High-level algorithms and data structures requirements for security-by-contract on Java cards

Nicola Dragoni

DTU Informatics,
Technical University of Denmark,
2800 Kgs. Lyngby, Denmark
E-mail: ndra@imm.dtu.dk

Olga Gadyatskaya*, Fabio Massacci and Anton Philippov

Department of Information Engineering and Computer Science,
University of Trento,
Povo, Trento, 38123, Italy
E-mail: olga.gadyatskaya@unitn.it
E-mail: fabio.massacci@unitn.it
E-mail: anton.philippov@unitn.it
*Corresponding author

Abstract: The Java Card technology has progressed to the point of running web servers and web clients on a smart card. Yet concrete deployments of multi-applications smart cards have remained extremely rare because the business model of the asynchronous download and update of applications by different parties requires the control of interactions among possible applications after the card has been fielded. The current security models and techniques do not support this type of evolution. We propose in this paper to apply the notion of security-by-contract ($S \times C$), that is a specification of security-related behaviour of an application that must be compliant with the security policy of the hosting platform. This compliance can be checked at the application loading time, avoiding in this way the need of costly runtime monitoring. We show how $S \times C$ can be used to prevent illegal information exchange among applications on a single smart card platform in presence of dynamic changes on the card.

Keywords: security-by-contract; $S \times C$; load time verification; Java Card; application interactions; security policies for multi-application smart cards.

Biographical notes: Nicola Dragoni received his MSc and PhD in Computer Science, respectively in 2002 and 2006, both at University of Bologna, Italy. He visited the Knowledge Media Institute at the Open University (UK) in 2004 and the MIT Centre for Collective Intelligence (USA) in 2006. In 2007 and 2008, he was a post-doctoral research fellow at University of Trento, working on security for mobile systems. Between 2005 and 2008, he also worked as freelance IT consultant. In 2009, he joined Technical University of Denmark (DTU) as an Assistant Professor in security and distributed systems and was promoted to an Associate Professor in 2011.

Olga Gadyatskaya received her PhD in Mathematics in 2008 at Novosibirsk State University (Russia). In 2007 and 2008, she also worked as a researcher at the Institute of Computational Mathematics and Mathematical Geophysics (Novosibirsk, Russia). Since 2009, she has been a post-doctoral research fellow in Department of Information Engineering and Computer Science of the University of Trento (Italy). Her research interests include load time verification and run-time enforcement of information and control flow policies.

Fabio Massacci received his MEng in 1993 and PhD in Computer Science and Engineering at University of Rome La Sapienza in 1998. He joined University of Siena as an Assistant Professor in 1999, and was a visiting researcher at IRIT Toulouse in 2000, and joined University of Trento in 2001 where he is currently a Full Professor. His research interests are in security requirements engineering, verification for load time security and software vulnerabilities studies. He is currently a scientific coordinator of a multimillion R&D European project on sociological and economic aspects in security (http://seconomicsproject.eu/).

Anton Philippov received his MS in Mathematics from Novosibirsk State University (Russia). From 2008 to 2010, he was a research student at the Intel-NSU Laboratory (Novosibirsk, Russia). In 2008 and 2009, he also worked as a software developer at Sibers (Novosibirsk, Russia). He is currently a PhD candidate in the Information and Communication Technology Doctoral School at the University of Trento, Italy. His research interests include security of service platforms and software verification.

This paper is a revised and expanded version of a paper entitled ‘Can we support applications’ evolution in multi-application smart cards by security-by-contract?’ presented at Workshop on Information Security Theory and Practices (WISTP-2010), Passau, Germany, April 2010.

1 Introduction

Open multi-application smart cards aim at making it possible to run several applications on a single smart card and to dynamically load and remove applications during the card’s active life. However, in spite of the large number of research papers on the topic there are very few real-life deployments.
One reason is the lack of solutions to an old problem: the control of interactions among applications. Many techniques can be used to check information flow (Bieber et al., 2002; Hubbers et al., 2003; Ghindici and Simplot-Ryl, 2008) if we know and install all applications at once before distributing the card to the public. Yet the natural business model for open multi-application smart cards is the asynchronous loading and updating of applications by different parties. Hence we need a method to check application interactions at load or run time.

A firewall security mechanism run by the Java Card run-time environment (JCRE) isolates each application from the other applications within its own space. Thus internal operations of an application have no effect on other applications embedded on the card. Applications can interact in this environment by explicitly implementing shared methods (application services). So, if application \( A \) knows a service of application \( B \), then it may use it for its own purposes, and there is no way for \( B \)'s owner to prevent this, unless special controls are hardwired into \( B \)'s code. This problem hinders the asynchronous download or update of different applications.

