Controlling Application Interactions on the Novel Smart Cards with Security-by-Contract

Olga Gadyatskaya and Fabio Massacci

DISI, University of Trento,
via Sommarive, 14, Povo 0, Trento, Italy, 38123
{name.surname} @unitn.it

Abstract. In this paper we investigate novel use cases for open multi-application smart card platforms. These use cases require a fine-grained access control mechanism to protect the sensitive functionality of on-card applications. We overview the Security-by-Contract approach that validates at load time that the application code respects the interaction policies of other applications already on the card, and discuss how this approach can be used to address the challenging change scenarios in the target use cases.

1 Introduction

The smart card technology supports asynchronous coexistence of multiple applications from different providers on the same chip since a long time ago. However, the actual use cases for such cards have appeared only recently. In this paper we briefly overview two novel use cases for multi-application smart cards, which we explore as illustrative examples. The first use case is the Near Field Communication (NFC)-enabled smartphone, where the (U)SIM card hosts payment, transport and other types of sensitive applications coming from different vendors. The second use case is a smart card-based enhancement of a smart meter system. A telecommunications hub, implemented as a smart card and installed at a house, hosts and manages traditional utility consumption applications, such as gas or electricity consumption applications, and also a set of telecare applications. The telecare applications enable remote monitoring of health status of the inhabitants; they are connected to devices such as weights or a heart rate monitor. This architectural solution was developed within the Hydra Project [12].

In both these scenarios the applications deployed on the multi-application smart cards are quite sensitive, as they may have access to the private data of the device owner or the application provider. However, these applications are not necessarily fully sandboxed. On the contrary, the applications might need to interact with each other on the card in order to provide an enhanced functionality to the device owner. Thus the key challenge for such cards it ensuring that only trusted partners will have access to the shared functionality (called *service* in the smart card jargon).

The existing solutions for control of application interactions on multi-tenant platforms mostly propose to verify of a pre-defined set of applications off-card [2] or enforce the desired policies at run-time [1]. The first approach is not appealing from the business perspective: as the platform is open, each time a new application will be loaded a full offline re-verification will be required. The second approach is simply not suitable for smart cards due to the resource constraints. The approach that is currently adopted by the smart card community is embedding the access control checks into the functional code. Each time the sensitive service is invoked it checks that the caller is authorized to use it. However, this approach suffers from the fact that partial code updates are not available on smart cards; only the full reinstallation is supported by the runtime environment. Therefore each time a new trusted caller needs to be added a full reinstallation of the application will be required.

Recently load time verification was adopted for multi-application smart cards [4, 5, 8, 6, 7]. With this approach the platform is always in a secure state across all possible changes, such as loading of a new application or removal of an old one. In the current paper we overview the Security-by-Contract ($\mathsf{S}\times\mathsf{C}$) approach for load time verification on multi-application Java Cards and identify how the novel multi-application smart card use cases can be handled with this approach.

The paper is structured is follows. We overview the target use cases in Sec. 2 and present the workflows of the $\mathsf{S}\times\mathsf{C}$ approach for each change scenario in Sec. 3. The necessary background on Java Card is presented in Sec. 4, the design details of the framework are summarized in Sec. 5, and the concrete contracts for the identified motivational scenarios are listed in Sec. 6. We overview the related work in Sec. 7 and conclude in Sec. 8.

2 Multi-Application Java Card Use Cases

In this section we present the target use cases recently introduced in the multi-application smart cards domain.

2.1 NFC-Enabled Phones

Currently NFC offerings from various vendors include *payment* applications (the Google wallet[1], PayPass from MasterCard[2], payWave from VISA[3]), *ticketing* applications (Calypso is a set of technical specifications for NFC ticketing; its handbook contains an overview of the NFC ticketing status in various countries[4]) and *entertainment* applications (including NFC tap-triggered messages from Santa Claus[5]).

[1] http://www.google.com/wallet/
[2] http://www.paypass.com/
[3] http://www.paypass.com/
[4] http://www.calypsonet-asso.org/downloads/100324-CalypsoHandbook-11.pdf
[5] http://www.nfcworld.com/2012/12/07/321480/christmas-app-conjures-up-santa-with-an-nfc-tap/

The NFC functionality requires a secure element to store the sensitive NFC credentials, and one of the existing solutions is usage of the (U)SIM card within the phone to store this data. For instance, an NFC-enabled multi-application (U)SIM card, called UpTeq[6], is currently offered by Gemalto. This card is certified by the VISA, MasterCard and Amex payment systems.

