractExt Fields in the Applet Code

ContractExt.Provides Field.

We represent the Provides set as a byte array. Each cell contains a token that corresponds to the provided service. We note that for the $\mathsf{Provides}_\mathsf{A}$ set AID of the application A is listed in the Header component of the CAP file and needs to be extracted from there for further usage.

Example 2 *Application* ePurse *provides only one service* payment *and it will have token* $\langle 0, 0\rangle$ *in the export file (tokens of the interface and the method). Therefore,* ePurse.ContractExt.Provides $= [0x00, 0x00]$.

Application jTicket *does not provide any services and its* jTicket.ContractExt.Provides *object will be empty.*

ContractExt.Calls Field.

The set Calls needs to contain all the called services. For each of the called services we have the server's package AID and the service tokens. Thus the identifier of the method composed in this way is unique. We can define method identifier as a 18 bytes array (filling empty bytes with zeros). Below is a table that represents the Calls set.

Table 1: ContractExt.Calls

Callee Package AID: byte array	Callee Method ID:tokens

Example 3 *Application* jTicket *calls service* payment *in* ePurse *application.* ePurse *has package AID* [0x01, 0x01, 0x01, 0x01, 0x01, 0x02] *and* payment *has token* ⟨0x00, 0x00⟩. *Therefore,* jTicket.ContractExt.Calls *will look as follows:*

Callee Package AID: byte array	*Callee Method ID:tokens*
[0x01, 0x01, 0x01, 0x01, 0x01, 0x02]	0x00, 0x00

ContractExt.Secrules **Field.**

The $\mathsf{secrules}_\mathsf{A}$ authorization set needs to contain for all the services of applet *A* the set of applets that are authorized to call it (in terms of package AIDs). But for the implementation purposes it's better that we give for each applet in the system which services it may call. Thus, later the search in the policy object will be faster. The secrules table is below.

Caller Package AID: byte array	Callee Method ID: tokens
Application that wants to use a method	

Example 4 *Application* ePurse *allows* jTicket *to use its service* payment. jTicket *has AID* [0x02, 0x02, 0x02, 0x02, 0x02, 0x01] *and* payment *has token* ⟨0x00, 0x00⟩. *Therefore,* ePurse.ContractExt.Secrules *will be represented as following table:*

Caller Package AID: byte array	*Callee Method ID: tokens*
[0x02, 0x02, 0x02, 0x02, 0x02, 0x01]	0x00, 0x00

ContractExt.Funcrules **Field.**

The $\mathsf{funcrules}_\mathsf{A}$ contain all the necessary services for applet *A*. As the funcrules are subset of Calls the representation for them are the same.

5 Mapping

Every applet presented on the card has a unique AID which is too big to operate with frequently on a smart card. As our main goal is to speed up the computations and to reduce the amount of memory needed, we will use the bit vectors. The main assumption required is limitation of the number of applets loaded on the system.

We expect that each card can contain at most 8 loaded packages. This is justified by modern multi-application smart cards, as they usually contain 2-3 applets. Each applet can contain at most 8 services. Thus the set of provided services can be written using just 1 byte per applet. If the system is bigger than 8 loaded packages or some applet wants to provide more than 8 services, the model can be scaled at run-time. Scaling process will be discusses in Section 9. Our representation also eases updates of the system. If some applet is deleted we will not need to restructure the policy.

Thus once the ContractExt object is received by the card it will be transformed. Further we provide details of mapping for AIDs and services into our internal representation.

5.1 Mapping Object

All the mapping information is stored in a Java object Mapping on the card.

The information about the mapping of AIDs is stored in the field Applications of the object Mapping. Applications object is a fixed length Java array of byte arrays. Elements of Applications array are AIDs. Its structure is illustrated in table below:

AID:byte array	Internal ID:byte

Second column is optional and unnecessary for the implementation since elements of Java arrays are numbered by default. It is provided in the report for better understanding of the structures described. Thus, in total for 8 applications on the card Applications will require 136 bytes.