What we need is a quick way to deploy new applications on a smart card once it is in the field without a costly manual review. Owners of different trust domains would like to make sure their applications cannot be accessed by new applications added after their ones. Actually, current smart card developers have to prove that all the changes that can be applied to the card are security-free, so that their formal proof of compliance with common criteria is still valid and they do not need to obtain a new certificate (e.g., Narasamdya and Perin, 2009; Tanaka, 2008). Otherwise no smart card customers (banks, governments, airline companies, etc.) will accept to issue such cards for their needs. The natural consequence is that there are essentially no interacting multi-application smart cards, though the Java Card specifications support them.

1.1 Contribution of the paper

We claim that open multi-application smart cards need the notion of security-by-contract \((S\times C)\). A contract is a description of the security behaviour of an application in terms of interactions the application might have with the hosting smart card and with the applications running on the same card. The intuition is that a new application is loaded on the card if its contract is compliant with the smart card policy, i.e., the combined constraints of all the pre-loaded applications. Otherwise the application is rejected.

In this paper we focus on contracts and policies describing the interactions among the different applications running on a single smart card platform. Intuitively, we say that an application \( A \) interacts with an application \( B \) if either \( A \) calls a service from \( B \), or \( B \) calls a service from \( A \). In this dynamic setting, we are primarily interested in addressing two specific security challenges:

1. **Problem \( P_1 \):** a new application \( A \) should not interact with applications already on the smart card unless they have explicitly permitted the interactions in their policies.
2 Problem \( P_2 \): a dynamic change must not affect the correct execution of an application on the smart card. In particular, an application must be able to work when the following changes happen:

- \( P_{2a} \): addition of a new application to the smart card
- \( P_{2b} \): update of an existing application
- \( P_{2c} \): changes in the platform security policy
- \( P_{2d} \): removal of some existing application from the smart card.

We present contract and policy notations, specifying applications in terms of provided and called services and focus on a concrete contract notation that can be implemented on a smart card, and detail the contract-policy matching approach for this notation. This approach is memory-saving and enables efficient contract-policy matching.

In the next section we introduce the \( S \times C \) workflow for smart cards and present a hierarchy of contract-policy models. Section 3 provides an overview of the Java Card technology and highlights the necessary details of on-card application interactions. Section 4 contains definitions of an application contract and a smart card security policy on the level L0 of the proposed hierarchy. We present an updated Java Card architecture with the \( S \times C \) components added. The on-card algorithms of the contract-policy matching are listed in Section 6. We overview the related work in Section 7 and conclude in Section 8.

2 \( S \times C \) for smart cards

The \( S \times C \) framework was developed for mobile code (Desmet et al., 2008; Dragoni et al., 2007) building upon the notion of model carrying code (MCC, Sekar et al., 2003). In \( S \times C \) the mobile code is augmented with a claim on its security behaviour (an application’s contract) that has to be matched against the mobile platform’s policy before downloading the code. The key intuition is that in this way a digital signature does not just certify the origin of the code but also binds together the code with a contract, providing a semantics for the digital signature, as advocated by Dragoni et al. (2009).

In a nutshell, at the load time the target platform follows a workflow similar to the one depicted in Figure 1. The first step concerns the trustworthiness of the mobile application that one wants to load on the smart card. To do this one needs some evidence to be checked on the platform. Such evidence can be a trusted signature as usually done for mobile applications or a proof that the code satisfies the contract [by means of the proof-carrying-code techniques (Necula, 1997)].

Once there is the evidence that the contract is trustworthy, the platform checks whether the contract is compliant with the policy that the platform wants to enforce. This is a key phase called contract-policy matching. If they do match, then the application can be run without further ado, because the application is compliant with both a trusted contract and the platform’s security policy. In contrast, if this match results in a failure, then we might reject the application or use another means to enforce the smart card’s security policy. This can be done, for instance, through inlining techniques or
monitoring the application at run-time. In both cases the application will run with some overhead.

The S×C approach for mobile platforms proposed to use an inlined reference monitor (IRM) for run-time monitoring of non-compliant applications (Desmet et al., 2008), but smart cards do not provide the means to inline the policy. Thus the S×C workflow for smart cards does not comprise the step of policy inlining, nor it allows to install potentially dangerous applications due to sensitivity of smart card applications. Therefore, in case of any fails during contract-code or contract-policy matching the application should be rejected.