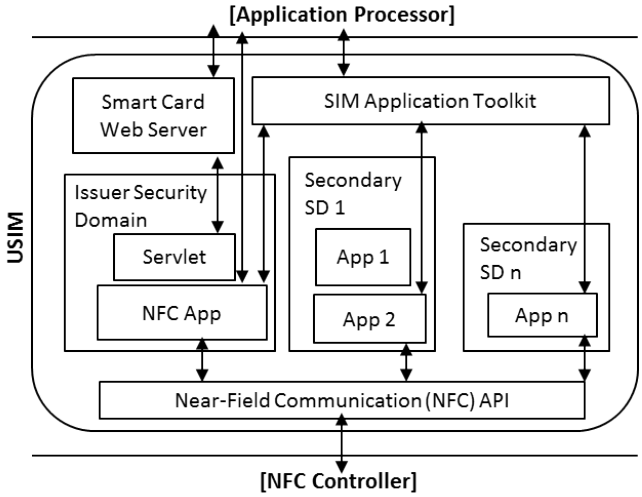


Fig. 1. (U)SIM as a secure element [14].

Figure 1 presents an architecture of a (U)SIM card used as a secure element within an NFC-enabled phone. The NFC controller enables a communication link between the phone and various NFC tags and devices.

Scenario 1: New Application Is Loaded We consider the following scenario of an NFC-enabled smartphone with a (U)SIM-based secure element. The scenario is purely fictional scenario and we use real commercial product names only for the sake of clarity. Two applications (*applets* for short) are already hosted by the (U)SIM card: the payment application payWave from VISA and the ticketing application Touch&Travel from Deutsche Bahn[7]. The phone holder can use the payWave application for executing payment operations in shops, and the Touch&Travel application to pay for train tickets and display ticket barcodes to the phone holder and the train authorities. The VISA consortium and Deutsche Bahn are business partners, and therefore the applications can interact on card: Touch&Travel relies on payWave for the ticket payments. The (U)SIM card is managed by a telecom operator, which has agreements with both VISA and

[6] http://www.gemalto.com/telecom/upteq/index.html
[7] https://www.touchandtravel.de/

Deutsche Bahn; these agreements do not limit the telecom operator in which other applications can be loaded on the card, provided the operator guarantees that only authorized applications will interact with payWave and Touch&Travel.

The phone holder travels from Germany to France, where she installs the public ticketing application Navigo[8] produced by the French public transportation organization STIF. STIF and VISA do not have an agreement, therefore the Navigo application can be recharged only at the metro stations, and not through payWave. However, the telecom operator still has to ensure that Navigo will not try to access payWave directly on the card. In the next sections we will discuss how this can be implemented.

2.2 Communication Hubs within the Smart Grids

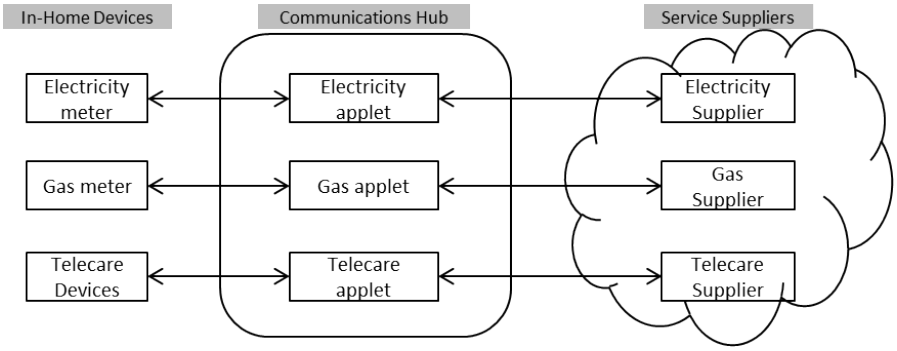


Fig. 2. Extension of a smart metering system with telecare services [12].

Our second motivating use case for multi-application smart cards is the telecommunications hub within the smart metering system proposed by the Hydra project [12]. The project aims to introduce remote care services as an extension to the smart metering system. The architecture of this extension is presented in Fig. 2. Existing utility meters and telecare devices, such as blood pressure monitor or heart-rate monitor, are connected to the smart card, which acts as a proxy between the metering devices and the corresponding utility/telecare providers.