Example 5 *If there are 3 applets:* EMV, ePurse, jTicket *installed on the card, the* Mapping.Applications *object will have the following structure:*

AID:byte array	*Internal ID:byte*
[0x01, 0x01, 0x01, 0x01, 0x01, 0x01]	*0*
[0x01, 0x01, 0x01, 0x01, 0x01, 0x02]	*1*
[0x02, 0x02, 0x02, 0x02, 0x02, 0x01]	*2*
null	*3*
null	*4*
null	*5*
null	*6*
null	*7*

For every application we define a mapping table for its services. For every services in the application we assign a number in a range from 0 to 7. Given that all the applications are now numbered, we store the information about mapping of services in the field Services of the object Mapping. Services is a fixed size Java array of byte arrays. For every application from the Applications objects there is an array of services it provides. Therefore, an element of the Services object is structured as follows:

Method ID:token	Internal Method ID:byte

Again, the column "Internal Method ID" is left for the sake of better illustration and is unnecessary for the implementation. The structure of the Services object is illustrated in the table below:

Table 2: Services object on the card

	Service 0: token	Service 1:token	...	Service 7:token
Application 0			...	
Application 1			...	
...	...	...	...	...
Application 7			...	

Total size of Services is 8*8*2 = 128.

Example 6 *Application* EMV *has one service* transaction *that has the token* 0x00 *and therefore, will be assigned the number* 0*. Application* ePurse *has one service* payment *that has the token* 0x00 *and therefore, will be assigned the number* 0*.*

For the card with applications EMV*,* ePurse *and* jTicket *loaded, the* Mapping.Services *object will have the structure provided in the table 3*

Table 3: Example of Mapping.Services object

	Service 0	S. 1	S. 2	S. 3	S.4	S. 5	S. 6	S. 7
Application 0	0x01	null	null	null	null	null	null	null
Application 1	0x01	null	null	null	null	null	null	null
Application 2	null	null	null	null	null	null	null	null
Application 3	null	null	null	null	null	null	null	null
Application 4	null	null	null	null	null	null	null	null
Application 5	null	null	null	null	null	null	null	null
Application 6	null	null	null	null	null	null	null	null
Application 7	null	null	null	null	null	null	null	null

5.2 WishListServices Object

If during the installation of an applet there are services mentioned in Calls that are not on the card, we put them in the array WishListServices.

Callee Package AID: byte array	Callee Method ID: tokens	Caller Internal ID: byte
Application that contains the desired method, but not on the card yet		Application on the card that wants to use the method

Potentially there can be any number of wished services. For 8 wished services the WishListServices object will require 8*(17+3+17) = 296 bytes.

5.3 MayCallApplets Object

If in Sec.Rules applet allows the access for applets that are not on the card at the moment, we put them in the array MayCallApplets.

Caller Application ID	Callee Application ID	Callee Method ID

Again, there can be any number of authorizations for application not yet on the card. For 16 such authorizations the MayCallApplets object will require 16*(17+17+3) = 592.

When application is being installed, we check the tables and update the card policy (if needed). If applet (service) is removed, and there are some applications having it in secrules(Calls), then we update the tables. Policy update is specified in the section 8.

In total, the Mapping object will require 1276 bytes (under assumption of the specified bounds on the MayCallApplets and WishListServices objects).

6 Internal representation of Contract and Policy Objects on the Card

For every application installed on the card, a ContractExt object that comes with the application is transformed into a new Java object ContractInt. Further we provide details for each of the ContractInt fields. Then we proceed with specification of the security policy object Policy on the card. ContractInt object will be incorporated into the Policy object and will be used in the checks of the PolicyChecker algorithms.

6.1 ContractInt Fields

ContractInt.Funcrules and ContractInt.Calls Bit Representation.

The ContractInt.Calls and the ContractInt.Funcrules are Java byte arrays. Every element of these arrays is a bit array. Each bit is a flag for a service that application calls.

Example 7 *For application* jTicket *the* ContractInt.Funcrules *object will have the following structure:*
[00000000, 00000001, null, null, null, null, null, null]

ContractInt.Provides Bit Representation.

The Provides is represented on the card as a bit array with 1 set on i-th place if service number i is provided. Internal number for services is defined during this service installation (or appearance in someone's Calls set).

6.2 ContractInt.Secrules Bit Representation

Representation of the secrules is similar to the sets previously described. secrules is a byte array, where each byte represents an application. The bit number j in the byte ContractInt.Secrules[i] is set to 1 if the applet grants to applet i an access to the service j.

