Department of Information Engineering and Computer Science DISI

UNIVERSITY OF TRENTO - Italy

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

LOAD TIME CODE VALIDATION FOR MOBILE PHONE JAVA CARDS

Olga Gadyatskaya, Fabio Massacci, Quang-Huy Nguyen and Boutheina Chetali

October 2012

Technical Report # DISI-12-025
Version 1.0

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*

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 and 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 formal security model of the approach, its overall architecture and the implementation footprint which can fit on a real secure element. We also report the lessons learned and the intricacies of integrating a research prototype with a protected loader of the manufacturer.

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

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 as shown in Fig. 1.

A common assumption is that only few and limited applications will be loaded on the secure element in Fig.1 but this is no longer the case: the usage of trusted elements evolves quickly following the trends of smartphone markets [3]. 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 [4] was recently proposed.

From a security perspective it is important that the applications are confined (the Java Card firewall does precisely that) 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[2] and VISA pushes its payment SIM application payWave[3], they may want to collaborate. Therefore,

*Corresponding author
Email addresses: `olga.gadyatskaya@unitn.it` (Olga Gadyatskaya), `fabio.massacci@unitn.it` (Fabio Massacci), `quang-huy.nguyen@trusted-labs.com` (Quang-Huy Nguyen), `boutheina.chetali@trusted-labs.com` (Boutheina Chetali)

[1]GlobalPlatform^{TM} is a standard set of specifications for card contents management [1].

[2]`http://www.nfctimes.com/news/german-transit-authorities-launch-joint-nfc-ticketing-service`, accessed on the web in July 2012

[3]`http://www.visaeurope.com/en/cardholders/visa_paywave.aspx`, accessed on the web in July 2012

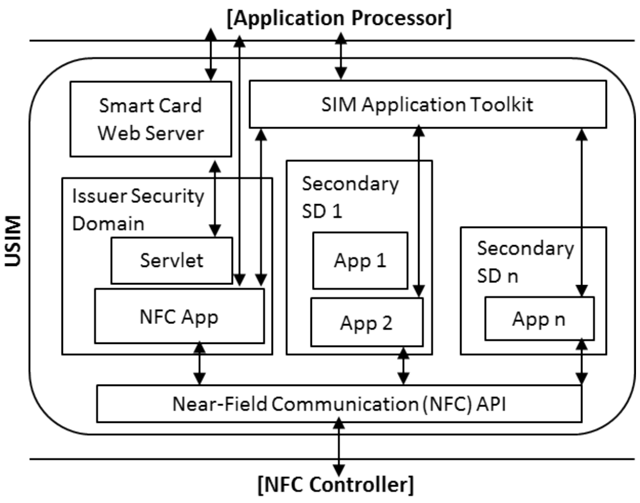


Figure 1: (U)SIM as a secure element [2].

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 [5, Sec. 6.2.4 of the JC Run-time Environment specification]. A Shareable interface method (or *service* for short) is just a Java method that can be called through the firewall. Unfortunately, once a service can be called, it can be called by everybody, unless access control checks are embedded in the application code. Consequently, the only way to add or remove possible callers is to re-install the application. In many cases it is not possible to remove an application referenced by other applications on the card. So, even if we just want to add the possibility of being called by another application, we need first to delete all other calling applications, then re-install the updated application, and then re-install all callers again. Separation of access control to an applet's service and the implementation of the service the applet provides is a desirable feature: when considering multi-tenancy, applet developers want to be able to restrict access to their services in a declarative 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 push the application validation scenario onto the secure element itself (the (U)SIM card in Fig. 1 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 application identifiers (AIDs), even if at any time we only have few of them installed;
- applications' bytecode should be *validated by the card itself* to respect the 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);
- Third-party 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 the manifest of the applications and use the phone normal processor for checking [6, 7]. Other approaches allow to check interactions at run-time requiring VM/platform modifications [8, 9] or suggest to check interactions off-device [10].

So far for smart cards the combination of all elements has not been achieved, and our new contribution is to achieve it. In the smart card world interactions among the applications can be certified, but then the card has to be locked. For example, the TaiwanMoney Card (Taiwan) based on the Multos technology combines a Mondex payment application with a transport application [11]. 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. [12]), but they cannot work in practice for the OTA loading, because an independent verification authority is very expensive, especially if one needs to negotiate all policy updates; and full formal verification cannot be ported to the device itself because of the computational constraints.

Our Contribution. Our target is not to achieve more security than the current methods of embedding access control checks in the application code or off-line bytecode validation. What we want is to achieve the same security while allowing the flexibility of OTA updates on a very restricted platform. Our proposal allows to address the currently needed 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 and are sufficient for the intended applications.

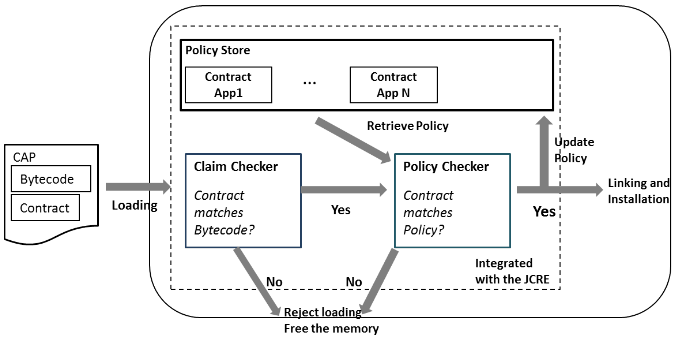
In this paper we report on the engineering aspects that can achieve all the goals mentioned above along with the constraints of the secure element environment: at most 10KB of memory footprint and at most 1KB of RAM consumed for validation. Our system is able to process applications of sizeable complexity, such as the electronic identity applet [13]. 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 and what useful lessons we have received, from the point of view of both the academic partner and the smart card manufacturer partner. We also present the formal model of a multi-tenant secure element platform based on deployed applications and shared/invoked services and prove 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. The formal model of the secure element platform and the security theorem are presented in §6. We present the final architecture of the framework in §7 and detail the on-card policy management in §8. Then we discuss the performance (§9) and security (§10) of our solution. We overview related work (§11) and then conclude (§12).