**Figure 1** S×C for load time evolution (see online version for colours)

2.1 *A hierarchy of contract/policy models*

In our threat model we assume that an adversary can try to load her applications on the card in order to access sensitive services of the installed applications. The adversary can behave in the same way as honest application providers – she can communicate with her own applets and use the card mechanisms to remove/update her applets. We assume the adversary cannot impersonate other application providers and spoof their applications; reliable mechanisms to authenticate entities communicating with the card are provided, for example, by the GlobalPlatform card management framework.

Let us now intuitively define the *contract-policy matching* on card. A contract of an application A matches a platform policy if there is no illegal information exchange between the application A and the applications already on the card. Intuitively, there is an illegal information exchange between two applications A and B if the interaction between these applications might cause an information leak. For instance, if A is allowed to interact with a party C while B is not, then communication between A and B might cause an illegal disclosure of information from C to B through A. We can call this scenario a *collusion* of A and B.
Example 1: Let us consider an application $EMV$ of a company $Bank$ carrying the following contract $\text{Contract}_{EMV}$: “$EMV$ provides a service $\text{fill\_purse}$ and it is willing to allow the usage of this service only to application $\text{Credit}$ of the company $Bank$. It also needs the service $\text{check\_credit}$ of the application $\text{Credit}$ in order to function properly”.

Let us suppose that the card has the following policy:

- “The application $\text{Credit}$ of the company $Bank$ provides the service $\text{check\_credit}$ and allows the usage of this service only to applications $EMV$ and $\text{Debit}$ of the company $Bank$”.
- “The application $\text{Weather}$ of the company $Sky$ wants to interact with the service $\text{get\_location}$ of the application $Geo$, but can provide basic functionality without it”.

We can easily notice that the security contract $\text{Contract}_{EMV}$ of the application $EMV$ matches the security policy of the card.

Contracts can be specified at different levels of abstraction as defined below:

- $L0$: application as services. This level models applications as a list of called and provided services. Information exchange between applications on this level does not take into account collusions between applications, but only a direct service invocation is considered as a security-related event.
- $L1$: allowed control flow. This level provides a call graph of an application $A$, where vertices are the states of the application and edges represent the invocation of different services. Then we can do a bit of history-based access control and more fine grained information exchange control.
- $L2$: allowed and desired control flow. This level adds to the previous one the notions of correct and error states.
- $L3$: full information flow. This level extends the previous one considering also the information flow among variables.

In the rest of the paper we will discuss the level $L0$ in practical details highlighting its limitations. However this model has the advantage of being practically implementable on a real smart card which has severe resource constraints.

3 A primer on the Java Card technology

The architecture of the Java Card platform is depicted in Figure 2. The architecture comprises several layers, which include device hardware, an embedded operating system (native OS), the JCRE and the applications installed on top of it. Important parts of the JCRE are the Java Card virtual machine (JCVM) (its Interpreter part) and the installer, which is an entity responsible for loading and installation of applications.

Applications (also known as applets in the Java Card jargon) are supplied on the card in packages. The source code of a package is converted by the developer into class files and then into a converted applet (CAP) file. A CAP file is a JAR-format file which contains the executable binary representation of the classes in a Java package. A
CAP file consists of several components, which typically arrive on a card in a known order [Sun Microsystems, Inc., (2006), Chapter 6]. The information in each of the component is organised in a way which allows any implementation of the installer and the JCVM to process the received CAP file. The CAP file is transmitted onto a card, where it is processed, linked and transformed into a platform-specific executable format (defined by the platform vendor). Application providers do not need to consider different on-card executable formats, the only requirement is compliance of the CAP file with the Java Card specifications. Then, upon finalisation of the linking process, an applet instance is installed.

Figure 2  The Java Card architecture (see online version for colours)

3.1 Application interactions

Applications on Java Card are separated by a firewall, which 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 (Sun Microsystems, Inc., 2006). 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 an interface that extends the javacard.framework.Shareable interface.

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 service usage starts with a client’s request to the JCRE for a reference to \( A \)’s object (that is implementing a 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 (Sun Microsystems, Inc., 2006).

In order to interact the client has to import the shareable interface of the server. In addition, the client needs to obtain the export file of the server, which lists the 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 instead of the symbolic names, thus during conversion from class files into a CAP file the client needs to know the 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.

### 3.2 Tokens and token-based linking

Tokens are used by the JCRE for on-card linking 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. 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 a unique application identifier (AID) of a package providing the service \( s \), \( I \) is a token for the shareable interface and \( t \) is a token for the method \( s \) in the interface \( I \).