Example 8 *For the* ePurse *application the* ContractInt.Secrules *object will have the following structure:*
[00000001, 00000001, 00000001, null, null, null, null, null]

6.3 Transformation of Received ContractExt object to Internal ContractInt

Both representations consist of 4 objects: Provides, Calls, secrules$_a$nd funcrules. We will show how each of them transforms, when arrives on the card.

Transformation of Provides.

Object ContractExt.Provides is an array of bytes, where each byte is a name of service it provides. When the applet arrives on the card each method name is mapped to an internal byte number in a range from 0 to 7. Let the arrived application have internal number a. The following algorithm transforms an initial ContractExt.Provides array to an internal object ContractInt.Provides:

```
for i := 0 to 7 do
 if (Mapping.Services[a][i] != null)
   ContractInt.Provides = ContractInt.Provides | (1<<i);
```

Transformation of ContractExt.Funcrules and ContractExt.Calls into Internal Objects.

1. Map AIDs and method IDs of initial objects to the internal representation.

2. For every row in the initial object set a corresponding bit to 1 in the internal object.

Transformation of ContractExt.Secrules.

1. Map AIDs and method IDs of initial objects to the internal representation.

2. For every row in the initial object set a corresponding bit to 1 in the internal object.

In total, an internal contract object ContractInt will require 25 bytes.

6.4 Policy Object Policy on the Card

Before introducing algorithms for policy maintenance we will describe how the internal policy object Policy actually looks like.

Policy object on the card is a static Java object Policy which has 4 fields: Provides, Calls, secrules and funcrules. These fields unify the corresponding sets of each installed application contract.

Although Java Card 2.2.2 does not support multidimensional arrays, some properties are represented as 2-dimensional arrays. Every array is of fixed size and can be easily transformed to 1-dimensional array during the implementation process.

Policy.Secrules Field.

Policy.Secrules is a Java array of byte arrays. Elements of Policy.Secrules are the ContractInt.Secrules objects collected of all the applications installed on the card.

Thus Policy.Secrules has the following structure:

	Caller Application 0	Caller Application 1	...
Callee Application 0	...	11000100	
Callee Application 1	00100101		
...			

Application j can use 0..7 services of each application i.

Policy.Funcrules and Policy.Calls Fields.

Policy.Funcrules and Policy.Calls are Java arrays of byte arrays. Elements of Policy.Funcrules and Policy.Calls are the ContractInt.Funcrules and the ContractInt.Calls objects collected from all the applications installed on the card.

Policy.Funcrules and Policy.Calls have the following structure:

	Callee Application 0	Callee Application 1	...
Caller Application 0	...	11000100	
Caller Application 1	00100101		
...			

Application i can call 0..7 services in each application j.

Policy.Provides Field.

Policy.Provides is a Java array of byte arrays. Elements of Policy.Provides are the ContractInt.Provides objects collected of all the applications installed on the card.

Policy.Provides has the following structure:

Application	Services bit array

Totally, the Policy object will require 200 bytes.

7 The PolicyChecker Algorithms

We proceed as follows. For each type of the update first we recall the algorithms defined for the PolicyChecker in the deliverables D6.3 and D6.4 [5, 4]. Then we provide listings of the PolicyChecker algorithms modified for the proposed on-card models.

7.1 The PolicyChecker for a New Application

Definition 7.1 (Policy Checker for New Applet) *An Optimized* PolicyChecker *(or just* PolicyChecker*) is an algorithm for verification of new application B, that returns true iff the conditions below are true for all applications $A \in \mathcal{A}$ on the platform:*

1. $B \in \mathsf{secrules_A}(\mathsf{Provides_A} \cap \mathsf{Calls_B})$*;*

2. $\mathsf{funcrules_B} \subseteq \bigcup_{\mathsf{A} \in \mathcal{A}} \mathsf{Provides_A}$*;*

3. $A \in \mathsf{secrules_B}(\mathsf{Provides_B} \cap \mathsf{Calls_A})$*;*

The algorithm can be implemented as a sequence of procedures, each of which checks one of the above mentioned conditions. Let application A have a mapping number a and new application B have a mapping number b. We remind that for the PolicyChecker algorithms we will need to use Policy and $\mathsf{ContractInt_B}$ objects.

Let us also define *iCheck* as a boolean variable such that the algorithm returns *true* if $iCheck == 1$ and $false$ otherwise. It has value 1 by initialization.