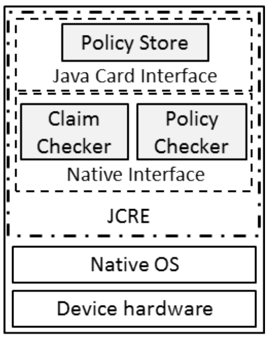
2. Our Approach

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 the CAP file conversion of all applets was correct



(a) The workflow for loading



(b) The stack

Figure 2: The validation workflow for loading and the Java Card stack with the new components

and the bytecode respects the Java abstractions. The interested reader can refer to [14, 3] for information on attacks on the Java Card bytecode and countermeasures against them. These attacks are out of scope of the current paper. We also do not consider application spoofing in the threat model, because the means for protection against this attack are already provided by the GlobalPlatform middleware [1, Sec. 10.4].

Our solution. We propose a system to ensure secure co-existence and sharing of capabilities between multiple applications in a mobile phone multi-tenant Java Card. Our system does so through requiring access control-style 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. 2(a) summarizes the workflow for load time validation and the new components of our framework: the ClaimChecker, the PolicyChecker and the PolicyStore. Fig. 2(b) shows the position of the new components in the stack, for more details see Fig. 7.

Following the strategy to keep the platform secure after each 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). A completely new evolution scenario introduced by the S×C framework is 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. We further discuss the application policy update scenario in more details in Sec. 8.

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

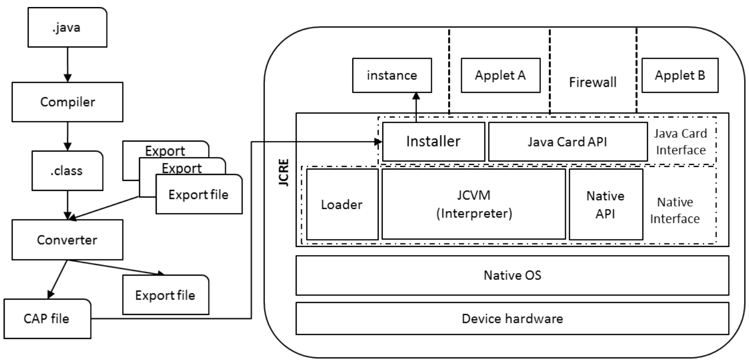


Figure 3: The traditional Java Card architecture, applet development and deployment process.

Table 1: The CAP file components.

Header component //contains the package AID, details if Applet and Export components are present
Directory component //details the size of each component and if Custom components are present
Applet component //contains the suggested applet instance AID and the pointer to the `install`() method; is optional, is not present in library packages
Import component //details the imported packages AIDs and local identifiers
Constant Pool component //a unified constant pool of the package
Class component //describes each class and interface defined in this package
Method component //contains the instruction set of each method in this package
Static Field component //contains the information required to initialize all static fields defined in this package
Reference Location component //contains pointers to the Constant Pool component for the methods of this package
Export component //is optional, lists all static elements in this package that may be imported by classes in other packages
Descriptor component //contains details of elements declared in other components (classes, interfaces and methods)
Debug component //is optional, contains metadata to be used in a debugging environment
Custom component //is optional, one CAP file can contain several Custom components; we use it to deliver contracts

aware of it, and vice-versa.

3. Background on Java Card

Fig. 3 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 [5]. The standard implementation of the JCRE includes components implemented in Java Card (the Java Card Interface in Fig. 3) and components implemented in C (the Native Interface in Fig. 3). 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 predefined format in order to reduce the amount of memory needed for storing an applet; the components are listed in Table 1. For conversion the Export files of imported packages are required, and during conversion the Export file of the converted package is produced.

A package can contain one or multiple applets; an applet is a class extending `javacard.framework.Applet`. A package may not contain any applets, such packages are called *libraries*. 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. An applet may contain a large number of classes.

The deployment includes the following steps. Upon receiving a CAP file, the Installer uses the Loader API to process the file and perform some specified checks [5]. 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 $\mathtt{AID_A}$.

Application Interactions. Applets from different packages are isolated by the JCRE firewall. It 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.

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's Export file is necessary for conversion of the client's package into a CAP file. In a CAP file all methods are referred to by their *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. 3.

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.

Figure 4 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. 4) and the requesting client AID check upon provision of the object implementing the SI (line 32 in Fig. 4).

The access control list (ACL) `clientAIDs`[ ] is currently hard-coded within the Purse code (line 6 in Fig. 4) 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.

```
byte ClientsNumber = 1;
byte [] TransportAIDset =
{0x01,0x02,0x03,0x04,0x05,0x0C,0x0A};
final AID TransportAID = JCSystem.lookupAID
(TransportAIDset,(short)0,(byte)TransportAIDset.length);

//the access control list
AID [] clientAIDs = {TransportAID};
//ACL check implementation
public short authorizedClient(AID clientAID){
   for (short i=0; i<ClientsNumber; i++)
      if (clientAIDs[i].equals(clientAID))
         return i; //clientAIDs is in the ACL
   return -1;
}
//SI definition
public interface PaymentInterface extends Shareable {
   //definition of the payment service
  byte payment(short account_number);
}
public class PaymentClass implements PaymentInterface {
  byte payment_code = 0x08;
  public byte payment(short account_number){
     //implementation of  the service
     AID clientAID = JCSystem.getPreviousContextAID();
     if (authorizedClient(clientAID) == -1) //ACL check
         return (byte) 0x00; //no service is provisioned
     else return payment_code; //provision of the service
  }
}
public PaymentClass PaymentObject;

public Shareable getShareableInterfaceObject(AID oAid,
byte bArg){
 if (authorizedClient(oAid) != -1) // ACL check
    if (bArg == InterfaceDetails)
       //provision of the SI object
       return (Shareable) (PaymentObject);
    else
       ISOException.throwIt(ISO7816.SW_WRONG_DATA);
 return null; //nothing is provisioned
}
```

Figure 4: A sanitized snippet of the Purse applet. It contains the ACL defined in the code, and definition, implementation and provision of the payment service.

For instance, the device holder can further load a new application MessagingApp, which provides him access to the common 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 $\langle \mathtt{AID_s}, \mathtt{t_I}, \mathtt{t_m} \rangle$, where $\mathtt{AID_s}$ is the unique AID of the package that provides the service s, $\mathtt{t_I}$ is the token identifier for the Shareable interface where the service is defined, and $\mathtt{t_m}$ is the token identifier for the method s in the Shareable interface. A *called service* can be identified as a tuple $\langle \mathtt{AID_B}, \mathtt{t_I}^{\mathtt{B}}, \mathtt{t_m}^{\mathtt{B}} \rangle$, where $\mathtt{AID_B}$ 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 $\mathsf{AppClaim_A}$. We denote the provided services set of application A as $\mathsf{Provides_A}$ and the called services set as $\mathsf{Calls_A}$.

```
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
```

Figure 5: Shareable interface and method token identifiers of the Purse's payment service in the Export file.

