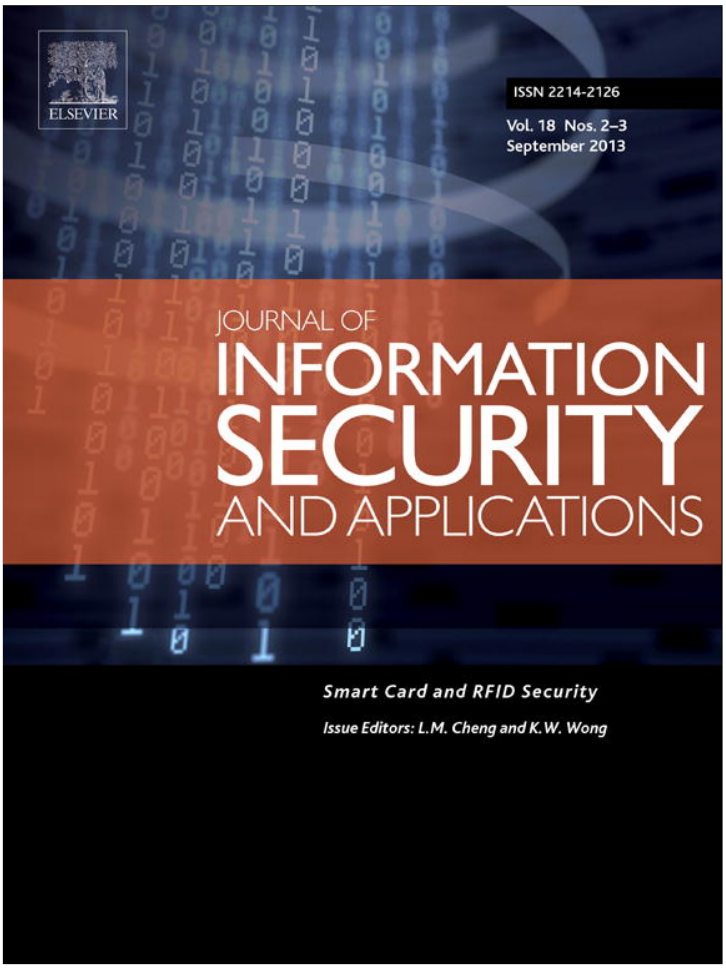


JOURNAL OF INFORMATION SECURITY AND APPLICATIONS 18 (2013) 108–129

Available online at www.sciencedirect.com

SciVerse ScienceDirect

journal homepage: www.elsevier.com/locate/jisa

Load time code validation for mobile phone Java Cards

Olga Gadyatskaya [a,*], *Fabio Massacci* [a], *Quang-Huy Nguyen* [b], *Boutheina Chetali* [b]

[a] *Department of Information Engineering and Computer Science, University of Trento, via Sommarive 14, 38123 Trento, Italy*

[b] *Trusted Labs, rue du Bailliage 5, 78000 Versailles, France*

Keywords:
Load time application validation
Secure elements
Security-by-Contract
Java Card

ABSTRACT

Over-the-air (OTA) application installation and updates have become a common experience for many end-users of mobile phones. In contrast, OTA updates for applications on the secure elements (such as smart cards) are still hindered by the challenging hardware and certification requirements.

The paper describes a security framework for Java Card-based secure element applications. Each application can declare a set of services it provides, a set of services it wishes to call, and its own security policy. An on-card checker verifies compliance and enforces the policy; thus an off-card validation of the application is no longer required.

The framework has been optimized in order to be integrated with the run-time environment embedded into a concrete card. This integration has been tried and tested by a smart card manufacturer. In this paper we present the architecture of the framework and provide the implementation footprint which demonstrates that our solution fits on a real secure element. We also report the intricacies of integrating a research prototype with a real Java Card platform.

1. Introduction

Smart handsets are providing increasingly sensitive services (e.g. finance, access) that are often updated over the air (OTA) in a dynamic fashion. The deployment of Java-enabled (U)SIM cards, which use the GlobalPlatform[1] card technology, have further enabled OTA application downloads for 3G and GSM mobile networks (some hundred millions (U)SIM cards utilize the GlobalPlatform infrastructure). For security reasons, financial or similarly sensitive services are usually hosted together by a *secure element*, such as a smart card (Langer and Oyrer).

A common assumption is that only few and limited applications will be loaded on the secure element, but this is no longer the case: the trusted elements quickly evolve into *multi-tenant* platforms following the trends of smartphone markets (Bouffard et al., 2011b). For example, leading smart card manufacturers, such as Gemalto, Oberthur and Giesecke&Devrient, already offer Facebook or Twitter applications to be loaded onto the (U)SIM card, and a healthcare application[2] was recently proposed.

From a security perspective it is important that the applications are confined (the Java Card firewall does precisely that),

* *Corresponding author*. Tel.: +39 0461 283828; fax: +39 0461 282093.
E-mail addresses: olga.gadyatskaya@unitn.it, gadyatskaya@disi.unitn.it (O. Gadyatskaya), fabio.massacci@unitn.it (F. Massacci), quang-huy.nguyen@trusted-labs.com (Q.-H. Nguyen), boutheina.chetali@trusted-labs.com (B. Chetali).

[1] GlobalPlatform™ is a standard set of specifications for card contents management (GlobalPlatform Inc., 2011).

[2] http://medicmobile.org/2011/06/06/medic-mobile-announces-the-first-mobile-sim-app-for-healthcare. Accessed on the web in Jan. 2013.

2214-2126/$ – see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.jisa.2013.07.004

but from a business perspective we would like them to talk to each other within the secure element: when German transit authorities launch a Near-Field Communication (NFC)-based ticketing service[3] and VISA pushes its payment SIM application payWave,[4] they may want to collaborate. Therefore, control of interactions among applications is a crucial requirement for the overall protection guaranteed by the secure element.

In order to allow interactions across the firewall, Java Card (JC) applications interact through Shareable interfaces (Classic Edition, 2011). A Shareable interface method (or *service* for short) is just a Java method that can be called through the firewall. The traditional solution to restrict access to a service on Java Card is to embed access control checks in the service code. In this case the only way to add or remove possible callers from the code is to re-install the application (this is how the code updates are implemented on Java Card). In many cases it is not possible to remove an application referenced by other applications on Java Card. So, even if we just want to add the possibility of being called by another application, we will need first to delete all other calling applications, then re-install the updated application, and then re-install all callers again. Therefore, on Java Card separation of access control to a service from implementation of the service provides is a desirable feature: when considering multi-tenancy, applet providers want to be able to restrict access to their services in a declarative and independent fashion.

Our alternative solution would be to validate the bytecode to be well-behaved with respect to interactions while loading on a secure element. The target of our research is to perform application validation directly on the secure element (the (U)SIM card with its severely limited resources) to ensure the following goals:

- applications can be loaded OTA;
- applications can declaratively control (allow or block) access to their shared services by other applications on the card, without mixing it with functional code;
- the access control policy can mention any applications, even if at any time we only have few of them installed;
- application bytecode should be *validated by the card itself* to respect the interaction policies of the other applications already on the card during loading time.

This must be achieved under the following constraints:

- no modification to the current application loading protocol, the JC firewall and the virtual machine (VM) implementation of the secure element;
- most part of the trusted computing base is in ROM (non-modifiable non-volatile memory);
- application providers can set-up their security policies directly without bothering the secure element owner for individual policy changes.

For mobile phones, a number of proposals for application certification at load time have been put forward in the past years, but most research proposals stop at load time checking of the application manifest and use the phone normal processor for checking (Enck et al., 2009; Ongtang et al., 2009). Other approaches propose to check interactions at run-time requiring VM/platform modifications (Bugiel et al., 2012; Enck et al., 2010), suggest to check interactions off-device (Chin et al., 2011) or advocate application rewriting (Xu et al., 2012).

So far for smart cards the combination of all elements has not been achieved, and our new contribution is to achieve it by designing a complete working solution. In the smart card world interactions among the applications can be certified, but then the card has to be locked (new applications cannot be loaded, existing ones cannot be removed). For example, the TaiwanMoney Card (Taiwan) based on the Multos technology combines a Mondex payment application with a transport application.[5] This approach of locking the card is not feasible for OTA-loaded applications. The off-device validation techniques were proposed for Java Card applications (e.g. the works by Bieber et al. (2002) and Avvenuti et al. (2012)), but they cannot work in practice for the OTA loading, because they require an independent verification authority, with whom application providers need to negotiate any single change; and full formal verification cannot be ported to the device itself because of the computational constraints.

Our contribution. Our target is to achieve the same security level, as offered by Java Card itself, while allowing the flexibility of OTA updates on a very restricted platform. We do not aim to achieve more security than the current methods of embedding access control checks in the application code or off-line bytecode validation. Our proposal allows to immediately address the OTA-loading demands for access control mechanisms in the context of application communication and service calls. The policies that our system can enforce are simple, but they are substantial given the resources available.

In this paper we report on the engineering aspects that can achieve all the goals mentioned above along with the constraints of the secure element environment: at most 10 KB of memory footprint and at most 1 KB of RAM consumed for validation. Our system is able to process applications of sizeable complexity, such as the electronic identity applet (Philippaerts et al., 2011). A further challenge that we have faced is the need to maintain confidentiality of the Java Card platform implementation. In the article we report how we had overcome this problem. We also present an abstract model of a multi-tenant secure element platform based on deployed applications and shared/invoked services and demonstrate that the validation process of our framework keeps the platform secure across the updates.

The rest of this article is structured as follows. Section 2 presents a high-level overview of our solution. The background information on the Java Card technology is given in §3; the notions of a contract and a security policy of the platform are introduced in §4. Algorithms of the framework components are discussed in §5. We demonstrate correctness of the presented solution in §6. We present the final architecture of the framework in §7 and discuss the performance (§8) and security (§9) of our solution. We overview related work (§10) and then conclude (§11).

[3] https://www.touchandtravel.de/, accessed on the web in Jan. 2013.

[4] http://www.visaeurope.com/en/cardholders/visa_paywave.aspx, accessed on the web in Jan. 2013.

[5] http://en.wikipedia.org/wiki/TaiwanMoney_Card. Accessed on the web in Jan. 2013.

2. Our approach

The threat model. The third-party application providers do not trust each other. We assume an attacker that can load applications (*applets* for short) on the secure element, remove her own applets or update the security policy of her applets. The attacker aims to gain access to sensitive services of other applet providers, which possibly are former business partners.

The platform owner is trusted by the application providers to make sure that the platform implementation is correct. However, she does not want to be involved in the costly security validation of day-by-day policy or code changes for applet providers. The responsibility of the platform owner is to make sure that the platform implementation is correct. So we assume that for any applet its development and deployment steps were correct and the bytecode respects the Java abstractions. We do not consider application spoofing in the threat model, because the means for protection against this attack are already provided by the GlobalPlatform middleware (GlobalPlatform Inc., 2011) (GlobalPlatform offers a set of primitives to implement authentication with external clients and establish encrypted communication channels).

Our solution. We propose a system to ensure secure coexistence and sharing of capabilities between multiple applications in a mobile phone multi-tenant Java Card. Our system does so through requiring access control lists for each service interface that can be verified by the system at load time. The specification of these lists is moved from functional code to a dedicated bytecode component and stored on card separately in a cumulative policy structure; so that the policy can be updated without requiring cumbersome reinstalls upon changes. The system fits within a state of the art (U)SIM card, is compliant with the standard applet deployment protocol and the Java Card Run-time Environment (JCRE).

Our framework improves the current JC security architecture by performing the application code validation upon loading. An applet aware of the new security architecture will now bring a *contract* (a component of the code stating the policy and the details of the inter-application communications the applet participates in). Contracts of deployed applets will be collected by the platform and stored as the *platform security policy* in a memory-efficient format. The contract of a new applet will be matched with the actual loaded code on the secure element and with the current security policy of the platform. If both checks are successful, the applet will be accepted and its contract will be added to the policy. Otherwise it will be rejected and removed. Fig. 1 summarizes the workflow for load time validation and the new components of our framework: the `ClaimChecker`, the `PolicyChecker` and the `PolicyStore`. Fig. 1(b) shows the position of the new components in the stack, for more details see Fig. 6.

Following the strategy to keep the platform secure after each addition and deletion of an applet (we call these changes *platform evolution*), our framework during the application removal process checks that the platform after the removal will continue to be secure. The `ClaimChecker` is not invoked in this case, because the code was already verified to be compliant with the contract. Only the `PolicyChecker` component is invoked, and it decides if the application can be removed (see §5.2 for more details). We also support a flexible application policy update. On Java Card the application code is updated by removing the current application and subsequently loading its new version, because the security policy of an applet is embedded into the functional code. The S×C approach enables a way to update the security policy of an application without reinstalling the code. The checks executed for application policy update are similar to those done for application deletion.

Our framework has been integrated with the existing JCRE components. We do not deal with applet authentication in this paper, because we rely on the GlobalPlatform authentication and delegation mechanisms, and we only focus on the access control. We do not modify the standard application deployment process, the Java Card Virtual Machine (JCVM) or the existing firewall mechanism. Our approach ensures backward compatibility: cards that are not aware of the new framework can work with applets that are aware of it, and vice-versa.

3. Background on Java Card

Our solution targets devices in the field, thus we have developed it for Java Card 2.2.2 (the previous stable generation (SUN Microsystems, 2006)) and Java Card 3.0.4 (the latest specification of the Classic edition (Classic Edition, 2011)), that is fully backward-compatible with Java Card 2.2.2. The alternative version of Java Card is 3.0.1 Connected edition, that supports more standard Java features, such as servlets and service factories. However, to the best of our knowledge this version is not yet adopted by the industry. This claim can be supported by the fact, that, while the Java Card 3.0 appeared in two editions in 2008 (SUN Microsystems, 2008), only the Classic edition is regularly updated by Oracle (the latest version 3.0.4 dated 2011 (Classic Edition, 2011)), while the Connected edition is frozen at the version 3.0.1 dated 2009 (SUN Microsystems, 2009).

There are not so many novel features of the Java Card platform that are available in the last specification 3.0.4, but were not available in the Java Card 2.2.2. The new features are: guaranteed integer support, the latest cryptographic algorithms (4096-bit RSA), alignment with the latest contactless protocol standards and garbage collector; but these features are irrelevant for the scope of this paper.