The main idea behind the smart card utilization is privacy of the utility consumption data. The smart metering systems measure the utility consumption at high granularity. By gathering a lot of data points throughout the day the utility companies can learn a lot about the private life of their customers: when they leave to work and come back, when they wake up and go to sleep. In an industrial setting the utility consumption can reveal details of the production

[8] `http://test.navigo.fr/`

process: what machinery is used, or when a new process is adopted [13]. The fact that the utility companies can leak this private data to third parties is even more disturbing [17]. Solutions to this privacy problem in the smart grid have started to emerge, focusing on introducing a mediator for the utility consumption or devising privacy-preserving protocols among the utility provider, the user and the meter. The mediator (for instance, a battery in case of the energy consumption) can obfuscate the actual consumption of the house owner by withdrawing the energy from the grid in a manner that would be probabilistically-independent of the actual consumption [13]. In contrast, with the privacy-preserving protocols the user will compute the utility bill from the actual utility usage and transmit this bill to the provider alongside a zero-knowledge proof that ensures the calculation to be correct and leaks no utility consumption calculation [15].

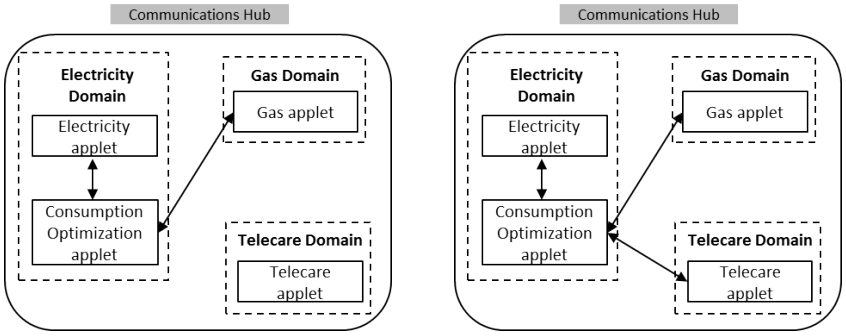
Similarly to the privacy-preserving protocols idea, in the smart meter system architecture proposed in the Hydra project the utility consumption is computed and billed directly on the smart card; afterwards the billing is displayed to the customer via the standard web interface. Therefore the private consumption data does not leave the metering system, and the utility provider can only see the total amount of the consumed utility. The Hydra architecture invites multiple utility providers to share the platform; this is possible with the multi-application smart card solution. The secure communication channels are established between a utility meter, the corresponding application on the card and the provider. These channels are available due the GlobalPlatform middleware present on the card.

GlobalPlatform is a set of card and device specifications produced and maintained by the GlobalPlatform consortium[9]. The specifications identify interoperability and security requirements for development, deployment and management of smart cards. However, the application interactions are not controlled by GlobalPlatform; they are managed by the Java Card run-time environment (JCRE), which we will further overview.

Scenario 2: Existing Application Updates Its Policy We refine the scenario in Fig. 2 and introduce an additional Consumption Optimization application from the electricity provider. This application is connected to the household appliances in order to manage the electricity consumption in a cost-efficient manner. For instance, this application will turn on the washing machine only in the evenings when the electricity cost is the lowest.

The Consumption Optimization applet receives data on the current electricity consumption from the Electricity applet; therefore these applets interact directly on the hub. We consider that the gas provider would also like to provide the optimization services for the customer: he would like to manage the gas consumption (for instance, for heating) in a cost-efficient manner. For this purpose the Consumption Optimization applet can be used, because it is already connected to all the appliances. The electricity and the gas providers sign an agreement, and the interaction between the Gas applet and the Consumption Optimization applet is established on the hub. We emphasize that our focus in

[9] `www.globalplatform.org`



(a) Allowed Applet Interactions Before Policy Update

(b) Allowed Applet Interactions After Policy Update

Fig. 3. Application Interactions on the Smart Meter System Communications Hub

this illustrative scenario is authorization of applications for interactions; we do not consider the privacy concerns that potentially arise from the interaction of the Gas and Consumption Optimization applets. The privacy problem must be handled by the providers separately.

Initially the telecare provider and the electricity provider do not have an agreement; therefore, their applets cannot interact. However, later the electricity provider might be interested in lending the Consumption Optimization applet services also to the Telecare provider, and he will allow the interaction. Fig. 3 summarizes the authorized interactions before and after this policy update of the Consumption Optimization applet.

Notice, that the standard smart card application update procedure is full deletion of the old version and loading of the new one; partial code updates are not supported. In this setting the cost of adding a single authorization is quite significant: for some cards each new code version needs to be agreed with the platform controlling authority or the application provider himself does not have a code loading privilege and has to request the entity with this privilege to perform the code loading. Therefore the provider would like to execute the policy update independently, using the standard protocols for communication with the platform. In the S×C approach we enable the providers with this option.