Check $B \in sec.rules_A(Provides_A \cap Calls_B)$**.**

The procedure must check that for every application A on the card, all the services, such that application B calls and A provides, are allowed for B to call. In other words, for every application A on the card the procedure must do the following:

1. Find the set S_1 of services that A provides and B calls from A.
2. Find the set S_2 of services that A allows B to call.
3. If $S_1 \neq S_2$ than return $false$.

The procedure can be implemented as follows:

Listing 1: PolicyChecker for a New Application.1

```
for a := 0 to 7 do
 if ((byte)Policy.secrules[a][b] & ((byte)Policy.Provides[a]
  & (byte)ContractInt.Calls[a]) !=
   (byte)Policy.Provides[a] & (byte)ContractInt.Calls[a]))
 then iCheck := 0;
```

Check $\mathsf{funcrules_B} \subseteq \bigcup_{\mathsf{A}} \mathsf{Provides_A}$.

$\mathsf{funcrules_B} = \mathsf{ContractInt.Funcrules}$ looks like $[\mathsf{00011000}, ..., \mathsf{00100010}]$ (1 byte per application, bit is a flag for service). $\mathsf{Provides_A} = \mathsf{Policy.Provides}[a]$ looks like: 00011111(bit is a flag for service).

The procedure that checks the condition can be implemented as follows:

Listing 2: PolicyChecker for a New Application.2

```
for i := 0 to 7 do
  if (ContractInt.Funcrules[i] & Policy.Provides[i] !=
      ContractInt.Funcrules[i])
  then iCheck := 0;
```

Check $A \in \mathsf{secrules_B}(\mathsf{Provides_B} \cap \mathsf{Calls_A})$.

$\mathsf{Provides_B} = \mathsf{ContractInt.Provides_B}$ is a bit field. $\mathsf{Calls_A}$ is a cell in the table Policy.Calls.

Table 4: Policy.Calls table

	Calls in Application 0	Calls in Application 1	...
Application 0	00000000	00000001	...
Application 1	00100100	00110001	...
...	...	...	...

7.2 The PolicyChecker for Removal of an Applet

Definition 7.2 (Policy Checker for Removal of an Applet) *A* PolicyChecker *for verification of security of the platform after removal of an application $B \in \mathcal{A}$ is an algorithm , that returns true iff the conditions below are true for all applications $A \in \mathcal{A}$, $A \neq B$ on the platform:*

- $\mathsf{Provides_B} \cap \{\bigcup_{\mathsf{A} \in \mathcal{A}} \mathsf{funcrules_A}\} = \emptyset$; .

$\mathsf{Provides_B} \cap \left\{ \bigcup \mathsf{funcrules_A} \right\}$.

Listing 3: The PolicyChecker for Removal of an Applet

```
for a := 0 to 7 do
  if ((Policy.FuncRules[a][b] & Policy.Provides[b]) != 0)
  then iCheck := 0;
```

7.3 The PolicyChecker for Update of an Applet

Definition 7.3 (Policy Checker for Updated Applet) *A* PolicyChecker*for verification of update in the application $B \in \mathcal{A}$ is an algorithm, that returns true iff the conditions below are true for all applications $A \in \mathcal{A}$, $A \neq B$ on the platform:*

1. *Addition of a service s to* $\mathsf{Provides_B}$*: $A \in \mathsf{sec.rules_B}(\mathsf{s} \cap \mathsf{Calls_A})$;*
2. *Removal of a service s from* $\mathsf{Provides_B}$*: $s \notin \bigcup_{A \in \mathcal{A}} \mathsf{funcrules_A}$;*
3. *Addition of a service s to* $\mathsf{Calls_B}$*: $B \in \mathsf{sec.rules_A}(\mathsf{Provides_A} \cap \mathsf{s})$;*
4. *Removal of a service s from* $\mathsf{Calls_B}$*: return true;*
5. *Addition of an authorization rule for some application C to a service s of B in* $\mathsf{secrules_B}$*: return true;*
6. *Removal of an authorization rule for some application C to a service s of B from* $\mathsf{secrules_B}$*: $s \notin \mathsf{Calls_C}$;*
7. *Addition of a service s to* $\mathsf{funcrules_B}$*: $s \in \bigcup_{A \in \mathcal{A}} \mathsf{Provides_A}$*
8. *Removal of a service s from* $\mathsf{funcrules_B}$*: return true;*

Addition of a service s to $\mathsf{Provides_B}$.