Fig. 2 summarizes the main components of the platform and the steps of applet development and deployment. The JCRE comprises the JCVM, a set of the Java Card API, the Installer and the Loader (Classic Edition, 2011). The standard implementation of the JCRE includes components implemented in Java Card (the Java Card Interface in Fig. 2) and components implemented in C (the Native Interface in Fig. 2). Calls from the JC components to the native components are processed without hinderance; calls from the native components to the JC components are prohibited, and lower level primitives have to be used.

Application development and deployment. A developer writes a package in Java, then he compiles it into .class files and afterwards converts it into a CAP (Converted APplication) format. CAP files consist of several optimized components in a

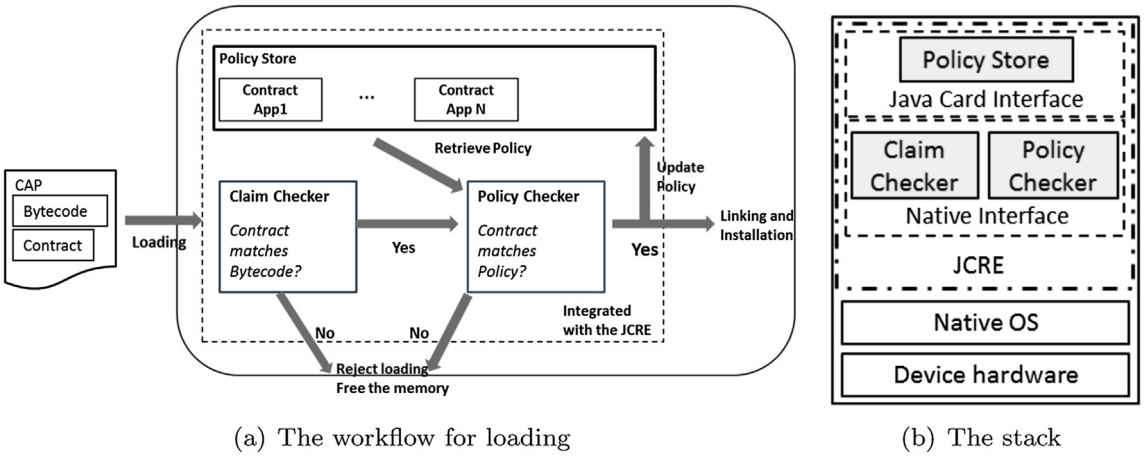


Fig. 1 – The validation workflow for loading and the Java Card stack with the new components.

predefined format in order to reduce the amount of memory needed for storing an applet; they include a single Constant Pool component, a Method component with all methods instruction sets, etc. For conversion the Export files of imported packages are required, and during conversion the Export file of the converted package is produced. Export files contain fully-qualified names and signatures of exported interfaces and methods, and are used for interoperability purposes.

A package can contain one or multiple applets; an applet is a class extending `javacard.framework.Applet`. A *library* package does not contain any applets. Libraries cannot be remotely invoked from a terminal or executed. For simplicity we will consider that each package contains exactly one applet and we will use words *package*, *application* and *applet* interchangeably, except for when explicitly stated otherwise.

The deployment includes the following steps. Upon receiving a CAP file, the Installer uses the Loader API to process the file and perform some checks specified in Classic Edition (2011). Upon finalization of the linking process an applet instance can be created. When the applet is no longer wanted, the Installer, upon performing the necessary checks, can remove the applet instance and the CAP file from the memory (the *removal process*).

Java Card packages and applets are uniquely identified by their AID (Applet IDentifier) assigned by the ISO/IEC 7816-5 standard. An AID is a long byte array and the CAP file structure is optimized to avoid multiple repetitions of the same AID. The AID of an imported package is listed only once in the Import component of the CAP file and a 1 byte identifier (a tag) of this package is used in the CAP file. On the card the loaded packages are further referred to by their *local identifiers* assigned by the JCRE, which maintains the AID – local identifier correspondence. We will denote the AID of package A as AID_A.

Application interactions. Applets from different packages are isolated by the JCRE firewall. The firewall confines each applet's actions to the applet's *context*. Each JC package (a CAP file) has its own context, so objects can communicate freely within the same package. For this reason we can consider that each package contains one applet, as it is not possible to mediate the communications within a package. Since a package is loaded in one pass, a malicious applet cannot be later added to an honest applet package. However, a malicious applet can arrive in another package.

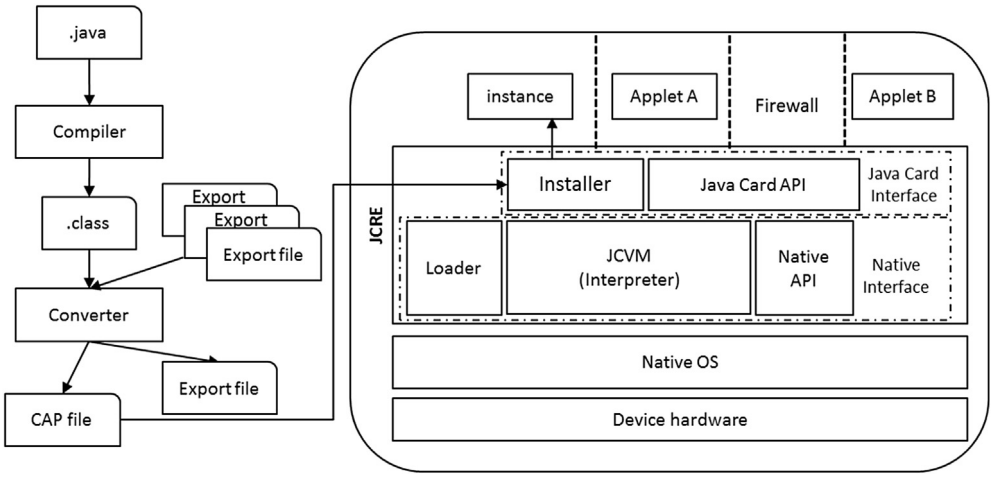


Fig. 2 – The traditional Java Card architecture, applet development and deployment process.

The interesting part is interactions of applets from different contexts. The JCRE allows only methods of *Shareable interfaces* (the interfaces extending `javacard.framework.Shareable`) to be accessible through the firewall. If an applet desires to share some methods, it implements a Shareable interface (SI). This applet is called a *server* and the shared methods are called *services*. An applet that calls a service is called a *client*.

In order to realize the applet interaction scenario the client has necessarily to import the Shareable interface of the server and to obtain the *Export file* of the server, which lists shared interfaces and services and contains their *token identifiers*. The server Export file is necessary for conversion of the client package into a CAP file. In a CAP file all methods are referred to by their *token identifiers*, thus during conversion from class files into a CAP file the client needs to know correct token identifiers for services it invokes from other applets. As Shareable interfaces and Export files do not contain any implementation, it is safe to distribute them. The Export files consumption for conversion is presented schematically in Fig. 2.

The current JC security mechanism to enforce access control for service invocations is the context control JC API. A server applet can check who is calling upon receiving a request for the shared object or using the `getPreviousContextAID()` API in the service code. We illustrate this in the following motivating example.

3.1. A motivating example

We consider two applets installed on a secure element. `Purse` is a payment applet (e.g. payWave) and `Transport` is a ticketing applet (e.g. the DBahn NFC-based ticketing applet). The public transportation system provides gate terminals that can communicate to the `Transport` applet and check if the ticket was paid. The ticket can be paid by the device holder through specific payment terminals. If the `Purse` applet allows to share its payment service with the `Transport` applet, then the tickets can be purchased through `Purse`, and the device holder does not need to wait in the line to payment terminals, as the ticketing process can be seamlessly executed by the gate terminals.

Fig. 3 contains a sanitized code snippet from the `Purse` applet (we stripped off the details for the sake of clarity, the functionality of the payment service of the actual applet is different). The `Purse` applet has a service `payment()` provided in Shareable `PaymentInterface`. Access control for this service is implemented as the context control API usage upon actual service execution (method `JCSystem.getPreviousContextAID()`, line 23 in Fig. 3) and the requesting client AID check upon provision of the object implementing the SI (line 32 in Fig. 3).

The access control list (ACL) `clientAIDs[]` is currently hard-coded within the `Purse` code (line 6 in Fig. 3) and it can be updated only if `Purse` is reinstalled. If the `Purse` provider does not want to reinstall the applet any time the ACL is updated, she might choose not to implement the service access control at all. Unfortunately, in this set-up any other applet on the card that knows the `appletAID` of `Purse` can invoke it. For instance, the device holder can further load a new application `MessagingApp`, which provides him access to the common

```
01 byte ClientsNumber = 1;
02 byte [] TransportAIDset =
{0x01,0x02,0x03,0x04,0x05,0x0C,0x0A};
03 final AID TransportAID = JCSystem.lookupAID
(TransportAIDset,(short)0,(byte)TransportAIDset.length);
04
05 //the access control list
06 AID [] clientAIDs = {TransportAID};
07 //ACL check implementation
08 public short authorizedClient(AID clientAID){
09    for (short i=0; i<ClientsNumber; i++)
10       if (clientAIDs[i].equals(clientAID))
11          return i; //clientAIDs is in the ACL
12    return -1;
13 }
14 //SI definition
15 public interface PaymentInterface extends Shareable {
16    //definition of the payment service
17   byte payment(short account_number);
18 }
19 public class PaymentClass implements PaymentInterface {
20   byte payment_code = 0x08;
21   public byte payment(short account_number){
22      //implementation of  the service
23      AID clientAID = JCSystem.getPreviousContextAID();
24      if (authorizedClient(clientAID) == -1) //ACL check
25          return (byte) 0x00; //no service is provisioned
26      else return payment_code; //provision of the service
27   }
28 }
29 public PaymentClass PaymentObject;
30
31 public Shareable getShareableInterfaceObject(AID oAid,
byte bArg){
32  if (authorizedClient(oAid) != -1) // ACL check
33     if (bArg == InterfaceDetails)
34        //provision of the SI object
35        return (Shareable) (PaymentObject);
36     else
37        ISOException.throwIt(ISO7816.SW_WRONG_DATA);
38  return null; //nothing is provisioned
39 }
```

Fig. 3 – A sanitized snippet of the `Purse` applet. It contains the ACL defined in the code, and definition, implementation and provision of the payment service.

social network websites. This new applet may try to access the sensitive `payment()` method of `Purse`, and if no controls were implemented, execute the payment process. However, as the payment service is sensitive, `Purse` has to use the cumbersome embedded access control checks.

We argue that the context control API usage is not flexible, as the list of the authorized clients is embedded in the code of the server applet. Our framework gives the `Purse` applet the possibility to redefine the ACL with the allowed clients for the `payment()` service without reinstallation.

4. Contracts

High-level description. A *provided service* s can be identified as a tuple $\langle \mathrm{AID}_s, t_I, t_m \rangle$, where AID_s is the unique AID of the package that provides the service s, t_I is the token identifier for the Shareable interface where the service is defined, and t_m is the token identifier for the method s in the Shareable interface. A *called service* can be identified as a tuple $\langle \mathrm{AID}_B, t_I{}^B, t_m{}^B \rangle$, where AID_B is the AID of the package providing the called service. More details on the called service identification are given in Sec. 5.1.

The services provided and called by the applet A are listed into the *application claim*, denoted $\mathtt{AppClaim}_A$. We denote the

```
export_classes {
  class_info {     // packagePurse/PaymentInterface
    token     0  // Shareable interface token
    …
    name_index   3  // packagePurse/PaymentInterface
    …
    export_methods_count  1
    methods {
      method_info {
        token  0   // shared method token
        …
        name_index  0  // payment
```

Fig. 4 – Shareable interface and method token identifiers of the `Purse`'s payment service in the Export file.

provided services set of application A as $\texttt{Provides}_A$ and the called services set as $\texttt{Calls}_A$.

The *application policy*, denoted `AppPolicy`, contains two parts: `sec.rules` and `func.rules`. For the applet A $\texttt{sec.rules}_A$ is a set of authorizations for access to the services provided by A. A security rule is a tuple $\langle \text{AID}_B, t_I, t_m \rangle$, where AID_B is the AID of the package B that is authorized to access the provided service with the interface token identifier t_I and the method token identifier t_m. In other words, $\langle \text{AID}_A, t_I, t_m \rangle$ is a service provided by A.

$\texttt{func.rules}_A$ is a set of *functionally necessary services* for A, we consider that without these services provided on the platform A cannot be functional (so there is no point to load it). Functionally necessary services can be identified in the same way as called services, moreover, we insist that $\texttt{func.rules}_A \subseteq \texttt{Calls}_A$. We do not allow to declare arbitrary services as necessary, but only the ones that are at least potentially invoked in the code.

The application claim and policy compose the *application contract*, denoted `Contract`. Contracts are delivered on the card within Custom components of the CAP files.

Definition 4.1. *For an applet A $\texttt{Contract}_A$ is a tuple $\langle \text{AppClaim}_A, \text{AppPolicy}_A \rangle$, where $\texttt{AppClaim}_A$ is a tuple $\langle \text{Provides}_A, \text{Calls}_A \rangle$ and $\texttt{AppPolicy}_A$ is a tuple $\langle \text{sec.rules}_A, \text{func.rules}_A \rangle$.*

Contract realization. Token identifiers 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 a service $\langle \text{AID}_A, t_I, t_m \rangle$ provided by A, the token identifier t_I is listed in the `class_info.token` structure of the corresponding interface declaration in the A's Export file, and the token identifier t_m is listed in the corresponding `method_info.token`. Fig. 4 presents an excerpt from the Export file of the `Purse` applet with the token identifiers of the Shareable interface and the method of the payment service. The Export file is consumed by the `Transport` applet during conversion in order to replace the fully-qualified names with the corresponding token identifiers.

In Table 1 we provide examples of contracts of the `Purse` and `Transport` applets in the standard Java fully-qualified names and in the token identifiers notation. `Contract` can be embedded within the Custom component of the CAP file using the CAP modifier tool we have developed, in this way it is delivered on board following the standard CAP file loading protocol.