### 4 Contract and policy as lists of services

Let us now present definitions of application contracts and smart card policy for the level L0. The main idea is that each applet can declare authorisation rules for its services in the contract, instead of embedding the access controls into the code. Let \( A \) be a set of applications already loaded on the platform. We denote applications already loaded as \( A \) or \( A_i \) and the application that is affected by a change (or new applet being installed) as \( B \). We use notation \( A:s \) for service \( s \) of application \( A \).

We extend the \( S \times C \) definition from Dragoni et al. (2007) by dividing the contract \( \text{Contract}_A \) of application \( A \) into two parts:

- The \( \text{Claim}_A \) describes the actual behaviour of application \( A \). \( \text{Claim}_A \) contains two parts: the services that are called by \( A \) and the published services of \( A \).
- The \( \text{AppPolicy}_A \) is a security policy declared by the application provider, specifying the desirable behaviour of other applications on the platform w.r.t. this application.

An update of some application in \( S \times C \) approach will be reflected in the Claim update or (and) AppPolicy update.

**Contract:** For application \( A \) its \( \text{Contract}_A \) is a pair \( \langle \text{Claim}_A, \text{AppPolicy}_A \rangle \), where
292  N. Dragoni et al.

- Claim\textsubscript{A} = \langle Provides\textsubscript{A}, Calls\textsubscript{A} \rangle, and Provides\textsubscript{A} is a set of services provided by \textit{A}, Calls\textsubscript{A} is a set of services called by \textit{A}

- AppPolicy\textsubscript{A} = \langle sec.rules\textsubscript{A}, func.rules\textsubscript{A} \rangle, and sec.rules\textsubscript{A} is a set of authorisations for services access, func.rules\textsubscript{A} is a set of functionally necessary services.

We assume that sec.rules\textsubscript{A} operates on values from Provides\textsubscript{A} and func.rules\textsubscript{A} \subseteq Calls\textsubscript{A}.

**Platform security policy:** Security policy of the platform \( \mathcal{P} \) consists of the contracts of all the applications \( A \) on the platform:

\[
\mathcal{P} = \bigcup_{A \in \mathcal{A}} \{ \text{Contract}_A \}
\]

**Example 2:** The contract of the application \textit{EMV} of a company \textit{Bank} described in Example 1 from Section 2.1 can be expressed in the above notation as follows:

\text{Contract}_{EMV} = \langle\langle \text{Provides}_{EMV}, \text{Calls}_{EMV} \rangle, \langle \text{sec.rules}_{EMV}, \text{func.rules}_{EMV} \rangle \rangle,\] where

- Provides\textsubscript{EMV} = \{\text{fill\_purse}\}
- Calls\textsubscript{EMV} = \{\text{Credit\_check\_credit}\}
- sec.rules\textsubscript{EMV} = \{\text{fill\_purse} \rightarrow \{\text{Credit}\}\}
- func.rules\textsubscript{EMV} = \{\text{Credit\_check\_credit}\}.

The policy of the card that hosts 2 applets (\textit{Credit} and \textit{Weather}) is following:

\[
\mathcal{P} = \{ \text{Contract}_{Credit}, \text{Contract}_{Weather} \},
\]

where

- \text{Contract}_{Credit} = \langle\langle \{\text{check\_credit}\}, \emptyset \rangle, \langle \{\text{check\_credit} \rightarrow \{\text{EMV}, \text{Debit}\}\}, \emptyset \rangle \rangle
- \text{Contract}_{Weather} = \langle\langle \emptyset, \{\text{Geo\_get\_location}\} \rangle, \langle \emptyset, \emptyset \rangle \rangle.

**Figure 3** The S\texttimes C architecture for Java Card (see online version for colours)
5 The S×C framework for Java cards

The S×C 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 in Figure 3, where the additions to the JCRE are in long dashed blue rectangles.

Every applet arrives on the card equipped with its contract that contains information regarding provided and called services and the applet’s policy. On the card the ClaimChecker verifies during the loading 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 inconsistencies, the applet’s installation process is stopped and it is rejected from the card. Consequently, only applications with compatible contracts can be installed on the card.

We present in this section a high-level version of the workflow that will be performed by the S×C framework, while the components will be described in detail later in the paper. In the case of loading of a new application, the following actions will take place:

1. A package A is sent to the card together with its contract Contract_A.
2. The ClaimChecker ensures that the received Contract_A is compliant with the bytecode.
3. The PolicyChecker runs the contract-policy matching algorithms.
4. If the PolicyChecker returned True, the application A is installed on the platform and the security policy is updated correspondingly (to include Contract_A).
5. If the PolicyChecker returned False, the loading process is stopped. The card returns to the previous state (it is possible due to the Java Card transaction mechanism).