Check: $A \in \mathsf{secrules_B}(\mathsf{s} \cap \mathsf{Calls_A})$

```
for a:= 0 to 7 do
  if (Policy.Calls[a][b] & (1 << s) != 0)
  then if (Policy.Secrules[b][a] & (1 << s) != 0)
       then iCheck := 0;
```

Removal of a service s from $\mathsf{Provides_B}$.

Check: $s \notin \bigcup_A \mathsf{funcrules_A}$

```
for a := 0 to 7 do
  if ((Policy.Funcrules[a][b] & (1 << s) != 0)
  then iCheck := 0;
```

Addition of a service s to $\mathsf{Calls_B}$.

Check: $B \in \mathsf{secrules_A}(\mathsf{Provides_A} \cap \mathsf{s})$;

```
if (Policy.Provides[a] & (1 << s) != 0)
then if (Policy.Secrules[a][b] & (1 << s) != 0)
     then iCheck := 0;
```

Removal of a service s from $\mathsf{Calls_B}$.

This update does not break security of a platform, thus no check is performed.

Addition of an authorization rule for some application C to a service s of B in $\mathsf{secrules_B}$.

This update does not break security of a platform, thus no check is performed.

Removal of an authorization rule for some application C to a service s of B in $\mathsf{secrules_B}$.

Check: $S \notin \mathsf{Calls_C}$

```
if (Policy.Calls[c][b] & (1 << s) != 0)
then iCheck := 0;
```

Addition of a service s to $\mathsf{funcrules_B}$.

Check: $s \in \bigcup_{A \in \mathcal{A}} \mathsf{Provides_A}$

We don't have standalone service s in the model, it's always in some application A: $A.s$. So we need to check only that A provides that service.

```
if (Policy.Provides[a] & (1 << s) == 0)
then iCheck := 0;
```

Removal of a service s from $\mathsf{funcrules_B}$.

This update does not break functionality of a platform, thus no check is performed.

8 Policy Update

In this section we provide listings of algorithms that are necessary for Policy object update. These algorithms are specified separately for each kind of update (scaling necessity is not expected).

8.1 New Application

Following is the script of the algorithm for installation of a new application B.

1. Make a copy of the objects Policy, WishListServices, MayCallApplets and Mapping.

2. Associate a number with a new application: iterate through applications mapping table Mapping.Applications and find the first empty spot.

3. Map methods: simply associate with numbers in natural order 0..M.

4. Convert $\mathsf{ContractExt_B}$ object on the applet to the card contract format $\mathsf{ContractInt_B}$ (using the mapping tables Mapping.Applications and Mapping.Services)

    (a) If there are services mentioned in $\mathsf{ContractExt.Calls_B}$ that are not on the card, update the WishListServices object.

    (b) If there are applications mentioned in $\mathsf{ContractExt.Secrules_B}$ that are not on the card, update MayCallApplets object.

5. Search in WishListServices and MayCallApplets for AID of B, update Policy object on the card.

6. Run the PolicyChecker algorithms for new application installation.

(a) If the PolicyChecker returns $false$, restore Policy, WishListServices, MayCallApplets and Mapping from the backup copies, **stop installation**.

(b) If Policy Checker returns $true$, update Policy object on the card: fill in corresponding rows in Policy.Calls, Policy.Provides, Policy.Secrules, Policy.Funcrules tables), delete copied objects, **proceed with installation**.

8.2 Removal of an Application

Further we provide a script of an algorithm for removal of already installed application B.

1. Upon receiving a request for removal of an application B, find a mapping number b of this applet in the Mapping.Applications.

2. Run corresponding PolicyChecker algorithm. If the PolicyChecker returns $false$, **stop removal**.

3. Search in Policy.Secrules for occurrences of application B authorization by some application. If found, for every occurrence do:

   (a) Delete occurrence from the Policy.Secrules object;

   (b) Add information about authorization to MayCallApplets object;

4. Search in Policy.Calls for occurrences of calls to services of B. If found, for every occurrence do:

   (a) Delete occurrence from Policy.Calls object;

   (b) Add information about potential call to WishListServices object;

5. Delete rows of the applet B from Policy.Provides, Policy.Calls, Policy.Secrules, Policy.Funcrules objects;

6. Delete rows of the applet B from Mapping.Applications, Mapping.Services objects.

8.3 Update of an Existing Application

Below we provide a specification of the algorithm for an update of existing application B. We consider an atomic updates now, for a sequence of atomic updates all corresponding checks need to be incorporated.