Table 2: Contracts of Purse and Transport applets.

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

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

$\mathsf{func.rules_A}$ 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 $\mathsf{func.rules_A} \subseteq \mathsf{Calls_A}$. 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* $\mathsf{Contract_A}$ *is a tuple* $\langle \mathsf{AppClaim_A}, \mathsf{AppPolicy_A}\rangle$*, where* $\mathsf{AppClaim_A}$ *is a tuple* $\langle \mathsf{Provides_A}, \mathsf{Calls_A}\rangle$ *and* $\mathsf{AppPolicy_A}$ *is a tuple* $\langle \mathsf{sec.rules_A}, \mathsf{func.rules_A}\rangle$.

Bytecode 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 $\langle \mathtt{AID_A}, \mathtt{t_I}, \mathtt{t_m}\rangle$ provided by A, the token identifier $\mathtt{t_I}$ is listed in the `class_info.token` structure of the corresponding interface declaration in the A's Export file, and the token identifier $\mathtt{t_m}$ is listed in the the corresponding `method_info.token`. Fig. 5 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 Tab. 2 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 [5]. 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.

From Loaded Contracts to the Platform's Policy. The *platform security policy* is composed by the contracts of all currently deployed applets and is stored in the PolicyStore. If a new applet provides a faithful compliant contract, it can be accepted, and its contract is added to the security policy. Otherwise it is rejected and removed from memory. The actual realization of the security policy structures is more complicated due to hardware and operational constraints. We discuss it in §8.

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 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.

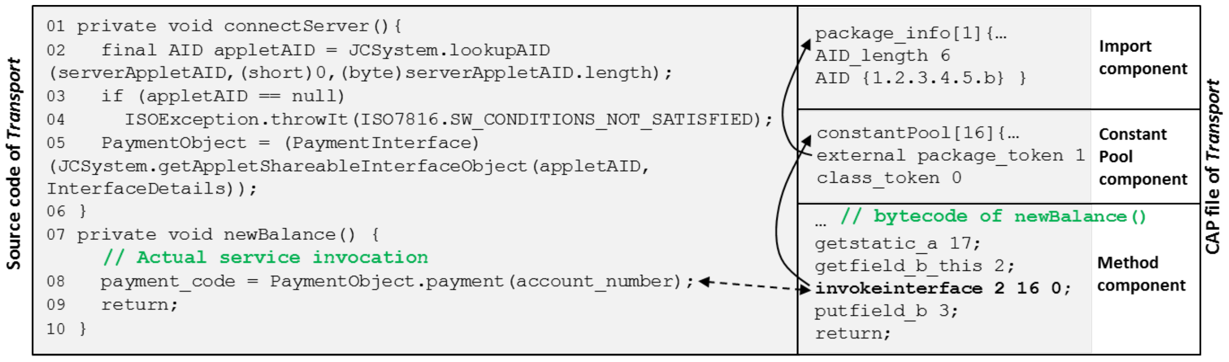
For the called services the algorithm starts again from the beginning of the Descriptor component. It retrieves the Method component offset to the beginning of each method of the current package and stores the offset in the temporary buffer. We note that in case there are too many methods in the CAP 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.

Bytecode realization. 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 $\langle \mathtt{nargs}, \mathtt{id}_{\mathtt{CP}}, \mathtt{t}_{\mathtt{m}} \rangle$ with this instruction and an object reference `ObjRef` on the stack. There `nargs` contains number of arguments of the invoked method (plus 1); $\mathtt{id}_{\mathtt{CP}}$ is an index into the Constant Pool of the current package, the Constant Pool item at $\mathtt{id}_{\mathtt{CP}}$ index should be a reference to the interface type `CONSTANT_Classref`; $\mathtt{t}_{\mathtt{m}}$ is the interface method token of the method to be invoked and `ObjectRef` is the reference to the object to be invoked. The $\mathtt{id}_{\mathtt{CP}}$ index in the Constant Pool component is used to identify the AID of the called package from the Import component.

The process of the called service identification is illustrated in Fig.6, 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

```
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



```
private void connectServer(){
  final AID appletAID = JCSystem.lookupAID
(serverAppletAID,(short)0,(byte)serverAppletAID.length);
  if (appletAID == null)
    ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
  PaymentObject = (PaymentInterface)
(JCSystem.getAppletShareableInterfaceObject(appletAID,
InterfaceDetails));
}
private void newBalance() {
  // Actual service invocation
  payment_code = PaymentObject.payment(account_number);
  return;
}
```

```
package_info[1]{…
AID_length 6
AID {1.2.3.4.5.b} }

constantPool[16]{…
external package_token 1
class_token 0

… // bytecode of newBalance()
getstatic_a 17;
getfield_b_this 2;
invokeinterface 2 16 0;
putfield_b 3;
return;
```

Figure 6: AID, interface and method token identifiers of the invoked payment service in the CAP file of the Transport applet.

corresponds in the bytecode to the instruction `invokeinterface` ⟨2 16 0⟩, which is resolved in the CAP file to invocation of the service ⟨0x01020304050B, 0, 0⟩, that is the Purse's payment service.

Algorithm 5.2 follows the English description. In order to access the components of the CAP files on the secure element we use a very simple 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 [5] and we indicate which component structures belong to in the object-oriented notation. The algorithm also 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 CAPlibrary (subset of the Loader API) is the library we use to access the beginning and the length of CAP components. CAPlibrary also contains 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 a 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.

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 $\mathsf{Calls}_{\mathsf{B}}$ B is authorized by their providers to call them; (2) for all