The S×C framework verifies that the following two properties will be satisfied on the card after every possible evolution identified in \( P_{2a,d} \):

- **service invocation security**: if an application A calls during its execution a service \( s \) of an application B, then B has authorised A to access \( s \) in B’s security policy.
- **available functionality**: if an application A has declared that it needs a service \( s \) of an application B in order to be functional, then the service \( s \) is indeed provided by B.

Thus the S×C framework addresses problem \( P_1 \) for direct service invocations.

5.1 The claim checker component and contract structure

Intuitively, the ClaimChecker has to check that the application code and the Claim are compliant: all services that B has are declared in Provides_B and all services that B
invokes are declared in Calls of \( B \). This can be done only if the ClaimChecker component has access to the installer, which can process the received CAP file. Due to the lack of space we do not provide a specification of the ClaimChecker component (and the corresponding algorithm for CAP file parsing) in the current paper, the interested reader can find the details in Gadyatskaya et al. (2011, 2012).

We will denote contract and policy objects in the internal format on the card as ContractInt and Policy correspondingly. Internal format is needed for optimisation of the memory usage. AIDs are space-consuming objects (each can occupy up to 16 bytes), so we need to avoid multiple repetitions of the same AID. We now elaborate more on the details of the received contract object (denoted ContractExt or just Contract). For an application \( A \) Contract\(_A\) is a byte array with the contents structured accordingly to the following rules.

- **Provided services.** A service is required to be listed in the Provides set if it is a method of an interface extending Shareable. A service is listed in Provides array as a pair \((i, t)\), where \( i \) is the export file token for shareable interface and \( t \) is the export file token for the method (1 byte each). We note that for the Provides\(_A\) set the AID of the application \( A \) is listed in the Header component of the CAP file.

- **Called and functionally necessary services.** An application provider should list a service (belonging to another package) in the Calls set, if an invocation of this service is present in the code of the applet. A service from a package with AID \( XXX \) is listed in the contract as \((XXX, i, t, \text{funcrules.tag})\), where \text{funcrules.tag} tags if this service is also functionally necessary or not. For optimisation purposes, the Calls set is then restructured to separate services provided by different servers.

- **Authorisation rules.** An authorisation rule is listed in the sec.rules set as a pair containing the service details (defined as a provided service) and the authorised client package AID. Thus the structure is the same as for a called service, with a difference that no tag for functionality is needed: \((AID, i, t)\). Then the same optimisation strategy as for called services is applied.

### 5.2 The policy checker component

We will further use the notation sec.rules\(_A\)(\( S \)), where \( S \) is a set of services provided by \( A \), to denote a subset of sec.rules\(_A\) which defines authorisations for access to services from the set \( S \).

The PolicyChecker implements a contract-policy matching algorithm for certification of evolution introduced by an applet \( B \), that returns True iff the conditions below are true for all applications \( A \in \mathcal{A} \) on the platform (considering the possible evolutions \( P_{2a...d} \)):

1. Installation of a new applet \( B \) on the platform.
   
   a. \( B \in \text{sec.rules}_A(\text{Provides}_A \cap \text{Calls}_B) \)
   
   b. \( \text{funcrules}_B \subseteq \bigcup_{A \in \mathcal{A}} \text{Provides}_A \)
   
   c. \( A \in \text{sec.rules}_B(\text{Provides}_B \cap \text{Calls}_A) \).
2 Removal of already installed application $B \in A$.
   a $\text{Provides}_B \cap \left\{ \bigcup_{A \in A} \text{func.rules}_A \right\} = \emptyset$.

3 An update of some application $B \in A$.
   a addition of a service $s$ to $\text{Provided}_B$: $A \in \text{sec.rules}_B(s \cap \text{Calls}_A)$
   b removal of a service $s$ from $\text{Provided}_B$: $s \notin \bigcup_{A \in A} \text{func.rules}_A$
   c addition of a service $s$ to $\text{Calls}_B$: $B \in \text{sec.rules}_A(\text{Provided}_A \cap s)$
   d removal of a service $s$ from $\text{Calls}_B$: return true
   e addition of an authorisation rule for some application $C$ to access a service $s$ of $B$ to $\text{sec.rules}_B$: return true
   f removal of an authorisation rule for some application $C$ to access a service $s$ of $B$ from $\text{sec.rules}_B$: $s \notin \text{Calls}_C \text{ OR } s \notin \text{Provided}_B$
   g addition of a service $s$ to $\text{func.rules}_B$: $s \in \bigcup_{A \in A} \text{Provided}_A$
   h Removal of a service $s$ from $\text{func.rules}_B$: return true.