0. Upon receiving a request for update of an application B, find a mapping number b of this applet in the Mapping.Applications. If b is not found, then **stop update**.

Then, depending on what the case is, do one of the following:

1. Addition of a service s to $\mathsf{Provides_B}$
   (a) Make a copy object of Policy, WishListServices and Mapping.
   (b) Search in the WishListServices and MayCallApplets objects for an occurrence of s and update the Policy object if an occurrence is found;
   (c) Add the service s to Mapping.Services in the first available cell in the row Mapping.Services[b];
   (d) Run corresponding PolicyChecker algorithm. If PolicyChecker returns $false$, restore objects from copies and **stop update**.
   (e) Add the internal ID of s to Policy.Provides[b] object;

2. Removal of a service s from $\mathsf{Provides_B}$
   (a) Run corresponding PolicyChecker algorithm. If PolicyChecker returns $false$, **stop update**.
   (b) Clear cell s in Policy.Provides[b];
   (c) Search Policy.Calls for occurrence. If found, for every occurrence do:
      i. Delete occurrence from Policy.Calls;
      ii. Add to related information to WishListServices object;
   (d) Clear cell s in Mapping.Services object;

3. Addition of a service $A.s$ to $\mathsf{Calls_B}$

(a) If the applet A is present in Mapping.Applications under number [a] and s is present in Mapping.Services[a] under number [s], then run corresponding PolicyChecker algorithm. If PolicyChecker returns $false$, **stop update**. If PolicyChecker returns $true$, add internal ID of s to Policy.Calls[b];

(b) If there is no A on the card or s is not in $\mathsf{Provides}_\mathsf{A}$, then add $A.s$ to WishListServices.

4. Removal of a service $A.s$ from $\mathsf{Calls}_\mathsf{B}$:

   (a) If the applet A is present in Mapping.Applications under number [a] and is present in Mapping.Services[a] under number [s], then clear bit s in Policy.Calls[b][a].

   (b) If there is no applet A on the card or A does not provide service s, then search WishListServices for occurrences of $A.s$ and delete if found.

5. Addition of an authorization rule for some application C to a service s of B in $\mathsf{secrules}_\mathsf{B}$:

   (a) If s is not present in Policy.Provides[b], then add the correspondent rule to MayCallAppletsand **stop update**;

   (b) If the applet C is present in Mapping.Applications, then set the bit s in Policy.Secrules[b][c] to 1;

   (c) If the applet C is not on the card, add the correspondent rule to MayCallApplets;

6. Removal of an authorization rule for some application C to a service s of B from $\mathsf{secrules}_\mathsf{B}$:

   (a) If the applet C is not on the card, search MayCallApplets for occurrences of $B.s$ and remove if found.

   (b) If the applet C is present on the card, then run corresponding PolicyChecker algorithm. If PolicyChecker returns $false$, **stop update**;

   (c) Set the bit s in Policy.Secrules[b][c] to 0;

7. Addition of a service $A.s$ to $\mathsf{funcrules}_\mathsf{B}$:

(a) If the applet A is present in Mapping.Applications under number [a] and s is listed in Policy.Provides[a], then run corresponding PolicyChecker algorithm. If PolicyChecker returns $false$, **stop update**;

(b) Add internal ID of s to Policy.Funcrules[b];

8. Removal of a service s from $\mathsf{funcrules_B}$:

   (a) Clear cell in Policy.Funcrules[b];

9 Scaling and Analysis of the Model

If more than 8 applets arrive or an applet provides more than 8 services, the model is scaled at runtime by doubling the correspondent data objects. This creates a limit of 16 for a number of running applications or a number of services per applet. If at some point it will not be enough, the model is scaled again.