The choice to use Custom components is motivated by the fact that CAP files carrying Custom components can be recognized by any JC Installer, as the JC specification requires. To write contracts we use structures and naming that are similar to the ones defined for CAP files (Classic Edition, 2011). After applying the standard JC tools (Compiler and Converter), we modify the converted CAP file by appending the Contract Custom component and modifying the contents of the Directory component (by increasing the counter of the Custom components amount and specifying the length of the Contract Custom component), so that the Installer can recognize that the CAP file contains a Custom component. Note that this part does not need to be trusted: whatever errors will be introduced in this part will simply mean that the applet is rejected by our framework. More details of the tool can be found in §7.

5. The components' algorithms

5.1. The `ClaimChecker` algorithm

High-level description. The `ClaimChecker` is the component that parses the CAP file and matches the contract with the CAP file bytecode. The algorithm starts by retrieving the contract from the Custom component, then it executes the check on the provided services. The Export file of the package contains the

Table 1 – Contracts of `Purse` and `Transport` applets.

Contract structure	Fully-qualified names	Token identifiers
`Purse`		
`Provides`	`PaymentInterface.payment()`	⟨0, 0⟩
`Calls`		
`sec.rules`	`Transport` is authorized to call `PaymentInterface.payment()`	⟨0x01020304050C, 0, 0⟩
`func.rules`		
`Transport`		
`Provides`		
`Calls`	`Purse.PaymentInterface.payment()`	⟨0x01020304050B, 0, 0⟩
`sec.rules`		
`func.rules`		

explicit token identifiers of each Shareable interface and its methods. However, as the Export file is not delivered on the card, the `ClaimChecker` algorithm relies on the CAP file itself and extracts the necessary token identifiers from the Export and the Descriptor components. In the Descriptor component the algorithm retrieves all the interfaces defined in this package, for each interface it checks within the Export component, whether this interface is marked as Shareable. If this is the case, the algorithm retrieves the method token identifiers for this interface in the Descriptor component interface entry and verifies that the pair <interface token, method token> is present in the `Provides` set. After processing all the interface entries in the Descriptor component, the algorithm ensures that all services declared in the `Provides` set were found.

For the called services the algorithm starts again from the beginning of the Descriptor component. It retrieves the Method component offset to the beginning of each method of the current package and stores the offset in the temporary buffer. We note that in case there are too many methods in the CAP file, the algorithm processes them in batches, to ensure that the limited temporary buffer is not exceeded. Then the algorithm accesses each method of the package in the Method component with these offsets and checks that the invoked services are all declared in the `Calls` set of the contract. Afterwards, the algorithm ensures that all called services declared in the `Calls` set were found. Algorithm 5.1 contains a short English description of the operations done with the CAP file components. We demonstrate correctness of this approach in Sec. 6.

The process of the called service identification is illustrated in Fig. 5, it presents a (sanitized) source code snippet of the `Transport` applet and the corresponding excerpts from the CAP file. `Transport` invokes the payment service in line 08 of the code snippet (the `Transport` source code is explained in §3). This invocation corresponds in the bytecode to the instruction invokeinterface ⟨2160⟩, which is resolved in the CAP file to invocation of the service ⟨0x01020304050B, 0, 0⟩, that is the `Purse`'s payment service.

To implement the `ClaimChecker` we needed a subset of the Loader API to access the beginning and the length of CAP components. This subset (`CAPlibrary`) also contained some of the data structures and constants available on the device and some additional functions necessary to access the data from the CAP file that was stripped off during loading and stored separately. An example of a function available in the `CAPlibrary` is the function serving the AID of the loaded package, because it was stored in the card registry together with the assigned local identifier, and is no longer available in the CAP file. The `ClaimChecker` algorithm uses variable-length temporary buffers, that do not exist on a smart card. The actual implementation explores just one 256 byte length temporary buffer. The academic partner followed the JC specifications for re-implementing the `CAPlibrary` for testing purposes and directing the prototype.

Sharing the `CAPlibrary` with the minimal Loader API among the smart card manufacturer and the academic partners was our solution to the platform confidentiality problem. Without this

Require: A CAP file.
Ensure: `True`/`False`, Contract.
1: **Custom Component:** get Contract;
2: // Start with the provided services
3: **Descriptor Component:** go through the interfaces and the interface methods;
4: **Export Component:** get tokens of Shareable interfaces;
5: check for match with the provided services in the contract;
6: //Proceed to the called services
7: **Import Component:** get package AIDs of imported packages and their indices;
8: for each AID check it is declared in the Contract;
9: **Descriptor Component:** go through the classes and obtain the offset of each method, store it in the temporary buffer;
10: **Method Component:** for each stored method offset parse the bytecode to identify called services;
11: **if** a service invocation is found **then**
12: **Constant Pool Component:** check the called service (AID, interface token, method token) to be present in the Calls set;
13: **Header Component:** get the current package AID;
14: **The Final Check:** return `True` iff the collected sets match with the Contract
15: **if** *Not Match* **then return** `False`
16: **else return** {`True`, *current package AID*, *Contract*}

Algorithm 5.1 – The `ClaimChecker` algorithm English description.

Engineering aspects. The JCRE imposes some restrictions on method invocations in the applet code. Only the opcode `invokeinterface` in the code allows to switch the context to another application. Thus, in order to collect all potential service invocations we analyze the bytecode and infer from the `invokeinterface` instructions possible services to be called. During execution the JCVM expects three operands $\langle \text{nargs}, \text{id}_{\text{CP}}, \text{t}_{\text{m}} \rangle$ with this instruction and an object reference `ObjRef` on the stack. There `nargs` contains number of arguments of the invoked method (plus 1); id_{CP} is an index into the Constant Pool of the current package, the Constant Pool item at id_{CP} index should be a reference to the interface type `CONSTANT_Classref`; t_{m} is the interface method token of the method to be invoked and `ObjectRef` is the reference to the object to be invoked. The id_{CP} index in the Constant Pool component is used to identify the AID of the called package from the Import component.

set of API our implementation could not be integrated with a real card. In the same time, one of the main concerns of the smart card manufacturer partner was minimization of disclosure of the proprietary implementation details.

5.2. *The* `PolicyChecker` *component*

The `PolicyChecker` component executes contract-policy compliance checks. It needs to retrieve the security policy of the card from the `PolicyStore` and the loaded contract from the `ClaimChecker`. The contract is then converted into the internal on-card format. Intuitively, during loading of applet *B* the `PolicyChecker` has to check that (1) for all the services from $\texttt{Calls}_B$, *B* is authorized by their providers to call them; (2) for all services from $\texttt{Provides}_B$ all applets that can invoke these services are authorized by *B*; (3) all the services from $\texttt{func.rules}_B$ are provided.

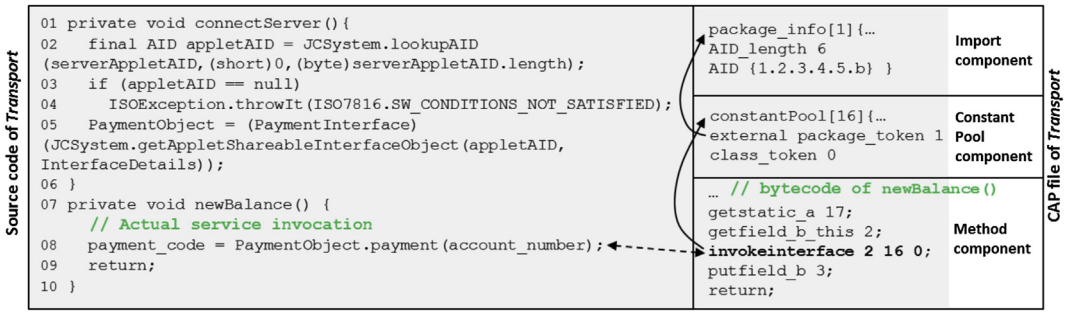


Fig. 5 – AID, interface and method token identifiers of the invoked payment service in the CAP file of the `Transport` applet.

Algorithm 5.2 specifies the policy checks to be executed for each type of change on the platform. It rejects all updates (returns `False`) if they do not comply with these checks. Notice, that the `PolicyChecker` component executes only the checks for deployment of a new applet and removal of an existing one (lines 2–8). In case of an application policy update the `PolicyStore` component handles the check (lines 9–18).

For applet A we denote as $\texttt{shareable}_A \subset \{\text{AID}_A\} \times \mathbb{N} \times \mathbb{N}$ the set of services provided by this applet. In practice, for a package A we define $\texttt{shareable}_A$ as a set of meaningful shared services. Namely, for each service $s = \langle \text{AID}_A, t_I, t_m \rangle$ such that t_I = `export_file.export_classes[i].token` and t_m = `export_file.export_classes[i].-methods[j].token` s belongs to $\texttt{shareable}_A$ (the structures in this definition

Require: Specification of the update on the platform *UpdateSpecification*, platform policy.
Ensure: `True` if the update is compliant with the policy; `False` otherwise.
1: **switch** *UpdateSpecification* **do**
2: **case** Deployment of a new package B
3: **for** all deployed applets $A \in \Lambda$ **do**
4: **if** $B \notin \text{sec.rules}_A(\text{Provides}_A \cap \text{Calls}_B)$ **then return** `False`;
5: **if** $A \notin \text{sec.rules}_B(\text{Provides}_B \cap \text{Calls}_A)$ **then return** `False`;
6: **if** $\text{func.rules}_B \not\subseteq \bigcup_{A \in \Lambda} \text{Provides}_A$ **then return** `False`;
return `True`;
7: **case** Removal of already deployed package B
8: **if** $\text{Provides}_B \cap \{\bigcup_{A \in \Lambda} \text{func.rules}_A\} \neq \emptyset$ **then return** `False`;
return `True`;
9: **case** Policy update for already deployed package $B \in \Lambda$
10: **switch** *PolicyUpdateSpecification* **do**
11: **case** Addition of an authorization rule for some applet C to access a service $B.s$ to sec.rules_B **return** `True`;
12: **case** Removal of an authorization rule for some application C to access a service $B.s$ from sec.rules_B
13: **if** $B.s \in \text{Calls}_C$ **then return** `False`;
14: **else return** `True`;
15: **case** Addition of a service $C.s$ to func.rules_B
16: **if** $s \notin \bigcup_{A \in \Lambda} \text{Provides}_A$ **then return** `False`;
17: **else return** `True`;
18: **case** Removal of a service $C.s$ from func.rules_B **return** `True`;

Algorithm 5.2 – The `PolicyChecker` algorithm description.

6. The correctness proof

In order to demonstrate correctness of the proposed S×C approach we define an abstract model of the JCRE, following the specification. Let Δ_Λ be a domain of package AIDs and Δ_Σ be a domain of services (identified as tuples $\langle \text{AID}_A, t_I, t_m \rangle$), where $\text{AID}_A \in \Delta_\Lambda$ is the AID of the package providing the service. A package execution is defined by the set of methods of this package. Let A be a CAP file and $\mathscr{M}_A$ be a set of methods of this CAP file. A method $m \in \mathscr{M}_A$ is defined by the set of its instructions. Let $\mathscr{B}_m$ be the set of opcodes of the method m. Let $\mathscr{B}_A = \bigcup_{m \in \mathscr{M}_A} \mathscr{B}_m$ be a bytecode of CAP file A. For the package A, we will also denote its CAP file data (specifically, the Constant Pool, Descriptor, Export and Import components) as $\texttt{ConstPool}_A$.

Definition 6.1 (Application). *On the secure element platform a deployed application A is a tuple* $\langle \text{AID}_A, \mathscr{B}_A, \text{ConstPool}_A \rangle$.

represent the contents of the Export file of the package A). If t_I or t_m are not meaningful token identifiers (there is no structure with the value t_I = `export_file.export_classes[i].token` or there is no method with the corresponding t_m token defined for this interface in the Export file), then $\langle \text{AID}_A, t_I, t_m \rangle \notin \text{shareable}_A$.

Definition 6.2. (Platform) *Platform Θ is a set Λ of currently deployed applications.*

Definition 6.3. (Platform Security Policy) *Security policy $\mathscr{P}$ of the platform consists of the contracts of all the applications $\Lambda = \{A_1, \ldots, A_n\}$ deployed on the platform:* $\mathscr{P} = \bigcup_{A_i \in \Lambda} \{\text{Contract}_{A_i}\}$

The taxonomy of the JCVM instructions. The JCVM specification v. 3.0.4 (Classic edition) defines 109 instructions, including 4 instructions that can be used to invoke methods. These are `invokeinterface`, `invokespecial`, `invokestatic`

and `invokevirtual`. The instruction `invokeinterface` is used for invocation of interface methods and it allows to invoke services across package contexts. Other method invocation instructions cannot be used for service invocations, as the firewall will allow (at most) to switch context to the JCRE context upon execution of these instructions. In Table 2 we present a taxonomy of the JCVM instructions that we will use to reason about applet execution. The taxonomy is based on the possibility of a context switch upon execution of instructions, and we cluster the method invocation instructions in a separate class of instructions.

Theorem 6.1. *In the presence of the* S×C *framework all methods invoked by any deployed application B are authorized by the platform policy, or are allowed to be invoked by the JCRE.*

Proof. The proof by contradiction goes over all possible cases of method invocation on the platform. We first assume the theorem does not hold: *B* is a deployed application and it invokes some method not authorized in the platform policy (*B* cannot invoke a method against the JCRE rules, otherwise the platform implementation is incorrect). Since *B* is a deployed application, it has been validated by the `ClaimChecker` and the `PolicyChecker`, also all executed application policy updates of *B* were validated. Possible cases are: *B* invokes its own method, *B* invokes a method of yet undeployed applet *A*, and *B* invokes a method of installed applet *A*. The first two situations are obvious. In the last situation we reason by the type of the executed instruction, discussing each possible instruction type in the taxonomy. We omit the cases I–V for brevity, the full proof is provided in the §Appendix B. Here we only discuss the case when the executed instruction is a method invocation instruction, as the most interesting.

Table 2 – The JCVM instructions taxonomy.

Type	Instructions
I	Arithmetic instructions and other instructions that do not modify executions. These are instructions like `iadd`, `bspush` or `dup`. These instructions cannot throw run-time exceptions or security exceptions. The JCVM after execution of this instruction proceeds to the next instruction.
II	Instructions that can throw a run-time exception (the JCVM can halt or modify the flow), but not a security exception. These are instructions like `irem` (remainder `int`) or `idiv`
III	Instructions that modify the execution flow. These are instructions like `goto`, `ifnull` or `jsr`. These instructions define branches in the execution flow.
IV	Instructions that define returns from methods, like `ireturn` or `return`.
V	Instructions that can throw `SecurityException`, excluding the method invocation instructions. These are instructions like `checkcast` or `iastore` (all operations with arrays). These instructions require the JCRE to check whether the access to objects is legal, but they do not invoke methods.
VI	Instructions that invoke methods: `invokeinterface`, `invokespecial`, `invokestatic` and `invokevirtual`.