```
Require: A CAP file, byte TempBufferCalls[ ], byte TempBufferOffsets[ ].
Ensure: True/False, Contract.
 1: //Custom Component: get Contract;
 2: //Descriptor Component: go through the interfaces and the interface methods;
 3: boolean found = False;
 4: short InterfaceToken;
 5: for i = 0 to Descriptor.classes_count do
 6:     if classes[i] has a flag ACC_INTERFACE = 0x40 in the access_flags then
 7:         //Export Component: get tokens of shareable interfaces;
 8:         for j = 0 to Export.classes_count do
 9:             if Export.class_exports[j].class_offset = Descriptor.classes[i].this_class_ref then
10:                 // this is an exported shareable interface;
11:                 found = True;
12:                 InterfaceToken = j;
13:         //check for match with the provided services in the contract;
14:         if found then
15:             for j = 0 to Descriptor.classes[i].method_count do
16:                 found = False;
17:                 for k = 0 to Contract.provides_count do
18:                     if ⟨ InterfaceToken, Descriptor.classes[i].method[j].token ⟩ = Contract.provides[k] then
19:                         found = True;
20:                 if found = False then
21:                     //there is no declared provided service return False
22:         else return False
23: check that all provides_info were found;
24: //Proceed to the called services
25: short PackageToken;
26: //Import Component: get package AIDs of imported packages and their indices;
27: //for each server AID in the Contract check it is imported;
28: for i = 0 to Contract.calls_count do
29:     for j = 0 to Import.count do
30:         if Contract.calls[i].server_AID matches with Import.packages[j].AID then
31:             PackageToken = j;
32:     if some declared called AID is not imported then return False;
33:     store the called services info in TempBufferCalls[ ] in the following format ⟨PackageToken, Contract.calls[i].interface_token,
    Contract.calls[i].service_token⟩;
34: short method_number = 0;
35: //Descriptor Component: go through the classes and obtain the offset of each method, store it in the temporary buffer;
36: for i = 0 to Descriptor.classes_count do
37:     for j = 0 to Descriptor.classes[i].methods_count do
38:         store Descriptor.classes[i].methods[j].method_offset in TempBufferOffsets[ ];
39:         method_number ++;
40: // Method Component: for each method offset parse the bytecode to find called services;
41: for i = 0 to method_number do
42:     CurrentMethod = Method.TempBufferOffsets[i]
43:     parse the bytecode of CurrentMethod
44:     if the invokeinterface instruction is found then
45:         store the operands into LocalInterfaceToken and ServiceToken;
46:         // 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;
47:         if the high bit of ConstantPool.constant_pool[LocalInterfaceToken] equals to 1 then
48:             InterfaceToken = ConstantPool.constant_pool.cp_info[LocalInterfaceToken].class_token;
49:             PackageToken = ConstantPool.constant_pool.cp_info[LocalInterfaceToken].package_token;
50:             if ⟨PackageToken, InterfaceToken, ServiceToken⟩ does not exist in TempBufferCalls[ ] then return False;
51: check that all calls_info were found;
52: // Header Component: get the current package AID; return {True, Header.package.AID, Contract}
```

Algorithm 5.2: The ClaimChecker Algorithm

services from $\mathsf{Provides_B}$ all the applets that can invoke these services are authorized by B; (3) all the services from $\mathsf{func.rules_B}$ are provided.

Algorithm 5.3 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).