9.1 Analysis of costs of the algorithms

We define the following constants for the general case:

- N is the current size of the Mapping.Applications object (the maximum of active applications);
- M is the current maximum of the number of services per application;
- P is the maximum number of applications one applet can call, $P \sim O(N)$;
- R is the maximum number of applications one applet can allow access to, $R \sim O(N)$;
- U is the maximum size of MayCallApplets, constant;
- W is the maximum size of WishListServices, constant.

These constants define complexity bounds for our algorithms. Further we provide a set of tables with analyzes of algorithmic costs for each type of update. We also investigate an average case example.

- Number of operations needed to install new application is provided in the Table 5. These estimates are derived from a simple refinement of the high level algorithms used in Sections 8.1, 6.3;
- Number of operations necessary for removal of an application is provided in the Table 6. These estimates are derived from a simple refinement of the high level algorithms used in Sections 8.2;
- Number of operations for each atomic update of an application is provided in the Table 7.These estimates are derived from a simple refinement of the high level algorithms used in Sections 8.3;
- For the low complexity example, when $N = 8$, $M = 8$, $P = 16$, $R = 16$, $U = 50$, $W = 50$, number of operations for install, delete and update of an application is provided in the Table 8.

9.2 Intuitive explanation of costs of the algorithms

In this section we try to explain the results provided in the Table 8 for the low complexity case.

Removal of a service s from $\mathsf{Provides_B}$ requires more operations than adding a service s to $\mathsf{Provides_B}$ because, when removing a service, we need to check all contracts for occurrences of s and for every occurrence perform a search and update of WishListServices. When adding a service, we need to perform a search in WishListServices just one time.

Addition of a service $A.s$ to $\mathsf{Calls_B}$ requires the number of operations of the same order of magnitude as removal, because both algorithms perform similar searches in Mapping.Applications, Mapping.Services and MayCallApplets objects. Addition of a service to $\mathsf{Calls_B}$ requires also an execution of the PolicyChecker algorithm, that is not executed when a service is being removed, but its cost is significantly less than the cost of the searches in Mapping.Applications, Mapping.Services and MayCallApplets objects.

Removal of a service s from $\mathsf{Provides_B}$ requires significantly more operations than removal of a service $A.s$ from $\mathsf{Calls_B}$, because the removal of a service from $\mathsf{Provides_B}$ means actually removing a service from the card, which may cause functionality conflicts with other applications and thus requires execution of a few checks. Removal of a service from $\mathsf{Calls_B}$ can cause no conflicts with other applications, thus it is quick.

Table 5: Costs of the algorithm for the new application installation

Operation	Maximum number of operations	Complexity
Mapping of the AID	N	$O(N)$
Mapping of services IDs	M	$O(M)$
Converting ContractExt to Contract:		
FuncRules	$((N-1)\times M)(N+M+3)$	$O(MN^2+M^2M)$
Calls (incl. update of WishListServices)	$(P\times M)(N+M+3)+W$	$O(MN^2+M^2N)$
SecRules (incl. update of MayCallApplets)	$(R\times M)(N+3)+U$	$O(MN^2)$
Provides	$M\times 4$	$O(M)$
Searching in WishListServices and MayCallApplets for the AID and updating Policy	$U\times(N+M+3)+$ $+W\times(M+N+3)$	$O(M+N)$
Verifying Contract with PolicyChecker	$N\times 4+N\times 2+N\times 4$	$O(N)$
Overall complexity: $O(MN^2+M^2N)$		

Addition of an authorization rule for some application C to a service s of B in $\mathsf{secrules_B}$ requires the number of operations of the same order of magnitude as removal of an authorization rule, because both algorithms perform similar searches in Mapping.Applications and MayCallApplets objects. Removal of an authorization rule requires also an execution of the PolicyChecker algorithm, that is not executed when an authorization rule is being added, its cost is significantly less than the cost of the searches in Mapping.Applications and MayCallApplets objects.

References

[1] R. Akram, K. Markantonikas, and K. Mayes. A paradigm shift in the smart card ownership model. In *Proc. of ICCSA 2010*, volume 6019 of

Table 6: Costs of the algorithm for the removal of an application

Operation	Maximum number of operations	Complexity
Run PolicyChecker	$N \times 2$	$O(N)$
Remove links in Policy.SecRules & update MayCallApplets	$N \times (M \times (3M + U) + 2)$	$O(M^2N))$
Remove links in Policy.Calls & update WishListServices	$N \times (M \times (3M + W) + 2)$	$O(M^2N)$
Clear Policy.Provides, Policy.Calls, Policy.SecRules, Policy.FuncRules correspondent rows	$1 + N \times 3$	$O(N)$
Clear references in Mapping.Services and Mapping.Applications	$M + 1$	$O(M)$
Overall complexity: $O(M^2N)$		

LNCS. SV, 2010.