Case VI. The next instruction is an invocation instruction (type VI). These instructions (except for the `invokestatic` instruction) expect to find an object on the stack and invoke a corresponding method of this object. The method *A.s* can be invoked if *B* has a reference to the object `ObjRef` of *A* that implements *A.s*. The JCVM does not check correctness of the object ownership upon execution of the invocation instructions, but does this during the casting instructions execution (instructions `checkcast` and `instanceof`).

B cannot maliciously cast an object of *A* into its own object or an object from a third party *C* due to the typechecking rules for casting. Therefore, an attempt of casting into *B*'s own (or third-party) interface or class will result in a run-time exception and the JCRE will halt *B*'s execution. If *B* will cast an object of *A* into the JCRE's own type (such as `Shareable`), the object will be accessible, but it will not be possible to invoke the method *A.s* from this object.

We now discuss the invocation instructions.

Case VI-invokeinterface. If the next instruction is `invokeinterface` which invokes a method of *A*, then the second operand $\texttt{id}_{\texttt{CP}}$ of this instruction references an externally defined interface (we prove this in §Appendix B). The JCRE firewall will allow to invoke a method across contexts if and only if the invoked interface method belongs to the JCRE or to a Shareable interface, as defined in [(Classic Edition, 2011), Sec.6.2.8 of the JCRE specification]. Therefore, the invoked method is a service of *A*; and no other method of *A* (not from a Shareable interface) can be invoked by the `invokeinterface` opcode.

The `PolicyChecker` verifies that for all services $A.s_1$, such that $A.s_1 \in \texttt{Calls}_B$ and $A.s_1 \in \texttt{Provides}_A$, there will be the corresponding service authorization present in $\texttt{sec.rules}_A$: $(A.s_1,B) \in \texttt{sec.rules}_A$. Therefore, either (**a**) $A.s \notin \texttt{Calls}_B$ or (**b**) $A.s \notin \texttt{Provides}_A$. We can prove that both these cases lead to a contradiction. Therefore, if `invokeinterface` is the next executed instruction in the context of *B* and a service of applet *A* is invoked, then *B* was authorized to invoke it in $\texttt{sec.rules}_A$.

Case VI-invokespecial. The next instruction is `invokespecial`. According to the JCRE specification the object reference on the stack cannot belong to another context when executing this instruction. Therefore only *B*'s own method can be invoked.

Case VI-invokestatic. The next instruction is `invokestatic`. This instruction accesses a static method that belongs to a class, and not an instance. Classes do not have contexts, as objects do; public static fields and methods are accessible from any context [(Classic Edition, 2011), Sec.6.2 of the JCRE specification]. Therefore, if *B* was able to invoke a static method of *A*, the JCRE allows it (no context switch happens, the invoked method belongs to the current context of package *B*).

Case VI-invokevirtual. The next instruction is `invokevirtual`. If the object reference on the stack references an object from another context, the firewall will allow the invocation if and only if the reference belongs to the JCRE [(Classic Edition, 2011), Sec. 6.2.8 of the JCRE specification]. Thus upon execution of this instruction *B* can only invoke its own method or a JCRE method, but cannot invoke methods of another applications.

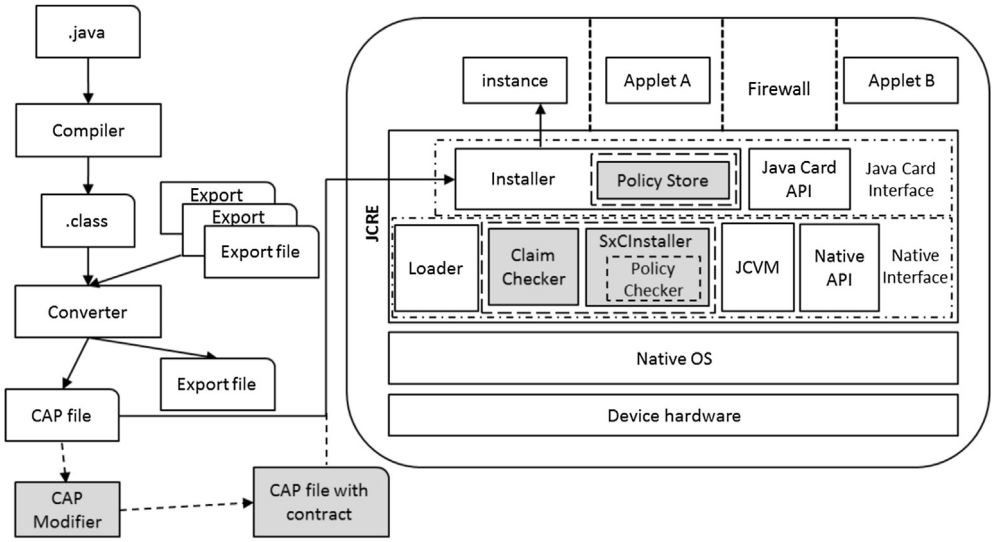


Fig. 6 – The Java Card architecture and the loading process enhanced with the S×C on-device validation.

So, for all JCVM instructions B cannot illegally invoke a method of another application A. The last case is if A used to authorize B to invoke $A.s$ and B was deployed legally, but at some point $\texttt{AppPolicy}_A$ was updated to remove this authorization. This update could have been executed if and only if $A.s \notin \texttt{Calls}_B$. Again, by construction of the `ClaimChecker`, B cannot invoke $A.s$ unless this is declared in $\texttt{Calls}_B$. Therefore, A cannot remove an authorization until B is removed.

7. The prototype design

The requirements on the implementation were elaborated by the smart card manufacturer.

- *The loading protocol should be unchanged.* Secure elements that ignored the framework should be able to work with applets that were aware of it and secure elements incorporating the framework should be able to work with applets ignoring it (backward compatibility). We use Custom components in the CAP files to deliver contracts. Cards ignoring the framework would just ignore the Custom component (i.e. the policy of the applet). And vice-versa, applets unaware of the new framework (those without a contract) in an industrial setting can be processed by the cards aware of the new security mechanism. These applets will be validated to provide 0 services and call 0 services. If some services or service calls are present in the code, an applet with an empty contract will simply be rejected.
- *Minimize changes to the existing JCRE code.* Modification to the loading code should be kept to a minimum, as the addition to the functions of the Loader API can have negative impact on its trustworthiness (and the certification with respect to Common Criteria[6]). Modification of some other parts of the JCRE, like the JCVM or the firewall, were ruled out of consideration due to prohibitive cost and required interaction with multiple stakeholders (e.g. Oracle).
- *Very small persistent and volatile memory footprints.* The prototype footprint could be up to 10 KB of non-volatile memory for storing the prototype itself and the security policy of the card across sessions, and could not use more than 1 KB of RAM (for computation and data structures). The latter requirement was further strengthened by the decision to use a 256 bytes auxiliary temporary buffer to store the temporary computational data. Because this is a buffer fixed by the platform, our prototype consumes no additional RAM for its computation. On different cards different amounts of RAM are available (from 1 KB to 5 KB on modern cards). Thus this temporary buffer restriction ensures the highest interoperability.

The S×C architecture. Fig. 6 depicts the modified architecture and the changes to the development and the deployment processes of Fig. 2, the gray elements belong to the S×C process and the dashed arrows denote the new steps of the development process. We can notice that the deployment process of Java Card is unchanged; and the S×C process adds just one step after the standard Java Card development process (the development and addition of the contract).

The most challenging task was identifying the location and the mechanism of interaction with the `PolicyStore`. The `PolicyStore` has to reside in the EEPROM, because the security policy has to be maintained across card sessions and it has to be modifiable. However, only the Java Card components (applets or the Installer) can allocate the EEPROM space upon the card issuance finalization, and the native components, such as the Loader, cannot do it. Thus the S×C prototype had to be broken into a native part and a Java Card part. The `ClaimChecker` was definitely the native part to be written in C, because it needed to access the Loader API. The `PolicyStore` was definitely a part to be written in Java Card. The `PolicyChecker` could, in fact, be written in both languages and successful implementations of the `PolicyChecker`

[6] Common Criteria is a standard for security certification.

component as an applet exist (Dragoni et al., 2011; Gadyatskaya et al., 2012). We have chosen to implement it in C to ease delivery of the contract. For the memory optimization reasons (to decrease the amount of separate functions) the PolicyChecker functionality was implemented in the S×CInstaller component. The S×CInstaller is implemented fully in C and it serves as an interface with the platform Installer.

The JCRE is implemented in a way that calls from a Java Card component to a native component (for example, from the Installer to the S×CInstaller) are processed without hinderance. Unfortunately, calls from a native component to a Java Card component are prohibited, unless lower level primitives are used. Our solution is to introduce the PolicyStore on the card as a class in the Installer. The Installer, when invoking the S×CInstaller, serves it a pointer to the current security policy array, and the access to this array is done through a native API.

An alternative architecture was to implement a PolicyStore applet that would maintain the security policy. The problem of native-Java Card communication was solved by the usage of the APDU buffer. This solution would have required less modifications to the platform implementation and the S×C prototype could have been tested directly on a PC simulator outside the premises of the platform implementation owner. The trade-off is that all policy data structures have to fit into the APDU buffer. Standard Java Card platforms have buffers of 128 B and 256 B, thus only a very small policy could be maintained (256 B buffer only allows up to 4 applications). In contrast, the usage of native API allows us to increase the security policy size and to have more applications loaded on the card (but it requires more modifications to the platform).

When the actual integration with the device was performed, we have found out that the APDU buffer could not be used during the loading process (platform-specific implementation detail of our smart card vendor). So we could not compare the practical efficiency of the two architectures.

The developer prototype. The standard Java Card Development Kit from Oracle[7] does not support Custom components, so we have developed a CAP modifier tool to embed contracts into CAP files. It is available in our developer version of the tool. The CAP modifier tool allows users to choose to add services to Provides, Calls/func.rules and sec.rules sets, then the dialog will appear where users can insert the necessary AIDs and tokens. When the contract is ready it can be saved for future usage. The contract can also be embedded into the chosen CAP file, and then the CAP modifier can generate the scripts necessary to communicate the CAP file to the card.

The CAPlibrary was shared by the partners; the smart card manufacturer has the actual implementation of the CAPlibrary runnable on the device. For the developer prototype we implemented the CAPlibrary following the JC specifications.

We have made available the S×C prototype version for testing purposes[8]; it runs on a PC and can be used by applet developers to practice the S×C scheme. It also includes several testing scripts, the CAP modifier tool to embed the contracts, the CAP files of the running example applets and a user manual.

7.1. Policy management

The PolicyStore is responsible for storing the security policy of the card. It has to be organized efficiently, so that the PolicyChecker algorithm is fast while the space occupied by the security policy data structures is small. Once a new applet has been validated (both the ClaimChecker and the PolicyChecker returned True), the security policy of the card is modified by including the contract of the new applet. The S×CInstaller stores the contract in the buffer, and the PolicyStore retrieves it and adds to the policy data structures. In case some applet is removed, after the PolicyChecker has approved this change, the PolicyStore will remove the contract of this applet from the policy.

For the contract-policy compliance check we have used bit vectors, assuming up to 10 loaded applets at each moment of time (the 11th will be rejected by the current implementation, but it is possible to free the space by removing something loaded), each applet can provide up to 8 services. These numbers are more than enough for modern secure elements: from our personal experience, current numbers are respectively 4–7 deployed packages (most of them libraries) and 0–1 services. In the same time, our policy format is not restricted with respect to possible authorized clients AIDs, these are unconditioned.

Notice that the PolicyStore only maintains the security policy data structures and performs updates, but the main contract-policy compliance check is executed by the PolicyChecker, which retrieves the policy from the dedicated platform buffer.

The possibility of applet policy update without reinstallation is one of the main benefits of the S×C approach. To update the policy, the applet provider needs to contact the PolicyStore. We consider atomic updates: addition or removal of an authorization to sec.rules and addition or removal of a necessary service to func.rules. A possible AppPolicy update scenario for the example in §3.1: the Purse applet provider chooses to allow the applet MessagingApp to call its service payment().

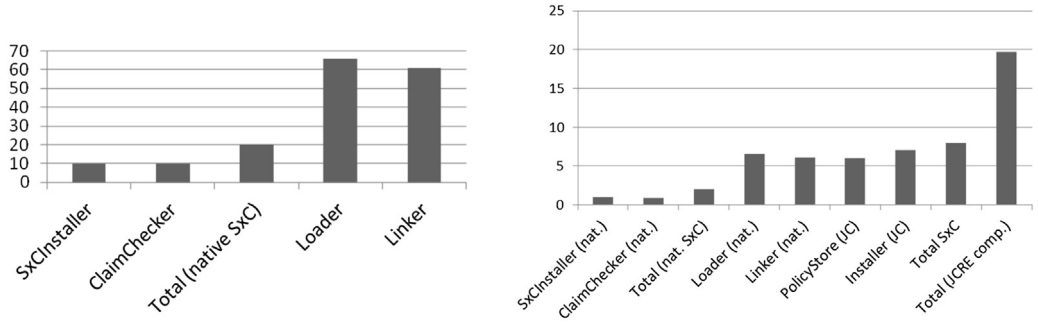
The application provider needs to transmit to the PolicyStore component an APDU (Application Protocol Data Unit) sequence specifying the type of the update to be executed, the

Table 3 – The S×C framework components sizes.

Component	Compiled (PC)	Compiled (device)	LOCs
S×CInstaller	10 KB	1 KB	178
ClaimChecker	10 KB	0.9 KB	170
Total (C)	20 KB	2 KB	348
PolicyStore	6 KB	6 KB	148
Total S×C	26 KB	8 KB	

[7] http://www.oracle.com/technetwork/java/javacard/overview/index.html, accessed on the web in Jan. 2013.

[8] Accessible from http://disi.unitn.it/~gadyatskaya/SxCdeveloper.zip.



(a) Comparison of embedded native components sizes (.obj files compiled on a PC Win32 Simulator, in KB).