```
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 $B \notin \mathsf{sec.rules_A}(\mathsf{Provides_A} \cap \mathsf{Calls_B})$ then return False;
 5:             if $A \notin \mathsf{sec.rules_B}(\mathsf{Provides_B} \cap \mathsf{Calls_A})$ then return False;
 6:         if $\mathsf{func.rules_B} \not\subseteq \bigcup_{A\in\Lambda} \mathsf{Provides_A}$ then return False;
            return True;
 7:     case Removal of already deployed package $B$
 8:         if $\mathsf{Provides_B} \cap \{\bigcup_{A\in\Lambda} \mathsf{func.rules_A}\} \neq \emptyset$ 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 $\mathsf{sec.rules_B}$ return True;
12:             case Removal of an authorization rule for some application $C$ to access a service $B.s$ from $\mathsf{sec.rules_B}$
13:                 if $B.s \in \mathsf{Calls_C}$ then return False;
14:                 else return True;
15:             case Addition of a service $C.s$ to $\mathsf{func.rules_B}$
16:                 if $s \notin \bigcup_{A\in\Lambda} \mathsf{Provides_A}$ then return False;
17:                 else return True;
18:             case Removal of a service $C.s$ from $\mathsf{func.rules_B}$ return True;
```
Algorithm 5.3: The PolicyChecker Algorithm Description.

6. A Formal Model of the Platform

Let Δ_Λ be a domain of package AIDs and Δ_Σ be a domain of services (identified as tuples $\langle \mathtt{AID_A}, \mathtt{t_I}, \mathtt{t_m}\rangle$), where $\mathtt{AID_A} \in \Delta_\Lambda$ 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 $\mathcal{M}_A$ be a set of methods of this CAP file. A method $m \in \mathcal{M}_A$ is defined by the set of its instructions. Let $\mathcal{B}_m$ be the set of opcodes of the method m. Let $\mathcal{B}_A = \bigcup_{m\in\mathcal{M}_A} \mathcal{B}_m$ 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 $\mathtt{ConstPool_A}$.

Definition 6.1 (Application). *On the secure element platform a deployed application A is a tuple $\langle \mathtt{AID_A}, \mathcal{B}_\mathtt{A}, \mathtt{ConstPool_A}\rangle$.*

For applet A we denote as $\mathsf{shareable_A} \subset \{\mathtt{AID_A}\} \times \mathbb{N} \times \mathbb{N}$ the set of services provided by this applet. In practice, for a package A we define $\mathsf{shareable_A}$ as a set of meaningful shared services. Namely, for each service $s = \langle \mathtt{AID_A}, \mathtt{t_I}, \mathtt{t_m}\rangle$ such that $\mathtt{t_I}=$ `export_file.export_classes[i].token` and $\mathtt{t_m}=$ `export_file.export_classes[i].`– `methods[j].token` s belongs to $\mathsf{shareable_A}$ (the structures in this definition represent the contents of the Export file of the package A). If $\mathtt{t_I}$ or $\mathtt{t_m}$ are not meaningful token identifiers (there is no structure with the value $\mathtt{t_I}=$ `export_file.export_classes[i].token` or there is no method with the corresponding $\mathtt{t_m}$ token defined for this interface in the Export file), then $\langle \mathtt{AID_A}, \mathtt{t_I}\mathtt{t_m}\rangle \notin \mathsf{shareable_A}$. The Export files, however, are not delivered on the card with the CAP files. Thus the ClaimChecker algorithm relies on the CAP file specifications [5] to collect the set of provided services from the CAP file itself.

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

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

The Taxonomy of the JCVM Instructions. The JCVM specification v. 2.2 defines 140 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

Table 3: 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`.

context upon execution of these instructions. In Table 3 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.

Proposition 6.1. *When the instruction* `invokeinterface` $\langle$`AID`, $\mathtt{id_{CP}}$, $\mathtt{t_m}\rangle$ [`ObjRef`] *is executed, if the* CP[$\mathtt{id_{CP}}$] *item is an externally defined interface, then* `ObjRef` *references an object belonging to another context. If the* CP[$\mathtt{id_{CP}}$] *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` $\langle$`nargs`, $\mathtt{id_{CP}}$, $\mathtt{t_m}\rangle$ [`ObjRef`] the CP[$\mathtt{id_{CP}}$] item refers to an externally defined interface; 2) upon execution of `invokeinterface` $\langle$`nargs`, $\mathtt{id_{CP}}$, $\mathtt{t_m}\rangle$ [`ObjRef`] the CP[$\mathtt{id_{CP}}$] item refers to an internally defined interface; 3) upon execution of `invokeinterface` $\langle$`nargs`, $\mathtt{id_{CP}}$, $\mathtt{t_m}\rangle$ [`ObjRef`] the `ObjRef` object reference is incompliant with the interface resolved from $\mathtt{id_{CP}}$.

Let us assume `invokeinterface` $\langle$`nargs`, $\mathtt{id_{CP}}$, $\mathtt{t_m}\rangle$ [`ObjRef`] is executed and the CP[$\mathtt{id_{CP}}$] 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[$\mathtt{id_{CP}}$] 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` $\langle$`nargs`, $\mathtt{id_{CP}}$, $\mathtt{t_m}\rangle$ [`ObjRef`] is executed and the CP[$\mathtt{id_{CP}}$] 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[$\mathtt{id_{CP}}$] 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` $\langle\mathtt{id_{CP}}\rangle$ [`ObjRef`] instruction, where CP[$\mathtt{id_{CP}}$] item is a reference to an interface type, requires that the object referenced by `ObjRef` implements the interface type referenced by CP[$\mathtt{id_{CP}}$], otherwise the `ClassCastException` is thrown upon execution of the casting instruction. □

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.*

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.

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 $B.m \in \mathcal{B}_B$) 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 3.

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 [5, 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 [5, Sec.6.2.8.8-6.2.8.10 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` $\langle$`nargs` $\texttt{id}_{\texttt{CP}}$ $\texttt{t}_{\texttt{m}}\rangle$ [`ObjRef`], where $\texttt{id}_{\texttt{CP}}$ 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 $\texttt{t}_{\texttt{m}}$ 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 $\texttt{id}_{\texttt{CP}}$ 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 $\texttt{t}_{\texttt{I}}$ of the target interface and an index $\texttt{id}_{\texttt{Import}}$ at the Import component of B, where the structure at this index is the AID $\texttt{AID}_{\texttt{A}}$ of the package A providing the interface.

`ObjRef` references the object whose method will be finally invoked (the token $\texttt{t}_{\texttt{m}}$ identifies it). If $\texttt{id}_{\texttt{CP}}$ references an externally defined interface, then `ObjRef` references an object belonging to a context different from the one of B; if $\texttt{id}_{\texttt{CP}}$ references an internally defined interface, then `ObjRef` belongs to B's context (Proposition 6.1).

If $\mathtt{id_{CP}}$ 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 $\mathtt{id_{CP}}$ 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 [5, Sec.6.2.8.5-6.2.8.6 of the JCRE specification]. Therefore, $A.s = \langle \mathtt{AID_A}, \mathtt{t_I}, \mathtt{t_m} \rangle$ 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 3 of the PolicyChecker algorithm) that for all services $A.s_1$ such that $A.s_1 \in \mathsf{Calls_B}$ and $A.s_1 \in \mathsf{Provides_A}$ there will be the corresponding service authorization present in $\mathsf{sec.rules_A}$: $(A.s_1, B) \in \mathsf{sec.rules_A}$. Therefore, either (**a**) $A.s \notin \mathsf{Calls_B}$ or (**b**) $A.s \notin \mathsf{Provides_A}$.