[2] A. Armenteros, B. Chetali, M. Felici, V. Meduri, Q-H. Nguyen, A. Tedeschi, F. Paci, and E. Chiarani. D1.1 Description of the scenarios and their requirements. *SecureChange EU project public deliverable, www.securechange.eu*, 2010.

[3] Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[4] A. Fontaine, O. Gadyatskaya, F. Massacci, J. Bernet, F. Bouquet, and Q.H. Nguyen. D6.4 impact analysis of new security requirements. *SecureChange EU project public deliverable, www.securechange.eu*, 2010.

[5] A. Fontaine, S. Hym, I. Simplot-Ryl, O. Gadyatskaya, F. Massacci, F. Paci, J. Jurgens, and M. Ochoa. D6.3 compositional technique to verify adaptive security at loading time on device. *SecureChange EU project public deliverable, www.securechange.eu*, 2010.

[6] P. Girard. Which security policy for multiplication smart cards? In *USENIX Workshop on Smartcard Technology*. USENIX Association, 1999.

[7] Sun Microsystems. Application programming interface specification. Java Card^{TM} platform, version 2.2.2. Specification 2.2.2, Sun Microsystems, 2006.

[8] Sun Microsystems. Runtime environment specification. Java Card^{TM} platform, version 2.2.2. Specification 2.2.2, Sun Microsystems, 2006.

[9] Sun Microsystems. Runtime environment specification. Java Card^{TM} platform, version 2.2.2. Specification 2.2.2., Sun Microsystems, 2006.

[10] Sun Microsystems. Virtual machine specification. Java Card^{TM} platform, version 2.2.2. Specification 2.2.2, Sun Microsystems, 2006.

[11] Oracle. Java Card platform: A week in the life of Jack Link, http://www.oracle.com/technetwork/java/javacard/week-140205.html. Available on the web, Retrieved on Febr. 2011.

[12] D. Sauveron. Multiapplication smart card: Towards an open smart card? *ISTR*, 2009.

Table 7: Costs of the algorithms for the update of an application

Type of update	PolicyChecker algorithm	Policy update algorithm	Complexity
Addition of a service s to $\mathsf{Provides_B}$	$N \times 7$	$W + M + 3$	$O(M + N)$
Removal of a service s from $\mathsf{Provides_B}$	$N \times 4$	$4 + N \times (W + 8) + 1$	$O(N)$
Addition of a service $A.s$ to $\mathsf{Calls_B}$	7	$M + N + W + 3$	$O(M + N)$
Removal of a service $A.s$ from $\mathsf{Calls_B}$	0	$M + N + W + 4$	$O(M + N)$
Addition of an authorization rule for some application C to a service s of B in $\mathsf{secrules_B}$	0	$N + U + 3$	$O(N)$
Removal of an authorization rule for some application C to a service s of B from $\mathsf{secrules_B}$	4	$N + U + 4$	$O(N)$
Addition of a service $A.s$ to $\mathsf{funcrules_B}$	4	$N + M + 3$	$O(M + N)$
Removal of a service $A.s$ from $\mathsf{funcrules_B}$	0	4	$O(1)$
Overall complexity: $O(M + N)$			

Table 8: Costs of the algorithms for the low complexity case

Procedure	Number of operations
Install new application	7032
Remove an application	9554
Add a service s to $\mathsf{Provides_B}$	117
Remove a service s from $\mathsf{Provides_B}$	508
Add a service $A.s$ to $\mathsf{Calls_B}$	76
Remove a service $A.s$ from $\mathsf{Calls_B}$	70
Add of an authorization rule for some application C to a service s of B in $\mathsf{secrules_B}$	61
Remove an authorization rule for some application C to a service s of B from $\mathsf{secrules_B}$	66
Add a service $A.s$ to $\mathsf{funcrules_B}$	23
Remove a service $A.s$ from $\mathsf{funcrules_B}$	4