(b) Comparison of embedded components sizes compiled on device (in KB).

Fig. 7 – Sizes of the prototype components and their comparison with the JCRE components sizes.

AIDs of the applets in question (which applet's policy has to be updated and which is the AID in the security or functional rule). The `PolicyStore` runs the check if the suggested update leaves the card in a secure state, and executes the update in case of a positive result. Notice that the necessary step of the applet provider authentication should be present in the policy update protocol; it can be implemented using the standard GlobalPlatform middleware.

8. Resource analysis of the implementation

Several features are important for the embedded software. Traditionally for smart cards the most important feature is the memory footprint of the new components. For instance, a study of commercial Java Cards (Mostowski et al., 2007) lists memory available on the card as the second card feature, after the supported specifications; while the run-time performance of cryptographic primitives arrives much later. In the NFC world the user experience is crucial, and it is required that the applet operations (such as execution of the payment process by the `Purse` applet and the ticketing operation by the `Transport` applet in the example in §3.1) are very fast (fractions of seconds). In contrast, the OTA deployment of applets can take more time, because it can be executed by the stakeholders, while performing other operations such as updating status or charging money. Therefore, the load time performance overhead for our framework is not as important as the memory footprint. The `S×C` framework components run algorithms that are linear in the size of the processed CAP file; the load time overhead is insignificant in comparison with the operations performed by the `Loader` and the `Linker` JCRE components. In the same time, our framework actually reduces the applets' execution time, because the ACL checks are not performed anymore.

Therefore, we first focus on the memory footprint. Since integration with an actual device is very costly, we first measured the footprint of the prototype compiled on a PC Win32 simulator (for compilation we used the Microsoft Visual C compiler `cl.exe` with appropriate options). Table 3 presents the data on the components sizes. The target device is an in-use Infineon smart card integrated circuit (an actual multi-application (U)SIM secure element). The `PolicyStore` component was measured as a CAP file, because it is the actual size on device. We also provide the number of lines of the source code (LOCs). To give a feeling on the level of optimization, Fig. 7 compares the embedded native `S×C` components sizes with sizes of the standard native JCRE components (the `Loader` and the `Linker` compiled on the PC simulator); Fig. 7(b) compares the sizes of the `S×C` components compiled on device with on-device sizes of the Installer (measured as a CAP file), the `Loader` and the `Linker` components. The total size of the `S×C` prototype is bigger than the `Loader` or the `Linker`, because the `PolicyStore` is implemented in Java Card, while `Loader` and `Linker` are native components and thus are highly optimized.

It should not be surprising that the size of the native components compiled on a PC is an order of magnitude bigger than the size of the native components deployed on a device. This decrease of size is explained by multiple optimizations to the native components structure carried out before deployment. For example, the device memory is smaller, so all pointers and integers are shorter.

For the RAM allocation, in addition to the auxiliary temporary buffer, the `S×C` prototype allocates less than 100 B (only local variables, no transient arrays are used). The EEPROM consumed by the `PolicyStore` for the security policy data structures is 390 bytes (two arrays, 135 B and 255 B).

Processed applets. Table 4 presents the relevant details of some of the applets we used to test the prototypes. The `Purse` and `Transport` applets were developed by the smart card

Table 4 – Details of applets used for testing and evaluating the `S×C` prototype.

Applet	CAP file size	# of methods in CAP file	# of services	LOCs (.java)
`Purse`	2.5 KB	6	1	66
`Transport`	2.5 KB	5	0	92
`EID`	11.2 KB	81	1	1419
`ePurse`	4.7 KB	16	1	431

manufacturer partner for functionality testing relevant for client–server interactions. The `ePurse` is another electronic purse applet provided by the smart card manufacturer. The `EID` applet is an open-source electronic identity applet (Philippaerts et al., 2011); originally it did not include any services, so we have added 1 Shareable interface including 1 method.

There is no agreed industry benchmark for the representative size of the "average" applet. However, generally a CAP file of 10 KB is already a big applet, most telecom applets are between 1 and 10 KB.

9. Security analysis

We now review and discuss the security assumptions behind our guarantees.

- **Correct implementation of the Java Card development, deployment and execution environments.** Soundness of the framework algorithms relies on the correct implementation of the JCRE and the JCVM, and we assume they are in full compliance with the specifications (Classic Edition, 2011). For instance, we require that the only way for applets to communicate is through Shareable interfaces. Another crucial assumption is that the bytecode is trustworthy and it respects the Java type safety assumptions. These assumptions are standard for the JCRE security.

We base correctness of our technique on the guarantees offered by the JCRE (§6). For example, we expect that illegal (security-violating) context switches upon execution of a JCVM instruction correspond to security exceptions thrown by the JCVM (class `SecurityException`) The JCRE specification defines how the context switches should be handled (the firewall rules). If an instruction makes an attempt to switch context illegally (not following the rules), a security exception will be thrown. The current execution will be aborted and the sensitive resources will be protected. This is why we consider instructions able to throw this exception separately in the taxonomy. Another special type of exceptions is `SystemException`, which can be thrown by the JCVM at any point of the execution. This exception type handles the JCVM errors. For the other exception types, besides `SecurityException` and `SystemException`, the specification expects that application providers can catch these exceptions and handle them correctly. Any uncaught exception results cause the JCVM to halt, the current applet execution is aborted. We consider that in case of an uncaught exception, the JCRE context will become the active context.

- **The package AIDs cannot be spoofed.** The AIDs are assigned uniquely following the ISO standard. The existence of the AID impersonation attacks (registration of a new applet instance with a spoofed AID (Montgomery and Krishna, 1999)) and the need for reliable CAP file and applet authentication techniques are acknowledged by the JC practitioners for a very long time. The GlobalPlatform middleware provides the means for secure card content management (including delegation) and offers sophisticated mechanisms for application and terminal authentication. A full industrial implementation of our framework can leverage these mechanisms. So we assume the applet code is authentic and assigned to an authentic AID. We also assume that the platform correctly authenticates applet owners for the policy updates. Our focus is on the code permissions and service invocations.
- **Access control to services is specified per a package rather than per an applet instance.** The service access control policies enforced by our framework are based on the package AIDs, while the current JC methods of service access control are based on the applet instance AIDs. However, the package AIDs are more trusted than the applet instance AIDs, as the package AIDs cannot be modified after the conversion, while the applet instance AIDs can be changed freely. In the same time, as all applets of the same package can freely communicate, granting access for one applet instance means in practice granting access for its whole package. Thus the package-based access control does not worsen the granularity of the current JC access control policies. As well, in practice the industry only needs the ACLs based on the applet provider identity (access is granted only to the trusted partners).

In the current paper we assumed that each package includes exactly one applet. Our approach can also be directly applied in the case when a single package includes multiple applets; no changes to the contract model or the framework components are required.

Regarding the abstract model of the platform, we have conjectured 1–1 correspondence between packages deployed on a card and instantiated applets. In fact, the card can host deployed packages that are not instantiated. If we do not consider library packages and enforce the condition that each package does not implement Shareable interfaces defined in other packages, then the un-instantiated packages can not participate in the inter-package communication (in both roles of a server and a client), therefore our security theorem still holds. A single package can be instantiated multiple times, but all applet instances will belong to the same context and they can be treated as the same instance.

- **Restricted amount of deployed packages.** There is no substantial limitation on the number of packages mentioned in the policy (as authorized clients), but in order to improve the policy management efficiency our prototype allows at most 10 packages to be deployed, validated and listed in the security policy at any given time. For modern secure elements 10 loaded packages is a significant amount: from our experience, usually high-end multi-applet cards carry around 4–7 packages, most of them being library packages used for personalization (like GlobalPlatform), loaded at the card manufacturer premises. However, the limit on the number of deployed packages can be restrictive for the industry, as we target open secure elements of the future. Our implementation can be improved by enabling dynamic scaling of the policy structures.
- **No services defined outside applets.** JC allows library packages that do not contain any applets, but they can define Shareable interfaces. We have investigated an extension of the current proposal in order to consider also library packages and to capture implementation of a service defined in a separate package and to strengthen the demands on the functionally necessary services by requiring that the service

is provided if there is a class implementing the interface defining this service. To deal with these problems it is possible to expand the contracts by including in the `AppClaim` also the set of service definitions declared in the current package. There will be a set of *defined services* and a set of *actually provided services*. The `Calls` set will be based on invocations of defined services (because CAP files contain the interface token, but not the actually invoked class), and the `Provides` set will refer to the services implemented in the current package. The `PolicyChecker` will ensure that the policy of the package implementing a service is more liberal than the policy of the package defining this service.

The pre-loaded libraries (those are deployed at the card manufacturer premises before the card issuance) in the industrial setting can be accounted in the policy structure from the very beginning.

- **Each applet implements only services declared in this package.** We interpret provided services as services that are declared in the Export file of the package. Thus the `S×C` approach to ensure the functionally necessary services availability requires a commitment from the server that the actual implementation of the declared services will exist at run-time.
- **The called services are identified in the bytecode by the static token identifiers.** While analyzing the code, we could try to track the object references on the stack, thus inferring all possible objects of the server that could be referenced by the client during the `invokeinterface` opcode execution. Unfortunately, only the server's code defines which objects it will provide and to whom. It is even possible the server is not yet on the card when the client is loaded (and it could never arrive). Thus the load time analysis can be only as precise as the static tokens provided in the client's code.
- **The application policy update is secure.** The `S×C` framework is fully compliant with the standard JC application update scheme – when an application is removed and then redeployed again. This scheme has to be used when the functional code needs to be updated (including removal of an external service invocation or addition of a provided Shareable interface). We have proposed a novel flexible approach to update the security policy of an application without redeployment. However, this scenario introduces new potential insecurities, because it exposes a new communication scenario with the Installer. Therefore to be used in practice full security evaluation and certification of the additional features of the Installer and the application policy update protocol is required.

10. Related work

10.1. *The Java Card security*

The Java Card platform attacks and countermeasures. The security of the Java Card platform and the novel security issues raised by the open multi-application architecture are discussed in the community for more than a decade (Girard and Lanet, 1999). The hardware and software attacks on the platform (the side-channel and fault-injection attacks) are outside of the scope of this article. The interested reader can find the detailed overview of the latest advances in these kinds of attacks and the efficient countermeasures in Barbu et al. (2011), Barbu et al. (2012a), Bouffard et al. (2011a), Bouffard et al., (2011b), Markantonakis et al. (2009) and Leng (2009). Among the tools for enabling security on Java Card are the bytecode verifier (executed off-card or on-card) (Bouffard et al., 2011a), a hardened defensive JCVM (Lackner et al., 2012; Dubreuil et al., 2012) and a system for control flow integrity verification for Java Card (Bouffard et al., 2011b). Regarding the applet interactions security on Java Card, W. Mostowski and E. Poll have investigated in Mostowski and Poll (2008) the potential of abuse of the Shareable interface mechanism (among discussing other types of attacks on JCRE); they have demonstrated that with the Shareable interface mechanism the JCVM can be tricked into the type confusion, and observed that all tested cards with the on-card bytecode verifier rejected applications with Shareable interfaces. The authors hypothesize that this rejection of the sharing applets (being, strictly speaking, incompatible with the JC specifications) is a safety measure, because the on-card bytecode verifiers are not able to detect the type mismatch. The run-time countermeasures against this attack are discussed in Bouffard and Lanet (2012). However, the only practical attack that exists in the literature (Mostowski and Poll, 2008) assumes collaborating malicious client and server applets. A standalone malicious applet (being it a client or a server) cannot lead the JCVM to a type confusion.

One of the most eminent features of the updated Java Card Classic edition is the garbage collector (Classic Edition, 2011). G. Barbu et al. have investigated the replay attacks against the Java Card platform with the enabled garbage collector (Barbu et al., 2012b). In the reply attack an honest applet is removed while a malicious applet retains the references to the honest applet objects (the references can be guessed by using the rules of the reference assignment on Java Card, that are quite simplistic). These objects are not garbage collected, but cannot be accessed due to the firewall mechanism. However, if a new applet is immediately installed, it might be assigned the same context identifier, and the malicious client will pass the object references to the new applet. Being an applet from the "same" context as the object references, the new applet can access the objects of the original applet, that is now removed; thus the firewall mechanism can be bypassed.

The communication protocols of the smart card and the host terminal and their security are also outside the scope of this article. The interested reader can find a summary of the control flow difficulties in these protocols in Li and Zdancewic (2004).

The paradigm shift in the card ownership model. The traditional smart card business model assumes a centralized controlling authority ("issuer") that manages the card. Application providers have to negotiate with the card issuer the terms for loading and removal of applications; the card holder does not take any decision or any responsibility in this model. Recently the proposals of a *user-centric* open multi-application card appeared (Akram et al., 2010; Sauveron, 2009). In the new proposed paradigm the card holder is responsible for taking the decision to install or remove an applet. In the same time, the application providers require certain security guaranteed for their applets on the shared platform; this can be ensured by sufficient security mechanisms available on the card (Akram et al., 2010; Dragoni et al., 2001).

The NFC secure element security. M. Roland et al. in (2012) overview the existing approaches for secure element implementation in the NFC-enabled mobile devices. They conclude that the introduction of smart cards into mobile phones as secure elements for NFC transactions can lead to unexpected vulnerabilities due to the current lack of standardization for the NFC card communication protocols. The standard smart card communication protocols used for NFC transactions can introduce, for instance, the denial of service attack: any phone application can trigger an authentication protocol with the device secure element, and the standard authentication protocols on some devices, such as Nokia 6131 and Samsung Galaxy S, allow only for limited number of authentication attempts. After reaching the threshold the card will be brought to the terminated state, from which it cannot be brought back to a working state. Therefore, a malicious smartphone application can render the secure element terminated.

Akram et al. (2012) identify a cooperative architecture scheme for the NFC-based mobile services that merges the Trusted Service Manager architecture, traditionally adopted in the NFC trials, and the user-centric smart card model. The authors allow the device holders to take the decision on installing an applet, but the users now have to pay for this to the card controlling authority (a telecom operator) that has provided the open secure element infrastructure.