The PolicyChecker rejects all updates if they do not comply with these checks. Notice that these checks allow to maintain security across possible changes on the platform, and the PolicyChecker prohibits changes that break security. This strategy may introduce conflicts among installed applications. For example, it might happen, that an application $A$ cannot be removed or updated, because some other application $B$ relies on it. We envisage that an industrial implementation of the framework will comprise a conflict resolution process, that will operate on card to ensure the application owners are satisfied. In the above scenario of the conflict between $A$ and $B$ a possible solution could be to execute the removal of $A$, while also locking $B$. However, resolution of conflicts is more a matter of business agreements rather than security considerations. Therefore, the proof-of-concept research prototype does not comprise it.

5.2.1 Data structures

Smart cards are resource-constrained devices, thus it is necessary to optimise the algorithms and the data structures as much as possible. To comply with this requirement we introduce the data structures for the security policy of the card and auxiliary data, and present the algorithms for the PolicyChecker component based on these data structures. As our main goal is to speed up the computations and to reduce the amount of memory needed, we will use the bit vectors for the internal policy data structures. 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 two to three applets. Each applet can contain at most eight services. Thus the set of provided services can be written using just one byte per applet. If the system is bigger than eight loaded packages or some applet wants to provide more than eight services, the model can be scaled at
run-time (i.e., if some applications provide from 9 to 16 services then instead of using one byte per applet for service set representation we will use two bytes, three bytes for 17 to 24 services, and so on). Our representation also eases updates of the system. If some applet is deleted we will not need to restructure the policy. Further we provide details of mapping for AIDs and services into our internal representation.

5.2.1.1 Mapping Object

All the information for the mapping between delivered contracts and on-card structures is stored in a Java object Mapping on the card that consists of the following structures: Applications, Services, WishListServices and MayCallApplets objects. 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 (up to 16 bytes each), so for 8 applications on the card Applications will require up to 136 bytes.

For every application we define a mapping table for its services. For every service of 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 Mapping.Services. Services is a fixed size Java array of byte arrays. For every application from Applications there is an array of services it provides. Therefore, an element of the Services object is structured as follows:

<table>
<thead>
<tr>
<th>Method ID: token</th>
<th>Internal Method ID: byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The structure of the Services object is illustrated in Table 1. The total size of Services 128 bytes.

If during the installation of an applet there are services mentioned in Calls and not in func.rules (i.e., the services that an applet can benefit from but not requires them for its main functionality) that are not on the card, we put them in the array WishListServices, its structure is presented in Table 2. Potentially there can be any number of wished services. For instance, for eight wished services the WishListServices object will require 296 bytes.

<table>
<thead>
<tr>
<th>Application 0</th>
<th>Service 0: token</th>
<th>Service 1: token</th>
<th>...</th>
<th>Service 7: token</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Application 7</th>
<th>Service 0: token</th>
<th>Service 1: token</th>
<th>...</th>
<th>Service 7: token</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 1 The Services object on the card

<table>
<thead>
<tr>
<th>Callee package AID: byte array</th>
<th>Callee method ID: tokens</th>
<th>Caller internal ID: byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application that contains</td>
<td>Application on the card</td>
<td></td>
</tr>
<tr>
<td>the desired method,</td>
<td>that wants to use the</td>
<td></td>
</tr>
<tr>
<td>but not on the card yet</td>
<td>method</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Table 2 The WishListServices object on the card
If the sec.rules set of an applet contains authorisations for applets not yet on the card, we put them in the array MayCallApplets, its structure is presented in Table 3. Again, there can be any number of authorisations for applications not yet on the card. For 16 such authorisations the MayCallApplets object will require up to 592 bytes.

When an application is being installed, we check the tables and update the card’s policy (if needed). If an applet (service) is removed, and there are some applications having it in sec.rules(Calls), then we update the data structures. In total, the Mapping object will require up to 1,276 bytes (under the assumption of the specified bounds on the MayCallApplets and WishListServices objects).

**Table 3** The MayCallApplets object on the card

<table>
<thead>
<tr>
<th>Caller application ID</th>
<th>Callee application ID</th>
<th>Callee method ID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 5.2.2 Contract and policy objects on the card

For every application installed on the card, the Contract 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 a specification of the security policy object Policy on the card. The ContractInt object will be incorporated into the Policy object and will be used in the checks of the PolicyChecker algorithms.

The ContractInt object has the following fields: ContractInt.Provides, ContractIntCalls, ContractInt.Secrules and ContractInt.Funcrules, correspondingly to the delivered contract structure. 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.

The ContractInt.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).