(**a**) Assume $A.s \notin \mathsf{Calls_B}$. This means, $\mathsf{Contract_B}$ 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. 5.2), 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. 5.2). As $\mathtt{invokeinterface} \in \mathcal{B}_{B.m}$, thus the ClaimChecker has found it (line 44) and retrieved the operands $\mathtt{id_{CP}}$ and $\mathtt{t_m}$ (line 45). For a successful context switch the JCVM specification requires that the high bit of the structure at the index $\mathtt{id_{CP}}$ within the Constant Pool component of B is equal to 1 (checked on the line 47), $\mathsf{CP_B}[\mathtt{id_{CP}}] = \langle \mathtt{id_{Import}}^{\mathtt{A}}, \mathtt{t_I} \rangle$ and the $\mathtt{Import_B}[\mathtt{id_{Import}}^{A}] = \mathtt{AID_A}$.

For the obtained element $\langle \mathtt{id_{Import}}^{\mathtt{A}}, \mathtt{t_I}, \mathtt{t_m} \rangle$ (lines 48-49 of the algorithm) the ClaimChecker matches it with an element of `TempBufferCalls[ ]` (line 50). However, the $\mathsf{Calls_B}$ set is transformed by the ClaimChecker into the form $\langle \mathit{local\_pack\_id}, \mathtt{t_I}, \mathtt{t_m} \rangle$ (line 33). Thus if $\langle \mathtt{AID_A}, \mathtt{t_I}, \mathtt{t_m} \rangle \notin \mathsf{Calls_B}$, then there is no element $\langle \mathit{local\_pack\_id}_A, \mathtt{t_I}, \mathtt{t_m} \rangle$ in `TempBufferCalls[ ]`, where $\mathit{local\_pack\_id}_A$ is an index within the Import component of B such that $\mathtt{Import_B}[\mathit{local\_pack\_id}_A] = \mathtt{AID_A}$. However, the ClaimChecker has verified that $\langle \mathtt{id_{Import}}^{\mathtt{A}}, \mathtt{t_I}, \mathtt{t_m} \rangle \in \mathtt{TempBufferCalls[\ ]}$ and $\mathtt{Import_B}[\mathtt{id_{Import}}^{\mathtt{A}}] = \mathtt{AID_A}$. We have come to a contradiction of the construction of the ClaimChecker with the assumption that $A.s \notin \mathsf{Calls_B}$.

(**b**) Assume $A.s \notin \mathsf{Provides_A}$. Since the service is actually invoked, $A.s \in \mathsf{shareable_A}$. As A is a deployed application, it was validated by the ClaimChecker and the PolicyChecker. Notice, that the set $\mathsf{Provides_A}$ could not have been updated through the $\mathsf{AppPolicy_A}$ update. Therefore, $\mathsf{Contract_A}$ presented at the deployment is unfaithful: there is a provided service in the code which was not declared in the $\mathsf{Provides_A}$ set. Thus $\langle \mathtt{AID_A}, \mathtt{t_I}, \mathtt{t_m} \rangle \in \mathsf{shareable_A}$, but $\langle \mathtt{AID_A}, \mathtt{t_I}, \mathtt{t_m} \rangle \notin \mathsf{Provides_A}$.

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 $\mathsf{Provides_A}$ set (lines 8-22). By definition of the $\mathsf{shareable_A}$ and by construction of the ClaimChecker (in compliance with the JCRE specifications), $\mathsf{shareable_A} \subseteq \mathsf{Provides_A}$. Notice that if $A.s \notin \mathsf{shareable_A}$, then it cannot be actually invoked.

Thus, if `invokeinterface` is the next executed instruction in the context of B and the service $\langle \mathtt{AID_A}, \mathtt{t_I}, \mathtt{t_m} \rangle$ of applet A is invoked, then B was authorized to invoke it in $\mathsf{sec.rules_A}$.

Case VI-invokespecial. The next instruction is `invokespecial` $\langle \mathtt{id_{CP}} \rangle$ [`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` $\langle \mathtt{id_{CP}} \rangle$ [ ]. The JCRE specification requires that the invoked method belongs to the current context of package B.

Case VI-invokevirtual. The next instruction is `invokevirtual` $\langle \mathtt{id_{CP}} \rangle$ [`ObjRef`]. If `ObjRef` references an object from another context, the firewall will allow the invocation if and only if `ObjRef` belongs to the JCRE [5, Sec. 6.2.8.4,6.2.8.11 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 $\mathsf{AppPolicy_A}$ was updated to remove this authorization. This update could have been executed if and only if $A.s \notin \mathsf{Calls_B}$, as defined in line 16 of Algorithm 5.3. Again, by construction of the ClaimChecker, B cannot invoke $A.s$ unless this is declared in $\mathsf{Calls_B}$. 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[4]). 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 10KB of non-volatile memory for storing the prototype itself and the security policy of the card across sessions, and could not use more than 1KB 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 1KB to 5KB on modern cards). Thus this temporary buffer restriction ensures the highest interoperability.

The S×C *Architecture.* Figure 7 depicts the modified architecture and the changes to the development and the deployment processes of Fig.3, the grey 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 component as an applet exist [15, 16]. 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 SxCInstaller component. The SxCInstaller 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 SxCInstaller) are processed without hinderance. Unfortunately, calls from

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

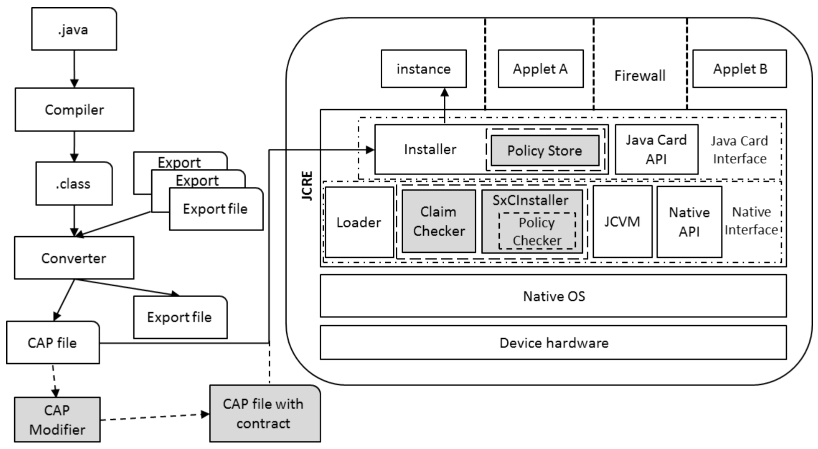


Figure 7: The Java Card architecture and the loading process enhanced with the S×C on-device validation.

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 SxCInstaller, 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 128B and 256B, thus only a very small policy could be maintained (256B 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[5] 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 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[6]; 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.

[5]`http://www.oracle.com/technetwork/java/javacard/overview/index.html`, accessed on the web in July 2012
[6]For requests of the binaries please contact the corresponding author via email.

8. 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. The data structures maintained by the PolicyStore are: Policy, Mapping, MayCallObj and WishListObj. The security policy object Policy is implemented in the fixed bit vectors format, enabling fast contract-policy compliance operations. We use a dedicated Mapping table to maintain correspondence between the loaded applet AIDs and the on-card S×C identifiers and between the deployed service token identifiers $\langle \mathtt{t_I}, \mathtt{t_m} \rangle$ and the on-card service indices.

The MayCallObj object stores authorizations for applets not yet loaded on the card and the WishListObj object stores services that loaded applets can try to invoke, but they are not yet present on the card. These two objects and the Mapping object are space-consuming, because they store the AIDs, but they are used rarely. The Mapping, MayCallObj and WishListObj objects are used for contract transformation into the internal format. For instance, when the Provides set is mapped into the internal format and the services obtain on-card identifiers, the delivered sec.rules set is mapped to the Policy security rules set and the MayCallObj object, depending if the allowed client is loaded.

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 SxCInstaller stored the contract in the on-card format 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, moving authorizations given for this applet to the MayCallObj list and updating the WishListObj structure if necessary.

Bytecode Realization. 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: 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.