Application code verification. The smart card applications are quite sensitive and usually require a tedious formal verification and certification process (Narasamdya and Perin, 2009). To facilitate the formal verification process G. Barthe et al. have developed the JACK tool for validation of security of the Java Card applets (Barthe et al., 2006). JACK can verify Java source and bytecode, it includes automatic annotation generation algorithms and integration with the Coq prover. VeriFast (Philippaerts et al., 2011) is another tool for automated verification of Java and C programs, that can be used for the Java Card programs verification.

Run-time verification for Java Card. Souza da Costa et al. (2012) are the pioneers in application of the run-time verification techniques on Java Card. They have proposed the JCML (a specification language for Java Card applications derived from the JML) and an implementation for it (a compiler that generates the run-time verification code). The solution is based on the Design-by-Contract approach: some logical assertions are added to the source code to specify its contract. The assertions are verified at run-time. In contrast to the S×C approach, the application contracts in Design-by-Contract deal with lower level code properties, such as code invariants, preconditions and postconditions for the code parts. The experiments in Souza da Costa et al. (2012) show viability of the JCML language adoption for Java Cards. However, this language does not improve the application communication access control mechanism, because the assertions are still embedded in the applet code.

10.2. Control of applet interactions on Java Card

Control of interactions for predefined sets of applets. Most of the proposals for the control of applet interactions consider only static scenarios, when the set of applets to be deployed on the card is known in advance and can be analyzed at the premises of the card issuer or the trusted controlling authority (Girard, 1999; Schellhorn et al, 2000; Bieber et al., 2002; Huisman et al., 2004). For example, Avvenuti et al. (2012) have developed the JCSI tool that verifies that a set Java Card applications respects pre-defined information flow policies. As we have discussed, the static scenarios are not appropriate with the dynamic nature of the novel NFC-enabled platforms.

Dynamic applet interactions scenarios. Similar to ours card evolution scenarios (application loading and removal) are considered in Fontaine et al. (2011b), where Fontaine et al. propose the TCF mechanism to enforce transitive control flow policies on Java Card. These policies capture application collusions, when two or more applications engage into a chain of method invocations. These policies are stronger than the policies enforced by the S×C framework, which captures only the direct method invocations. The main limitation of the TCF prototype is the focus on security domains and not on package AIDs. Security domains are very coarse grained administrative security roles, typically used to delegate installation privileges (usually a handful). As a consequence we can provide a much finer access control list. Therefore, while the S×C code-contract and contract-policy checking steps can be accommodated by the TCF mechanism, this mechanism does not support as rich set of authorized clients, as the S×C approach does. We do not see an immediate solution to this problem, because the finer access control lists for TCF will require substantial memory resources to store the policy.

In (2011a) Fontaine et al. develop other types of policies suitable for open multi-application Java Cards: the "global" policies that allow to specify in a centralized manner sets of prohibited method invocation chains across multiple applications and the full-fledged information flow policies, that are inspired by the work of Ghindici and Simplot-Ryl (2008). The information flow verification systems suitable for small Java-based devices proposed in Fontaine et al. (2011a) and Ghindici and Simplot-Ryl (2008) include off-device and on-device steps. The off-device step consists of creation of an information flow certificate (an information flow contract, that contains the information flows within the application and the secret/public annotations) for each application. Then on device this certificate is checked in a proof-carrying-code fashion and matched with the information flow policies of other applications. The information flow policies are very expressive, but no practical implementation of the proposed systems for Java Card exist, due to the resource and other constraints. For example, the mechanism proposed in Ghindici and Simplot-Ryl (2008) cannot be implemented for the Java Card Classic edition, because the latter does not allow custom class loaders, and even implementation for the Java Card Connected edition 3.0.1 may not be effective due to significant amount of memory required to store the information flow policies.

In the user-centric card ownership model the application interactions are very important, as there is no central controlling authority and it is difficult to have even implicit trust in applets installed on the card. Akram et al. (2011) propose an on-card framework for run-time applet authentication and verification, that can augment the Java Card firewall. The authors argue that the AID impersonation attack (Mostowski

and Poll, 2008), when an applet can masquerade itself as bearing a legitimate AID of another applet, is quite dangerous in the user-centric model, when there is no controlling authority to check which applets are installed; and therefore the platform requires a set of protocols (based on the trusted certification authority signatures and applet hashes) for establishing trust between applets on the card. However, in presence of the trusted certification authority the validity of the applet AIDs can be established off-card (at least for the applets that have passed the certification process). We can expect that the load time S×C code validation performed by the card itself can be very beneficial for the user-centric open smart cards, because with our proposal each platform is independent and is always maintained in a secure state with respect to applet interactions.

The investigation of the Security-by-Contract techniques for Java Card is carried out in Dragoni et al. (2011), Gadyatskaya et al. (2012) and Gadyatskaya et al. (2011) targeting dynamic scenarios when third-party applets can be loaded on the platform. Dragoni et al. (2011) and Gadyatskaya et al. (2012) propose an implementation of the PolicyChecker component as an applet. While possible in theory, it has not solved in any way the actual issue of communication between that native and the JC components that we have addressed here. This problem might only be solved if the authors of Dragoni et al. (2011) and Gadyatskaya et al. (2012) could have access to the full Java-based JCRE implementation. The specifications of the JC technology do not prohibit this, but in practice full Java-based implementations do not exist. Our ClaimChecker algorithm is more practical than the algorithm in Gadyatskaya et al. (2011), which runs in one pass over a CAP file, but needs to allocate memory to store temporary data. For big CAP files (e.g. EID) the dynamic memory allocation is prohibitive and it is necessary to reuse the space, though increasing the number of runs over the CAP file.

10.3. *Multi-tenant platforms*

We survey the existing techniques for other most relevant multi-tenant platforms.

Android. Typically, mobile applications (*apps*) for Android are written in Java and compiled into DEX binaries. These binaries are loaded on the Android platform and are executed by the DVM (Dalvik Virtual Machine). Access to sensitive resources on the platform is guarded by permissions, which are granted to apps at the installation time. For some sensitive permissions (like the GPS sensor access) the user is prompted.

Enck et al. (2009) have developed the Kirin security service for Android that performs lightweight app validation at installation time. The Kirin installer parses the manifest of the loaded app and extracts the requested permissions. These permissions are then compared with a predefined set of Kirin's security rules and if a dangerous functionality access is requested, the user is notified. Kirin is implemented as an app showing feasibility of running on device.

Ongtang et al. (2009) were the first ones to advocate the need of Android apps to protect themselves. They proposed new types of app policies to be enforced on Android by the Saint framework, among them permission assignment policy that protects permissions for accessing app interfaces and interface exposure policy that controls how the interfaces are used. Saint regulates permissions assigned to apps at installation and enforces the app interactions policies.

Proposals for Android that suggest off-device verification (such as Blasing et al., 2010; Enck et al., 2011) performed by the user generally do not take into account that an average user is not security-aware and he/she would probably not consider the security threats of inter-app communications. For secure elements this approach is not possible. Off-device app bytecode rewriting to enforce security is a powerful technique (Xu et al., 2012), as one could modify apps to use a specific policy-regulated API for communications, or even to remove unauthorized interactions. Unfortunately, rewriting is dubious from the business perspective. There is no clear understanding who is liable in case a rewritten app failed. Is it the developer or the rewriter (user/app market)?

Run-time monitoring of execution and inter-app communications is another known technique. Run-time monitors capture the exact app behavior and are more precise than the over-approximating static code analysis. An example of lightweight app interaction policies enforcement at run-time is presented in Ongtang et al. (2009), richer policies are elaborated, for instance, in Bugiel et al. (2012), Enck et al. (2010) and Felt et al. (2011). However, the precision comes at the price of run-time overhead.

JavaME, .Net. The S×C paradigm was proposed for multi-application mobile devices (JavaME and .NET technologies) (Bielova et al., 2009; Desmet et al., 2008). In the original S×C scheme an application arrives on the mobile platform equipped with a contract and signed by the developer. The contract contains a suitable formal model of application security-related behavior, such as the number of SMS sent per execution or access to the sensitive user agenda. A security policy set by the user or the telecom provider defines allowed and forbidden actions. The contract is matched by the device with the security policy before the execution (Bielova et al., 2009). In case of failure an inlined reference monitor is used (Desmet et al., 2008). This approach allows to run even potentially dangerous applications in a sandbox environment.

In our scheme for Java Card the contract is matched with the applet bytecode; while in the S×C scheme for mobile devices the contract-code compliance has to be trusted and is based on the digital signature of the provider. The contract-code matching step is, essentially, missing. Also, in the S×C scheme for mobile devices the security policy of the mobile platform is defined by the user or by the telecom provider. This is justified because the policy protects the sensitive resources of the user. However, in our scheme for JC the cumulative security policy is composed by the contracts of all applets currently loaded on the card, because the platform protects the sensitive resources of the applets.

On-market verification. There are smartphone markets, such as Apple Store and Google Play, where the platform providers (Apple and Google, respectively) perform some off-board checks of apps, but these checks do not aim at the app interactions. Apple's official statement[9] says "Most rejections are based on the application containing quality issues or

[9] http://www.apple.com/hotnews/apple-answers-fcc-questions/, accessed on the Web in Jan. 2013.

software bugs, while other rejections involve protecting consumer privacy, safeguarding children from inappropriate content, and avoiding applications that degrade the core experience of the iPhone".

Besides bug and nudity checking the process is geared to ban competitors of Apple or its partners. Google Voice was rejected for "replacing …Apple user interface with its own user interface for telephone calls, text messaging and voicemail". Aside from banning competitors, Apple mostly relies on identity verification to avoid malware on the market. In the same time, the off-device verification techniques that could be done on the app market are of limited applicability to the inter-app communication security, because the market does not fully know the set of apps already installed on devices and it is infeasible to validate all possible combinations of apps.

On-board credentials. The on-board credentials (ObC) approach is developed by Ekberg et al. (2008) and Kostiainen et al. (2009). The authors develop a security architecture for hosting credentials (secret keys and algorithms) on multi-tenant secure hardware platforms. Their approach enables open credential platforms, where each credential provider can load her secret data independently. There are also (restricted) means for interactions of the provisioned programs. To enable interactions (with the purpose to access a secret data or an algorithm), the credential provider has to create a new *family* and endorse the authorized programs (by submitting the family secret key and the program hash) to this family. On board ObC programs are validated with respect to hashes, that is yet another form of signature verification. We perform on device semantic validation on what programs do and invoke. A revocation of access is not directly supported in the ObC paradigm; in order to prevent usage of a credential by no longer trusted partner one needs to disable the old credential and load a new one with a different hash.

The S×C framework is complementary to the ObC technology and could be used for its enhancement. The current approach of endorsement induces a significant run-time overhead for credential execution. The ObC interpreter language can be modified to include the specific instruction for credential invocation, similarly to the Java Card system. Then the load time code validation can be leveraged in order to speed up the run-time computations and enable better revocation mechanism.

11. Conclusions and outlook

In the paper we have presented the S×C prototype implementation that can be embedded on a real device. The S×C prototype aims to ensure security of application interactions on Java Card during applet loading or removal. It also handles applet policy updates that do not require reinstallation. We have demonstrated that our framework is correct with respect to the JCRE specification.

We perceive the separation of the security code from the functional code as a significant improvement in the OTA loading setting, because it easies updates of the policy and decreases the execution time of applications. However, there is also a downside in it: the applications cannot execute selectively based on who is calling them at the moment.

If the platform owner wants to deploy a full isolation policy on the secure element, our framework provides a noninvasive way to do it. The `ClaimChecker` can ensure that loaded applets do not provide and do not call any services; the JCRE implementation does not need to be modified and re-certified.

We have presented a full ecosystem for the on-device S×C validation: the CAP modifier tool to embed contracts into CAP files and the S×C framework that includes the `ClaimChecker` and the `S×CInstaller` components written in C and integrated with the card native components, and the `PolicyStore` component written in Java Card and integrated with the Installer. We have also discussed integration with an actual device. We believe that these results are interesting for anyone looking into enhancing application security using secure elements.

Potential market acceptance. Besides the technical aspects there is also a more general question: how mature is the market to accept this solution? At present, most companies using JC are not yet ready to forgo the cushioned assurance of certification of interactions for the most sensitive applets locked on the card. Yet, there is an interesting trend that makes our technology appealing.

From an industry perspective what is important is the security of the whole *product* (the secure element platform combined with all loaded applets). This was ensured by security certification for compliance with Common Criteria or other industry standards (VISA, etc.). Due to the costs and operational constraints of the security certification, the industry is now partitioning applications into highly sensitive ones and less sensitive ("basic") ones. The topmost sensitive applications would still be certified at the manufacturer's premises and possibly pre-loaded, but the "basic" applets would no longer be certified. Rather, the product as a whole would be certified secure but open for OTA loading of "basic" applets.

Since "uncertified" (in the Common Criteria sense) does not mean "insecure", those "basic" applets are still subject to a large number of security rules and validation checks needed to ensure security of the final product. These checks are so far performed off-card before loading. In the context of OTA loading of the "basic" applets, the S×C approach is thus promising. It could allow to get rid of (a part of) the off-card security checks, performing them on board instead. This will reduce the time-to-market for service providers and facilitate the deployment of those applets.

Acknowledgments

We thank Eduardo Lostal for developing the prototype. This work was partially supported by the EU under grants EU-FP7-FET-IP-SecureChange and FP7-IST-NoE-NESSOS.