Representation of the ContractInt.Secrules is similar to the sets previously described. ContractInt.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.

#### 5.2.2.1 Transformation of the received ContractExt Object to ContractInt

Both representations consist of four objects: Provides, Calls, sec.rules and func.rules. We will show how each of them transforms when it arrives on the card. Received object Provides is an array of bytes corresponding to the tokens of the provided services. When the applet arrives on the card each service token is mapped to an internal byte number in a range from 0 to 7. Let the arrived application have an internal number a. The following algorithm transforms the initial Provides array to the internal object ContractInt.Provides:

```java
for i := 0 to 7 do
    if (Mapping.Services[a][i] != null)
        ContractInt.Provides | (1 << i);
```
Transformation of func.rules, Calls and sec.rules into internal objects can be performed in two steps. First, the card maps AIDs and service identifiers to the internal representation. Second, for every row in the initial object a corresponding bit in the internal object is set to 1. In total, an internal contract object ContractInt will require 25 bytes.

5.2.2.2 Policy Object Policy on the Card

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

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 (application $j$ can use 0..7 services of each application $i$):

<table>
<thead>
<tr>
<th>Caller Application 0</th>
<th>Caller Application 1</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callee Application 0</td>
<td>...</td>
<td>11000100</td>
</tr>
<tr>
<td>Callee Application 1</td>
<td>00100101</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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 (Application $i$ can call 0..7 services in each application $j$):

<table>
<thead>
<tr>
<th>Caller Application 0</th>
<th>Caller Application 1</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callee Application 0</td>
<td>Callee Application 1</td>
<td>...</td>
</tr>
<tr>
<td>Caller Application 0</td>
<td>...</td>
<td>11000100</td>
</tr>
<tr>
<td>Caller Application 1</td>
<td>00100101</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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

Totally, the Policy object will require 200 bytes. Thus, overall $S\times C$ data storage, which also includes the Mapping object described in Section 5.2.1, will require 1,476 bytes of memory. Considering that a smart card memory typically varies between 64 and 128 kb (Dragoni et al., 2011b), the amount of memory space required for the $S\times C$ data storage is reasonably small and acceptable for most cards.

6 Algorithms of the PolicyChecker component

We proceed as follows. For the intuitive rules defined for the PolicyChecker in Section 5.2 we provide listings of the PolicyChecker algorithms modified for the proposed on-card models. Due to the lack of space we discuss only the scenario of loading of a new application as the most representative one.
Let some installed application $A$ have a mapping number $a$ and new application $B$ have received a mapping number $b$. We remind that for the PolicyChecker algorithms we use now Policy and ContractInt$_B$ objects. Let us also define $iCheck$ as a boolean variable such that the algorithm returns True if $iCheck == 1$ and False otherwise, and $iCheck$ is initialised to value 1.

- **Check**: $B \in \text{sec.rules}_A(\text{Provides}_A \cap \text{Calls}_B)$ The procedure must check that for every application $A$ on the card that 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 $S_1 \not\subseteq S_2$ then return False.

The procedure can be implemented as follows:

```csharp
for a := 0 to 7 do
  then iCheck := 0;
```

- **Check**: $\text{func.rules}_B \subseteq \bigcup_{A} \text{Provides}_A$. The procedure that checks the condition can be implemented as follows:

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

- **Check**: $A \in \text{sec.rules}_B(\text{Provides}_B \cap \text{Calls}_A)$. This check is similar to the check $B \in \text{sec.rules}_A(\text{Provides}_A \cap \text{Calls}_B)$ and can be implemented accordingly.

### 6.1 New application

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

1. Make a copy of the objects Policy 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 $0..M$.
   a) if there are services mentioned in ContractExt.Calls$_B$ that are not on the card, update the WishListServices object
if there are applications mentioned in ContractExt.SecrulesB 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 the Policy and Mapping objects 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.

Thus, the data structures we have presented are very compact and they enable efficient contract-policy matching procedures for level L0.

7 Discussion and related work

We can now discuss the relevant literature that studies the problems of applet interactions on smart cards. Ghindici and Simplot-Ryl (2008) proposed a domain specific language for information flow policies targeting small embedded systems. In the framework they propose each application is certified at loading time, having an information flow signature assigned to each method. The information flow policies enforced capture the interactions between applications on the platform and are able to detect sensitive data leaks. The framework requires a modified loader to be implemented on the embedded system, thus it is not possible to apply it without major modifications to the Java Card 2.2.2 runtime environment.