The S×C prototype assigns the identifiers independently from the JCRE assignment of the local identifiers in order to avoid dependencies of the S×C prototype on a specific platform implementation.

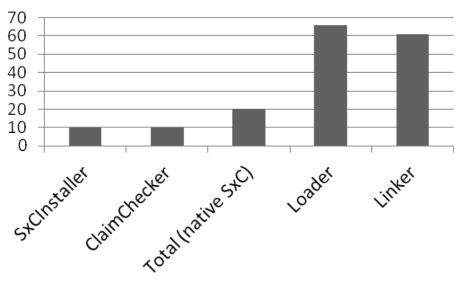
Notice that the PolicyStore only maintains the security policy data structures and performs updates, but the main contract-policy compliance check is executed by the PolicyChecker, which retrieves the policy from the dedicated platform buffer.

The Policy Update. 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()`.

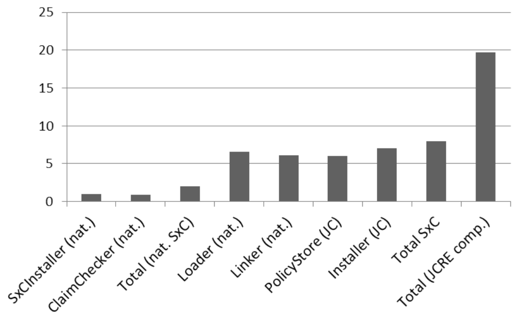
The application provider needs to transmit to the PolicyStore component an APDU (Application Protocol Data Unit) sequence specifying the type of the update to be executed, the AIDs of the applets in question (which applet's policy has to be updated and which is the AID in the security or functional rule). 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.

9. 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 [17] lists memory available on the card as the second card feature, after the supported specifications;



(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).

Figure 8: Sizes of the prototype components and their comparison with the JCRE components sizes

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 4 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. 8(a) 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. 8(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

Table 4: The S×C framework components sizes.

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

Table 5: 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	2.5KB	6	1	66
Transport	2.5KB	5	0	92
EID	11.2KB	81	1	1419
ePurse	4.7KB	16	1	431

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 100B (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, 135B and 255B).

Processed Applets. Table 5 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 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 [13]; 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 10KB is already a big applet, most telecom applets are between 1 and 10 KB.

10. 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 [5]. 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.

- **The JCVM instructions taxonomy defined in Table 3 is meaningful.** 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 [18]) 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 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 formal 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 boost 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: usually sophisticated 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.

11. Related Work

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

Multi-Tenant Platforms: Java Card. Fontaine et al. [19] propose a mechanism to enforce transitive control flow policies on JC. These policies are stronger than the access control policies enforced by our framework, because we capture only direct service invocations. The main limitation is their focus on ad-hoc security domains and not on package AIDs. Security domains are very coarse grained administrative security roles (usually a handful), typically used to delegate installation privileges. As a consequence we can provide a much finer access control list closer to actual practice.

Ghindici et al. [20] propose an information flow verification system for small Java-based devices. The system explores off-device and on-device steps. Off device an applet *certificate* is created (contains information flows within the applet and high/low annotations). On device the certificate is checked in a proof-carrying-code fashion and matched with the information flow policies of other applets. The information flow policies are very expressive and they can be considered for inter-application communications regulation. No practical implementation of the proposed system for Java Card exist. It cannot be implemented for JC2.2 because the latter does not allow custom class loaders, and even implementation for JC3.0 may not be effective due to significant amount of memory required to store the information flow policies.

Our solution targets devices in the field, thus we have developed it for Java Card 2.2. The latest version of Java Card is 3.0 (connected edition), which is not yet significantly deployed in the industry due to cost/resource/IPR constraints. However, JC 2.2 is more difficult for on-device components integration, so our solution also can be ported to JC3.0.

There were investigations [12, 21, 22, 23] of static scenarios, when all applets are known and the composition is analyzed off-device. For example, Avvenuti et al. [24] have developed the JBIFV tool which verifies whether a JC applet respects pre-defined information flow policies. This tool runs off-card, so it assumes an existence of a controlling authority, such as an application market, that can check applets before loading.

The investigation of the Security-by-Contract techniques for JC is carried out in [15, 16, 25] targeting dynamic scenarios when third-party applets can be loaded on the platform. Dragoni et al. [15] and Gadyatskaya et al. [16] 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 [15, 16] 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 [25], 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.

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 the apps at the installation time. For some sensitive permissions (like the GPS sensor access) the device user is prompted.

Enck et al. [6] 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. [7] 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 [26, 27]) 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 [28], 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 [7], richer policies are elaborated, for instance, in [8, 9, 29]. However, the precision comes at the price of run-time overhead, and run-time monitoring is not suitable for resource-constrained secure elements.

Multi-Tenant Platforms: JavaME, .Net. The S×C paradigm was proposed for multi-application mobile devices (JavaME and .NET technologies) [30, 31]. 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 [30]. In case of failure an inlined reference monitor is used [31]. This approach allows to run even potentially dangerous applications in a sandbox environment.

In our scheme for JC 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[7] says "Most rejections are based on the application containing quality issues or 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".

[7]`http://www.apple.com/hotnews/apple-answers-fcc-questions/`, accessed on the Web in July 2012

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. and Kostiainen et al. [32, 33]. 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.

12. 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 propose a formalization for multi-tenant secure element platforms and prove that our framework ensures security.

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 component 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 SxCInstaller 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.