Appendix A. Implementation details

The detailed `ClaimChecker` *algorithm*. Algorithm Appendix A.1 follows the English description in Alg. 5.1. In order to access the components of the CAP files on the secure element we use the `CAPlibrary` library provided by the smart card

manufacturer. For the sake of clarity some simple checks performed by the algorithm are written only in English. The received CAP file is a byte array which is structured accordingly to the CAP file specification. Thus the algorithm refers directly to items (fields) of the structures defined in the CAP file specification (Classic Edition, 2011) and we indicate which component structures belong to in the object-oriented notation.

```
Require: A CAP file, byte TempBufferCalls[ ], byte TempBufferOffsets[ ].
Ensure: True/False, Contract.
//Custom Component: get Contract;
//Descriptor Component: go through the interfaces and the interface methods;
boolean found = False;
short InterfaceToken;
for i = 0 to Descriptor.classes_count do
    if classes[i] has a flag ACC_INTERFACE = 0x40 in the access_flags then
        //Export Component: get tokens of shareable interfaces;
        for j = 0 to Export.classes_count do
            if Export.class_exports[j].class_offset = Descriptor.classes[i].this_class_ref then
                // this is an exported shareable interface;
                found = True;
                InterfaceToken = j;
        //check for match with the provided services in the contract;
        if found then
            for j = 0 to Descriptor.classes[i].method_count do
                found = False;
                for k = 0 to Contract.provides_count do
                    if ⟨ InterfaceToken, Descriptor.classes[i].method[j].token ⟩ = Contract.provides[k] then
                        found = True;
                if found = False then
                    //there is no declared provided service return False
        else return False
check that all provides_info were found;
//Proceed to the called services
short PackageToken;
//Import Component: get package AIDs of imported packages and their indices;
//for each server AID in the Contract check it is imported;
for i = 0 to Contract.calls_count do
    for j = 0 to Import.count do
        if Contract.calls[i].server_AID matches with Import.packages[j].AID then
            PackageToken = j;
    if some declared called AID is not imported then return False;
    store the called services info in TempBufferCalls[ ] in the following format ⟨PackageToken, Contract.calls[i].interface_token,
Contract.calls[i].service_token⟩;
short method_number = 0;
//Descriptor Component: go through the classes and obtain the offset of each method, store it in the temporary buffer;
for i = 0 to Descriptor.classes_count do
    for j = 0 to Descriptor.classes[i].methods_count do
        store Descriptor.classes[i].methods[j].method_offset in TempBufferOffsets[ ];
        method_number + +;
// Method Component: for each method offset parse the bytecode to find called services;
for i = 0 to method_number do
    CurrentMethod = Method.TempBufferOffsets[i]
    parse the bytecode of CurrentMethod
    if the invokeinterface instruction is found then
        store the operands into LocalInterfaceToken and ServiceToken;
        // Constant Pool Component: check the high bit of the structure is 1, then get the interface token and check the called
service (AID, interface token, method token) to be present in the Calls set;
        if the high bit of ConstantPool.constant_pool[LocalInterfaceToken] equals to 1 then
            InterfaceToken = ConstantPool.constant_pool.cp_info[LocalInterfaceToken].class_token;
            PackageToken = ConstantPool.constant_pool.cp_info[LocalInterfaceToken].package_token;
            if ⟨PackageToken, InterfaceToken, ServiceToken⟩ does not exist in TempBufferCalls[ ] then return False;
check that all calls_info were found;
// Header Component: get the current package AID; return {True, Header.package.AID, Contract}
```

Algorithm Appendix A.1 — The `ClaimChecker` Algorithm.

Appendix B. The correctness proof.

We first prove an auxiliary Proposition Appendix B.1.

Proposition Appendix B.1. *When the instruction* `invokeinterface` $\langle \text{AID}, \text{id}_{\text{CP}}, \text{t}_\text{m} \rangle$ `[ObjRef]` *is executed, if the* `CP [id`$_{\text{CP}}$`]` *item is an externally defined interface, then* `ObjRef` *references an object belonging to another context. If the* `CP[id`$_{\text{CP}}$`]` *item is an internally defined interface, then* `ObjRef` *references an object belonging to the current CAP file context.*

Proof. Three cases are possible: 1) upon execution of `invokeinterface` $\langle \text{nargs}, \text{id}_{\text{CP}}, \text{t}_\text{m} \rangle$ `[ObjRef]` the `CP[id`$_{\text{CP}}$`]` item refers to an externally defined interface; 2) upon execution of `invokeinterface` $\langle \text{nargs}, \text{id}_{\text{CP}}, \text{t}_\text{m} \rangle$ `[ObjRef]` the `CP[id`$_{\text{CP}}$`]` item refers to an internally defined interface; 3) upon execution of `invokeinterface` $\langle \text{nargs}, \text{id}_{\text{CP}}, \text{t}_\text{m} \rangle$ `[ObjRef]` the `ObjRef` object reference is incompliant with the interface resolved from `id`$_{\text{CP}}$.

Let us assume `invokeinterface` $\langle \text{nargs}, \text{id}_{\text{CP}}, \text{t}_\text{m} \rangle$ `[ObjRef]` is executed and the `CP[id`$_{\text{CP}}$`]` item refers to an externally defined interface, but the current `ObjRef` on stack refers to the object belonging to the current CAP file context. The referenced object has to implement the interface specified by the `CP[id`$_{\text{CP}}$`]` item, and therefore, since this object was created by the current package, either the current package has implemented this interface, or has extended and implemented this interface. This contradicts the assumption that all the CAP files implement only Shareable interfaces defined in the same CAP file.

If `invokeinterface` $\langle \text{nargs}, \text{id}_{\text{CP}}, \text{t}_\text{m} \rangle$ `[ObjRef]` is executed and the `CP[id`$_{\text{CP}}$`]` item refers to an internally defined interface, but the current `ObjRef` on stack refers to the object belonging to a different CAP file context. Again, the referenced object has to implement the interface specified by the `CP [id`$_{\text{CP}}$`]` structure, therefore, another package (the owner of the `ObjRef`) has to implement this interface, which contradicts the previously mentioned assumption.

The JCRE protects the object referenced by ObjRef from being cast to an incompliant interface upon reception of the object reference. Namely, the checkcast ⟨id_{CP}⟩ [ObjRef] instruction, where CP[id_{CP}] item is a reference to an interface type, requires that the object referenced by ObjRef implements the interface type referenced by CP[id_{CP}], otherwise the ClassCastException is thrown upon execution of the casting instruction. □

Theorem Appendix B.1. *In the presence of the* S×C *framework all methods invoked by any deployed application B are authorized by the platform policy, or are allowed to be invoked by the JCRE.*

Proof. The proof goes over all possible cases of method invocation on the platform. Assume the theorem does not hold: *B* is a deployed application and it invokes some method not authorized in the platform policy (it cannot invoke a method against the JCRE rules, unless the platform is implemented incorrectly). Since *B* is a deployed application, it has been validated by the ClaimChecker and the PolicyChecker, also all executed application policy updates of *B* were validated.

We consider the invocation of one's own method as obviously authorized, though the platform policy does not specify it explicitly. So the remaining case is when *B* tries to invoke a method *s* of some other application *A*. If *A* is not deployed or method *s* is not provided, *B* will obviously fail. We need only to consider the case when *A* is already deployed and *s* is actually provided by *A*. Applet *A* has been validated by the ClaimChecker and the PolicyChecker, and all executed policy updates of *A* were approved.

We reason inductively over the length of execution of a platform (number of executed instructions) that the invocation cannot happen. Let σ be a sequence of instructions executed by the JCVM leading to the context of applet *B* (the next instruction to be executed belongs to some method $B.m \in \mathscr{B}_B$) such that invocation has not occurred so far. The proof proceeds showing that σ cannot be extended with the unauthorized invocation, considering the taxonomy of the JCVM instructions we defined in Table 2.

Case I. The next instruction in the execution is one of the type I. Obviously this instruction cannot invoke a method or produce a context switch.

Case II. The next instruction is one of the type II. This instruction can produce a context switch only to the JCRE context, upon throwing an exception. The method *A.s* cannot be invoked.

Case III. Type III instructions cannot produce a context switch, because the execution flow only changes within the same method of *B* that is currently executed. The method *A.s* cannot be invoked.

Case IV. Type IV instructions are return instructions, they cannot invoke a new method and can only switch context to *A*'s context in case *A* was already in the execution stack. Method *A.s* could be invoked in the latter case, but not from *B*'s context (otherwise the illegal invocation would have occurred earlier in σ).

Case V. Type V instructions can produce a context switch, but cannot invoke a method. In this case, the context can only be switched to the JCRE context.

Case VI. The next instruction is an invocation instruction (type VI). These instructions (except for the invokestatic instruction) expect to find an object on the stack and invoke a corresponding method of this object. The method *A.s* can be invoked if *B* has a reference to the object ObjRef of *A* that implements *A.s*. The JCVM does not check correctness of the object ownership upon execution of the invocation instructions, but does this during the casting instructions execution (instructions checkcast and instanceof).

We now demonstrate that *B* cannot maliciously cast an object of *A* into its own object or an object from a trusted third party *C*. The type checking rules for the casting instructions require that the received object is cast into a compatible type [(Classic Edition, 2011), Sec.7.5 of the JCVM specification] and, specifically, if the object of another applet *A* does not implement a Shareable interface, it cannot be accessed for casting at all [(Classic Edition, 2011), Sec.6.2.8 of the JCRE specification], because a run-time exception will be thrown.

Type compatibility is verified by the casting instructions, and an object of *A* implementing a Shareable interface SI_A can be cast only into the same interface SI_A or its superinterface. Therefore, an attempt of casting into *B*'s own (or third-party) interface or class will result in a run-time exception and the JCRE will halt *B*'s execution. If *B* will cast an object of *A* into the JCRE's own type (such as Shareable), the object will be accessible, but it will not be possible to invoke the method *A.s* from this object.

We now reason by the invocation instructions. Further the instruction operands are written in angular brackets and the relevant stack contents in square brackets.

Case VI-invokeinterface. The next instruction is invokeinterface ⟨nargs, id_{CP}, t_m⟩ [ObjRef], where id_{CP} is an index into the Constant Pool of the currently executed application *B* (the item at this index is a reference to an interface); and t_m is a token identifier of a method of this interface. This interface can be defined in the application *B* (then the Constant Pool structure at the index id_{CP} is a pointer to the Class component of *B* and the high bit of this structure is 0) or can be an imported interface (the high bit of the pointed Constant Pool structure is 1). In the latter case the Constant Pool structure contains the token identifier t_I of the target interface and an index id_{Import} at the Import component of *B*, where the structure at this index is the AID AID_A of the package *A* providing the interface.

ObjRef references the object whose method will be finally invoked (the token t_m identifies it). If id_{CP} references an externally defined interface, then ObjRef references an object belonging to a context different from the one of *B*; if id_{CP} references an internally defined interface, then ObjRef belongs to *B*'s context (Proposition Appendix B.1).

If id_{CP} references an internally defined interface, the method invoked upon execution of the invokeinterface instruction is *B*'s own method. So we only need to consider the case when id_{CP} references an external interface. The JCRE firewall will allow to invoke a method across contexts if and only if the invoked interface method belongs to the JCRE or to a Shareable interface, as defined in [(Classic Edition, 2011), Sec.6.2.8 of the JCRE specification]. Therefore, $A.s = \langle AID_A, t_I, t_m \rangle$ is a service of *A*; and no other method of *A*

(not from a Shareable interface) can be invoked by the `invokeinterface` opcode.

The `PolicyChecker` verifies (Sec. 5.2, line 5.2 of the `PolicyChecker` algorithm) that for all services $A.s_1$ such that $A.s_1 \in \texttt{Calls}_B$ and $A.s_1 \in \texttt{Provides}_A$ there will be the corresponding service authorization present in $\texttt{sec.rules}_A$: $(A.s_1, B) \in \texttt{sec.rules}_A$. Therefore, either (a) $A.s \notin \texttt{Calls}_B$ or (b) $A.s \notin \texttt{Provides}_A$.

(a) Assume $A.s \notin \texttt{Calls}_B$. This means, $\texttt{Contract}_B$ is not faithful: B actually invokes $A.s$, but this is not described in the contract.

Upon validation of B the `ClaimChecker` has retrieved the offsets to each method of the B's CAP file (lines 35–39 of Alg. Appendix A.1), including the offset to the method $B.m$, because the CAP file specification requires that each method present in the CAP file has a valid offset stored in the Descriptor component.

For each retrieved method the `ClaimChecker` parses the full set of instructions of this method (lines 41–43 of Alg. Appendix A.1). As $\texttt{invokeinterface} \in \mathscr{B}_{B.m}$, thus the `ClaimChecker` has found it (line 44) and retrieved the operands $\texttt{id}_{CP}$ and $\texttt{t}_m$ (line 45). For a successful context switch the JCVM specification requires that the high bit of the structure at the index $\texttt{id}_{CP}$ within the Constant Pool component of B is equal to 1 (checked on the line 47), $\texttt{CP}_B[\texttt{id}_{CP}] = \langle \texttt{id}_{\text{Import}}{}^A, \texttt{t}_I \rangle$ and the $\texttt{Import}_B[\texttt{id}_{\text{Import}}{}^A] = \texttt{AID}_A$.

For the obtained element $\langle \texttt{id}_{\text{Import}}{}^A, \texttt{t}_I, \texttt{t}_m \rangle$ (lines 48–49 of the algorithm) the `ClaimChecker` matches it with an element of `TempBufferCalls[]` (line 50). However, the $\texttt{Calls}_B$ set is transformed by the `ClaimChecker` into the form $\langle \text{local\_pack\_id}, \texttt{t}_I, \texttt{t}_m \rangle$ (line 33). Thus if $\langle \texttt{AID}_A, \texttt{t}_I, \texttt{t}_m \rangle \notin \texttt{Calls}_B$, then there is no element $\langle \text{local\_pack\_id}_A, \texttt{t}_I, \texttt{t}_m \rangle$ in `TempBufferCalls[]`, where $\texttt{local\_pack\_id}_A$ is an index within the Import component of B such that $\texttt{Import}_B[\texttt{local\_pack\_id}_A] = \texttt{AID}_A$. However, the `ClaimChecker` has verified that $\langle \texttt{id}_{\text{Import}}{}^A, \texttt{t}_I, \texttt{t}_m \rangle \in \texttt{TempBufferCalls[]}$ and $\texttt{Import}_B[\texttt{id}_{\text{Import}}{}^A] = \texttt{AID}_A$. We have come to a contradiction of the construction of the `ClaimChecker` with the assumption that $A.s \notin \texttt{Calls}_B$.

(b) Assume $A.s \notin \texttt{Provides}_A$. Since the service is actually invoked, $A.s \in \texttt{shareable}_A$. As A is a deployed application, it was validated by the `ClaimChecker` and the `PolicyChecker`. Notice, that the set $\texttt{Provides}_A$ could not have been updated through the $\texttt{AppPolicy}_A$ update. Therefore, $\texttt{Contract}_A$ presented at the deployment is unfaithful: there is a provided service in the code which was not declared in the $\texttt{Provides}_A$ set. Thus $\langle \texttt{AID}_A, \texttt{t}_I, \texttt{t}_m \rangle \in \texttt{shareable}_A$, but $\langle \texttt{AID}_A, \texttt{t}_I, \texttt{t}_m \rangle \notin \texttt{Provides}_A$.