Huisman et al. (2004) present a formal framework and a tool set for compositional verification of application interactions on a multi-application smart card. Their method is based on construction of maximal applets, w.r.t structural safety properties, simulating all the applets respecting these properties. Model checking techniques are then used to check whether a composition of two applets A and B respects some behavioural safety property. Avvenuti et al. (2009) propose a tool for off-card verification of Java bytecode files, that might be later installed on the platform. Their method explores the multi-level security policy model and the theory of abstract interpretation.

Girard (1999) suggests to associate security levels (clearances) to application attributes and methods, using the traditional Bell/La Padula model. Bieber et al. (2002) adopt this approach and propose a technique based on model checking for verification of actual information flows. The same approach is used by Schellhorn et al. (2000) for their formal security model for operating systems of multi-application smart cards.

The S×C approach improves these approaches by addressing the problems related to the dynamic evolution of the applications and policies on the card. Moreover, suggested contract and policy models allow to perform the matching on the card, while the mentioned approaches are based on complicated logic reasoning and model checking, all requiring off-card verification.

Outside of the smart cards domain, the techniques for load time policy enforcement in multi-application environment were investigated also for mobile platforms and
operation systems. Ongtang et al. (2009) have proposed the Saint framework for Android mobile platform applications to impose their requirements on the usage of their services on other applications during installation time and run-time. Applications on a Saint-enabled Android platform can define their own permissions and demand the fulfillment of certain requirements by both their callers and callees. The Kirin framework was developed for Android by Enck et al. (2009) and it can check the permissions application requests at the installation time in order to capture possibly dangerous combinations of permissions and suggest a user not to install potentially dangerous software. Both Kirin and Saint did not enhance the $S\times C$ solution provided for mobile platforms by Desmet et al. (2008) and Dragoni et al. (2007). However their advantage is the simplicity of suggested policy semantics, which makes it easier for application developers and users to explore these solutions. The approach we propose in this paper is also rather simple, and it takes into account not only allowed service usage, but also the necessity of some services for application functionality. We also tried to take into account the strict computational limitations of our platform.

Fontaine et al. (2011a, 2011b) investigate load time validation of rich control flow and information flow policies for open multi-application smart cards. These proposals demonstrate that developments of the $S\times C$ framework on levels higher than 0 are practical. However, further enhancements of the framework must overcome the limitation of the solutions presented in Fontaine et al. (2011a, 2011b): the limit of the number of the authorised stakeholders mentioned in the policy. Current version of the $S\times C$ framework on the level L0 enables authorisations for unlimited (possibly yet unknown) applets, and this is one of the key benefits of the $S\times C$ technique.

7.1 Limitations of the $S\times C$ level L0

Intuitively, the key limitation of the first level of abstraction is that it does not capture the actual information exchange among the applications (which services the application will actually invoke and in which order), but only the possible direct information exchange, dealing only with the direct service invocations in the bytecode. Thus the level L0 provides an enhancement over the current Java Card services access control hardwired into the applets’ code, but it lacks ability to detect collusions of applications or actual information flow among applications.

8 Conclusions

In this article we have discussed the $S\times C$ framework as a possible security model for open multi-application smart cards. In a nutshell, an application comes with a specification of its security behaviour that must be compliant with the security policy of the hosting smart card platform. In particular, we have shown how the $S\times C$ approach can be used to address two challenges of smart card security, namely (1) preventing illegal information exchange among applications on a single smart card, and (2) dealing with dynamic changes in both contracts and platform policy. The framework has been specified at the first level of the proposed hierarchy of application models for smart cards.
Since the initial submission of the current article we have further developed the \(S \times C\) technique for smart cards with three implementations of an on-card prototype. These implementations have provided a valuable insight into development and deployment of a working solution that could be used in practice. The interested reader can refer to Dragoni et al. (2011b) and Gadyatskaya et al. (2010) for details of applet-based implementation of the PolicyChecker component, and to Gadyatskaya et al. (2011) and Gadyatskaya et al. (2012) for details of the full proof-of-concept implementation of the \(S \times C\) framework on the level L0. Future work will focus on two key issues:

1. developing all the levels of abstraction (starting from the limitations of the level L0)
2. extending the security relevant actions of both contracts and policies in order to consider rules restricting the use of resource APIs (an interesting task targeting Java Card 3.0 platforms).

**Acknowledgements**

This work was partially supported by the EU under Grants EU-FP7-FET-IP-SecureChange, EU-FP7-FET-IP-Aniketos and EU-FP7-IST-NoE-Nessos. A preliminary proposal of the security-by-contract hierarchy of contract and policy expressivity for open multi-application smart cards appeared in Dragoni et al. (2010, 2011a). The authors would like to thank the anonymous reviewers for their insightful comments.

**References**