Lessons Learned. We would like to share some thoughts appeared as a result of the work done together by all partners to implement the S×C prototype and to integrate it with a real device. One of the main concerns of the smart card manufacturer partner was minimization of disclosure of the proprietary implementation details. Due to cost and licensing issues a smart card manufacturer has usually few implementations of the JCRE. If an attacker gains any knowledge about the implementation details, he can potentially become more successful in breaking the cards. Since the same implementation is sold as banking cards or identity cards to customers, it is impractical to share any implementation details with anybody, even a research institute.

Our solution for this confidentiality problem was to share the CAPlibrary with some API of the Loader. The academic partner followed the JC specifications for re-implementing this library for testing purposes and directing the prototype.

Unfortunately, the JC specifications were not very precise in some parts and we only could rely on the knowledge of the persons who implemented the platform. This happened, for example, with the APDU buffer usage for the S×C components communication (retrieval and update of the policy). It is not stated explicitly in the specifications that the APDU buffer cannot be used during loading. So only when the actual device integration was done we have found out the communication between the native and the JC components did not work as expected and another solution was required. Also, many optimizations applied on the card to deployed CAP files also had to be communicated to the academic partner. A simple decision such as representing data with one or two bytes means that all addresses after it might be off by one or two bytes, thus derailing correctness. The amount of such optimizations is quite big, thus some of them were initially, inevitably, forgotten.

Overall, we can conclude that it was a very interesting experience of collaboration. We have found a way to share some details of the smart card platform functions while protecting the sensitive implementation details. The successful integration of a research prototype on an actual card was indeed a key result of the joint work.

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.

Acknowledgements. 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.

References

[1] GlobalPlatform Inc., GlobalPlatform Card Specification v.2.2.1 (2011).

[2] J. Langer, A. Oyrer, Secure element development, in: NFC Forum Spotlight for Developers, `http://www.nfc-forum.org/events/oulu_spotlight/2009_09_01_Secure_Element_Programming.pdf`, accessed on the web in July 2012.

[3] G. Bouffard, J. Lanet, J. Machemie, J. Poichotte, J. Wary, Evaluation of the ability to transform SIM applications into hostile applications, in: Proc. of CARDIS-11, Vol. 7079 of LNCS, Springer-Verlag, 2011, pp. 1–17.

[4] Medic Mobile announces the first mobile SIM app for healthcare, `http://medicmobile.org/2011/06/06/medic-mobile-announces-the-first-mobile-sim-app- for-healthcare`, Accessed on the web in July 2012.

[5] Sun, Runtime environment and virtual machine specifications. Java Card^{TM} platform, v.2.2.2, Specification (2006).

[6] W. Enck, M. Ongtang, P. McDaniel, On lightweight mobile phone application certification, in: Proc. of ACM CCS 2009, ACM, 2009, pp. 235–245.

[7] M. Ongtang, S. McLaughlin, W. Enck, P. McDaniel, Semantically rich application-centric security in Android, in: Proc. of ACSAC'2009, 2009, pp. 340–349.

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

[9] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, A. Sheth, TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones, in: Proc. of USENIX OSDI'10, USENIX Association, 2010, pp. 1–6.

[10] E. Chin, A. Felt, K. Greenwood, D. Wagner, Analyzing inter-application communication in Android, in: Proc. of MobySys'2011, ACM, 2011, pp. 239–252.

[11] http://www.taiwanmoney.com.tw (in Chinese). An alternative website: http://en.wikipedia.org/wiki/TaiwanMoney_Card, accessed on the web in July 2012.

[12] P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, J.-L. Lanet, Checking secure interactions of smart card applets: Extended version, J. of Comp. Sec. 10 (4) (2002) 369–398.

[13] P. Philippaerts, F. Vogels, J. Smans, B. Jacobs, F. Piessens, The Belgian electronic identity card: a verification case study, in: Proc. of AVoCS'11.

[14] G. Barbu, G. Duc, P. Hoogvorst, Java Card operand stack: fault attacks, combined attacks and countermeasures, in: Proc. of CARDIS-11, Vol. 7079 of LNCS, Springer-Verlag, 2011, pp. 297–313.

[15] N. Dragoni, E. Lostal, O. Gadyatskaya, F. Massacci, F. Paci, A load time Policy Checker for open multi-application smart cards, in: Proc. of POLICY'11, IEEE Press, pp. 153–156.

[16] O. Gadyatskaya, F. Massacci, F. Paci, S. Stankevich, Java Card architecture for autonomous yet secure evolution of smart cards applications, in: Proc. of NordSec'2010, Vol. 7127 of LNCS, SV, pp. 187–192.

[17] W. Mostowski, J. Pan, S. Akkiraju, E. de Vink, E. Poll, J. den Hartog, A comparison of Java Cards: State-of-affairs 2006, Tech. Rep. CS-Report CSR 07-06, TU Eindhoven (2007).

[18] M. Montgomery, K. Krishna, Secure object sharing in Java Card, in: Proc. of WOST'99, USENIX Association, 1999.

[19] A. Fontaine, S. Hym, I. Simplot-Ryl, On-device control flow verification for Java programs, in: Proc. of ESSOS'2011, Vol. 6542 of LNCS, Springer-Verlag, 2011, pp. 43–57.

[20] D. Ghindici, I. Simplot-Ryl, On practical information flow policies for Java-enabled multiapplication smart cards, in: Proc. of CARDIS'2008, Vol. 5189 of LNCS, Springer-Verlag, 2008, pp. 32–47.

[21] P. Girard, Which security policy for multiapplication smart cards?, in: Proc. of USENIX Workshop on Smartcard Technology, USENIX Association, 1999.

[22] M. Huisman, D. Gurov, C. Sprenger, G. Chugunov, Checking absence of illicit applet interactions: a case study, in: Proc. of FASE'04, Vol. 2984 of LNCS, Springer-Verlag, 2004, pp. 84–98.

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

[24] M. Avvenuti, C. Bernardeschi, N. De Francesco, Java bytecode verification for secure information flow, ACM SIGPLAN Notices 38 (12) (2003) 20–27.

[25] O. Gadyatskaya, E. Lostal, F. Massacci, Load time security verification, in: Proc. of ICISS'2011, Vol. 7093 of LNCS, Springer-Verlag, 2011, pp. 250–264.

[26] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. Camtepe, S. Albayrak, An Android application sandbox system for suspicious software detection, in: Proc. of MALWARE'10, pp. 55–62.

[27] W. Enck, D. Octeau, P. McDaniel, S. Chaudhuri, A study of Android application security, in: Proc. of the 20th USENIX Security, USENIX Association, 2011.

[28] J. Jeon, K. Micinski, J. Vaughan, N. Reddy, Y. Zhu, J. Foster, T. Millstein, Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android, Tech. Rep. CS-TR-5006, Dec. 2011, U. of Maryland, Dept. of Comp. Sc., also presented at BYTECODE2012.

[29] A. Felt, H. Wang, A. Moshchuk, S. Hanna, E. Chin, Permission re-delegation: attacks and defenses, in: Proc. of the 20th USENIX Security, USENIX Association, 2011.

[30] N. Bielova, N. Dragoni, F. Massacci, K. Naliuka, I. Siahaan, Matching in Security-by-Contract for mobile code, J. of Logic and Algebraic Programming 78 (5) (2009) 340–358.

[31] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, D. Vanoverberghe, Security-by-Contract on the .NET platform, Information Security Tech. Rep. 13 (1) (2008) 25 – 32.

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

[33] K. Kostiainen, J.-E. Ekberg, N. Asokan, A. Rantala, On-board credentials with open provisioning, in: Proc. of ASIACCS'2009, ACM, pp. 104–115.