All shareable interfaces are declared in the Export file and the Export component of the CAP file. Therefore, the `ClaimChecker` during validation of A parses all interfaces declared in the CAP file of A (lines 5–6) and checks with the Export component if the interface is exported. Thus the `ClaimChecker` successfully identifies all shareable interfaces (lines 9–12), and for each of these interfaces it goes through the declared method tokens matching them with the $\texttt{Provides}_A$ set (lines 8–22). By definition of the $\texttt{shareable}_A$ and by construction of the `ClaimChecker` (in compliance with the JCRE specifications), $\texttt{shareable}_A \subseteq \texttt{Provides}_A$. Notice that if $A.s \notin \texttt{shareable}_A$, then it cannot be actually invoked.

Thus, if `invokeinterface` is the next executed instruction in the context of B and the service $\langle \texttt{AID}_A, \texttt{t}_I, \texttt{t}_m \rangle$ of applet A is invoked, then B was authorized to invoke it in $\texttt{sec.rules}_A$.

Case VI-invokespecial. The next instruction is `invokespecial` $\langle \texttt{id}_{CP} \rangle$ `[ObjRef]`. According to the JCRE specification the object reference `ObjRef` on the stack cannot belong to another context when executing this instruction. Therefore only B's own method can be invoked.

Case VI-invokestatic. The next instruction is `invokestatic`. This instruction accesses a static method that belongs to a class, and not an instance. Classes do not have contexts, as objects do; public static fields and methods are accessible from any context [(Classic Edition, 2011), Sec.6.2 of the JCRE specification]. Therefore, if B was able to invoke a static method of A, the JCRE allows it (no context switch happens, the invoked method belongs to the current context of package B).

Case VI-invokevirtual. The next instruction is `invokevirtual` $\langle \texttt{id}_{CP} \rangle$ `[ObjRef]`. If `ObjRef` references an object from another context, the firewall will allow the invocation if and only if `ObjRef` belongs to the JCRE [(Classic Edition, 2011), Sec. 6.2.8 of the JCRE specification]. Thus upon execution of this instruction B can only invoke its own method or a JCRE method, but cannot invoke methods of another applications.

So, for all JCVM instructions B cannot illegally invoke a method of another application A. The last case is if A used to authorize B to invoke $A.s$ and B was deployed legally, but at some point $\texttt{AppPolicy}_A$ was updated to remove this authorization. This update could have been executed if and only if $A.s \notin \texttt{Calls}_B$, as defined in line 16 of Algorithm 5.2. Again, by construction of the `ClaimChecker`, B cannot invoke $A.s$ unless this is declared in $\texttt{Calls}_B$. Therefore, A cannot remove an authorization until B is removed.

REFERENCES

Akram RN, Markantonikas K, Mayes K. A paradigm shift in the smart card ownership model. In: Proc. of ICCSA 2010, LNCS 6019. Springer-Verlag; 2010.

Akram RN, Markantonakis K, Mayes K. Application-binding protocol in the user centric smart card ownership model. In: Proc. of ACISP-2012, LNCS 6812. Springer-Verlag; 2011. p. 208–25.

Akram RN, Markantonakis K, Mayes K. Coopetitive architecture to support a dynamic and scalable NFC based mobile services architecture. In: Proc. of ICICS-2012, LNCS 7618. Springer-Verlag; 2012. p. 214–27.

Avvenuti M, Bernardeschi C, De Francesco N, Masci P. JCSI: a tool for checking secure information flow in Java Card applications. Journal of Systems and Software 2012;85(11):2479–93.

Barbu G, Duc G, Hoogvorst P. Java card operand stack: fault attacks, combined attacks and countermeasures. In: Proc. of CARDIS-11, LNCS 7079. Springer-Verlag; 2011. p. 297–313.

Barbu G, Andouard P, Giraud C. Dynamic fault injection countermeasure: a new conception of Java Card security. In: Proc. of CARDIS-2012. Springer-Verlag; 2012a.

Barbu G, Hoogvorst P, Duc G. Application-replay attack on Java Cards: when the garbage collector gets confused. In: Proc. of ESSOS-2012, LNCS 7159. Springer-Verlag; 2012b. p. 1–13.

Barthe G, Burdy L, Charles J, Gregoire B, Huisman M, Lanet J-L, et al. JACKA tool for validation of security and behaviour of Java applications. In: Proc. of FMCO-2006, LNCS 4709. Springer-Verlag; 2006. p. 152–74.

Bieber P, Cazin J, Wiels V, Zanon G, Girard P, Lanet J-L. Checking secure interactions of smart card applets: extended version. In: JCS, vol. 10(4). IOS Press; 2002. p. 369–98.

Bielova N, Dragoni N, Massacci F, Naliuka K, Siahaan I. Matching in Security-by-Contract for mobile code. In: JLAP, vol. 78(5). Elsevier; 2009. p. 340–58.

Blasing T, Batyuk L, Schmidt AD, Camtepe SA, Albayrak S. An Android Application Sandbox system for suspicious software detection. In: Proc. of MALWARE'10 2010. p. 55–62.

Bouffard G, Lanet J-L. The next smart card nightmare. In: Cryptography and security: from theory to applications, LNCS 6805. Springer-Verlag; 2012. p. 405–24.

Bouffard G, Iguchi-Cartigny J, Lanet JL. Combined software and hardware attacks on the Java Card control flow. In: Proc. of CARDIS-2011, LNCS 7079. Springer-Verlag; 2011a. p. 283–96.

Bouffard G, Lanet J-L, Machemie J, Poichotte J, Wary J. Evaluation of the ability to transform SIM applications into hostile applications. In: Proc. of CARDIS-11, LNCS 7079. Springer-Verlag; 2011b. p. 1–17.

Bugiel S, Davi L, Dmitrienko A, Fischer T, Sadeghi A-R, Shastry B. Towards taming privilege-escalation attacks on Android. In: Proc. of NDSS'2012 2012.

Chin E, Felt AP, Greenwood K, Wagner D. Analyzing inter-application communication in Android. In: Proc. of MobySys'2011. ACM; 2011. p. 239–52.

Desmet L, Joosen W, Massacci F, Philippaerts P, Piessens F, Siahaan I, et al. Security-by-Contract on the .NET platform. In: Information security technical report, vol. 13(1). Elsevier; 2008. p. 25–32.

Dragoni N, Gadyatskaya O, Massacci F. Supporting software evolution for open smart cards by Security-by-Contract. In: Dependability and computer engineering: concepts for software-intensive systems. IGI Global; 2001. p. 285–305.

Dragoni N, Lostal E, Gadyatskaya O, Massacci F, Paci F. A load time policy checker for open multi-application smart cards. In: Proc. of POLICY-11. IEEE; 2011. p. 153–6.

Dubreuil J, Bouffard G, Lanet J-L, Cartigny J. Type classification against fault enabled mutant in Java based smart card. In: Proc. of ARES-2012. IEEE; 2012. p. 551–6.

Ekberg J-E, Asokan N, Kostiainen K, Rantala A. Scheduling execution of credentials in constrained secure environments. In: Proc. of ACM STC'2008. ACM; 2008. p. 61–70.

Enck W, Ongtang M, McDaniel P. On lightweight mobile phone application certification. In: Proc. of ACM CCS 2009. ACM; 2009. p. 235–45.

Enck W, Gilbert P, Chun B, Cox L, Jung J, McDaniel P, et al. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proc. OSDI-2010. USENIX; 2010. p. 1–6.

Enck W, Octeau D, McDaniel P, Chaudhuri S. A study of Android application security. In: Proc. of the 20th USENIX security. USENIX; 2011.

Felt AP, Wang HJ, Moshchuk A, Hanna S, Chin E. Permission re-delegation: attacks and defenses. In: Proc. of the 20th USENIX security. USENIX; 2011.

Fontaine A, Hym S, Simplot-Ryl I. Verifiable control flow policies for Java bytecode. In: Proc. of FAST-2011. Springer-Verlag; 2011a. p. 115–30.

Fontaine A, Hym S, Simplot-Ryl I. On-device control flow verification for Java programs. In: Proc. of ESSOS'2011, LNCS 6542. Springer-Verlag; 2011b. p. 43–57.

Gadyatskaya O, Lostal E, Massacci F. Load time security verification. In: Proc. of ICISS'2011, LNCS 7093. Springer-Verlag; 2011. p. 250–64.

Gadyatskaya O, Massacci F, Paci F, Stankevich S. Java card architecture for autonomous yet secure evolution of smart cards applications. In: Proc. of NordSec'2010, LNCS 7127. Springer-Verlag; 2012. p. 187–92.

Ghindici D, Simplot-Ryl I. On practical information flow policies for Java-enabled multiapplication smart cards. In: Proc. of CARDIS-2008, LNCS 5189. Springer-Verlag; 2008. p. 32–7.

Girard P. Which security policy for multiapplication smart cards?. In: Proc. of USENIX WOST-1999. USENIX; 1999.

Girard P, Lanet J-L. New security issues raised by open cards. In: Information security technical report, vol. 4(2). Elsevier; 1999. p. 19–27.

GlobalPlatform Inc.. GlobalPlatform card specification V.2.2.1; 2011.

Huisman M, Gurov D, Sprenger C, Chugunov G. Checking absence of illicit applet interactions: a case study. In: Proc. of FASE'04, LNCS 2984. Springer-Verlag; 2004. p. 84–98.

Kostiainen K, Ekberg J-E, Asokan N, Rantala A. On-board credentials with open provisioning. In: Proc. of ASIACCS'2009. ACM; 2009. p. 104–15.

Lackner M, Berlach R, Loining J, Weiss R, Steger C. Towards the hardware accelerated defensive virtual machine – type and bound protection. In: Proc. of CARDIS-2012. Springer-Verlag; 2012.

Langer J, Oyrer A. Secure element development. In: NFC forum spotlight for developers; 2009.

Leng X. Smart card applications and security. In: Information security technical report, vol. 14(2). Elsevier; 2009. p. 36–45.

Li P, Zdancewic S. Advanced control flow in Java card programming. In: SIGPLAN Not., vol. 39(7). ACM; 2004. p. 165–74.

Markantonakis K, Tunstall M, Hancke G, Askoxylakis I, Mayes K. Attacking smart card systems: theory and practice. In: Information security technical report, vol. 14(2). Elsevier; 2009. p. 46–56.

Montgomery M, Krishna K. Secure object sharing in Java Card. In: Proc. of WOST'99. USENIX; 1999.

Mostowski W, Poll E. Malicious code on Java Card smart cards: attacks and countermeasures. In: Proc. of CARDIS-2008, LNCS 5189. Springer-Verlag; 2008. p. 1–16.

Mostowski W, Pan J, Akkiraju S, de Vink E, Poll E, den Hartog J. A comparison of Java Cards: state-of-affairs 2006. In: CS-Report CSR 07-06, TU Eindhoven 2007.

Narasamdya I, Perin M. Certification of smart-card applications in common criteria. In: Proc. of SAC'09. ACM; 2009. p. 601–8.

Ongtang M, McLaughlin S, Enck W, McDaniel P. Semantically rich application-centric security in Android. In: Proc. of ACSAC'2009 2009. p. 340–9.

ORACLE. Java Card 3 Platform. Virtual machine and run-time environment specification, Classic Edition. Version 3.0.4.; 2011.

Philippaerts P, Vogels F, Smans J, Jacobs B, Piessens F. The Belgian electronic identity card: a verification case study. In: Automated verification of critical systemsElectr. Commu. of the EASST, vol. 76; 2011.

Roland M, Langer J, Scharinger J. Practical attack scenarios on secure element-enabled mobile devices. In: Proc. of NFC-2012. IEEE; 2012. p. 19–24.

Sauveron D. Multiapplication smart card: towards an open smart card?. In: Information security technical report, vol. 14(2). Elsevier; 2009. p. 70–8.

Schellhorn G, Reif W, Schairer A, Karger P, Austel V, Toll D. Verification of a formal security model for multiapplicative smart cards. In: Proc. of ESORICS'00, LNCS 1895. Springer-Verlag; 2000.

Souza da Costa U, Martins Moreira A, Musicante MA, Souza Neto PA. JCML: a specification language for the runtime verification of Java Card programs. In: Science of computer programming, vol. 77(4). Elsevier; 2012. p. 533–50.

SUN Microsystems. Runtime environment and virtual machine specificationsIn Java Card™ platform, V.2.2.2; 2006.

SUN Microsystems. The Java Card 3 platform; 2008. White paper.

SUN Microsystems. Java Card 3 Platform. Virtual machine, run-time environment and Java servlet for the Java Card platform specification, Connected Edition. Version 3.0.1; 2009.

Xu R, Saidi H, Anderson R. Aurasium: practical policy enforcement for Android applications. In: Proc. of USENIX security 2012.

Olga Gadyatskaya received a Ph.D. in Mathematics in 2008 at the Novosibirsk State University. From 2007 to 2008 she worked as a researcher at the Institute of Computational Mathematics and Mathematical Geophysics (Novosibirsk). Since 2009 she joined Department of Information Engineering and Computer Science of the University of Trento as a post-doctoral research fellow. Her research interests include security policies and load time verification approaches for smart cards and mobile devices.

Fabio Massacci is full professor at the University of Trento. He received his M.Eng. in 1993 and Ph.D. in Computer Science and Engineering at the University of Rome "La Sapienza" in 1998. He worked in Cambridge University (UK), the University of Siena and IRIT Toulouse (FR). His research interests are in automated reasoning at the crossroads between requirements engineering, computer security and formal methods. Currently he is actively working on industry level security engineering methodologies and is the coordinator of the EU SECONOMICS project on Security Economics.

Quang-Huy Nguyen holds a PhD in Computer Science since 2002. He enjoyed several post-doc stays at INRIA before joining Gemalto as a research engineer. Since 2011, he is a senior security consultant in Trusted Labs. His field of research covers various aspects of formal methods from both theoretical and applicative points of view, in particular, the use of these methods in computer security.

Boutheina Chetali received a Ph.D in Computer Science from the University of Henri Poincare & INRIA-Lorraine in 1996. After 10 years as R&D Manager in Gemalto Technology & Innovation, in 2010 she joined Trusted labs as the head of R&D. She is a member of the Java Card Forum security group and the GlobalPlatform security group.