3 The Security-by-Contract Components and Workflows

The illustrative use cases and scenarios presented in §§2.1-2.2 identify the need of the smart card system to provide access control facilities for applet interaction. We propose the S×C approach for multi-application smart cards to enable the applet authorization and validate the applet code with respect to the interaction policies at load time. This approach ensures that the platform is secure with respect to the applet interactions across the platform changes: installation of a new applet, removal or update of an old one. The main components of the S×C

framework are the ClaimChecker, the PolicyChecker and the PolicyStore, their responsibilities are specified for each type of the platform change.

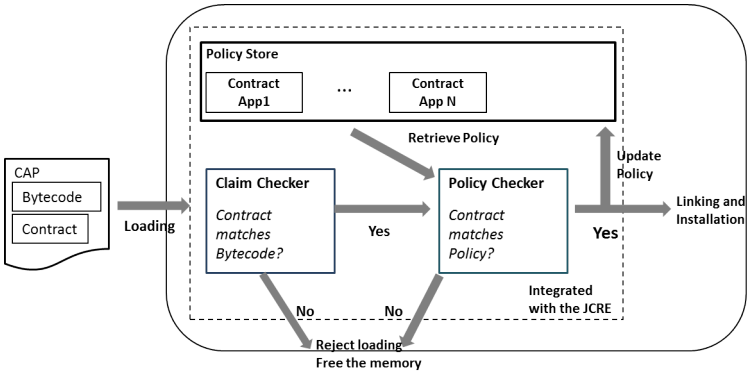


Fig. 4. The S×C workflow for load time validation.

The schema for load time validation is presented in Fig. 4. The application provider delivers on the platform the applet code together with the *contract*. The contract specifies the *claimed interactions* (the services that are provided in this applet and the services of other applets that the applet may invoke at run-time) and the *policy of this application* (which applets are authorized to interact with its services and the necessary services from other applets). Notice that the service deemed necessary for some application have to be a subset of the services called by this application. The first step of load time validation on the card is the check that the contract is compliant with the code; this check is executed by the ClaimChecker component. The second step is matching the contract and the *platform policy*, which is composed by the contracts of all currently installed applications; this check is performed by the PolicyChecker component that retrieves the platform policy from the PolicyStore. If both steps are successful, the S×C admits this applet on the card, and the contract of this applet is added to the platform policy. If any of the checks failed, the loading process will be aborted and this applet will not be admitted to the platform.

For the case of the applet deletion from the platform, the S×C framework has to check that the platform will be secure with respect to application interactions once the requested deletion is performed. We present the workflow for removal in Fig. 5. In the removal workflow only the PolicyChecker and the PolicyStore components are invoked; the ClaimChecker is not required. For the application policy update scenario (presented in Fig. 6) the PolicyChecker has to ensure that after the update the platform will be in the secure state. In this case the ClaimChecker is not invoked, because the code-contract compliance was already validated at the installation step. The S×C workflow for update is designed only

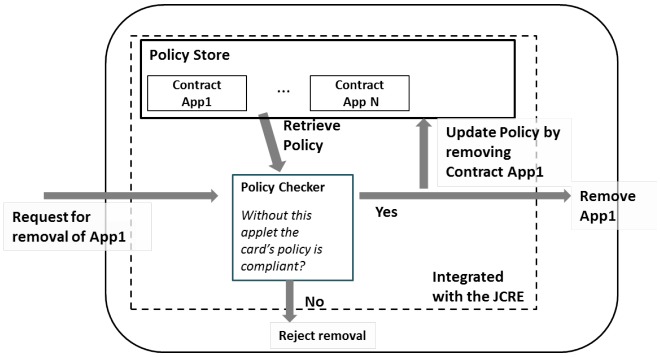


Fig. 5. The S×C workflow for load time validation.

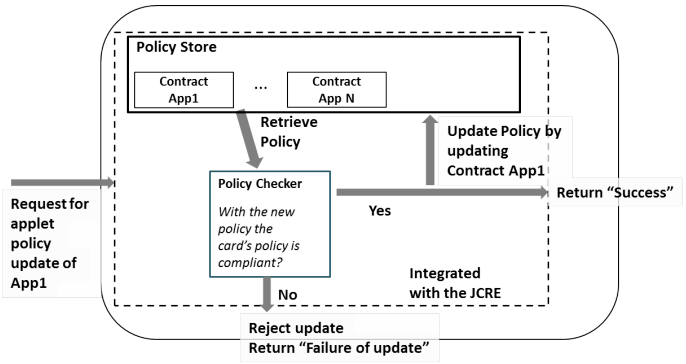


Fig. 6. The S×C workflow for applet policy update validation.

for the policy updates; it cannot handle the code updates, which have to be executed throughout the standard smart card code loading process.

4 A Primer on the Java Card Technology

Before describing the concrete contract and policy implementation and the S×C framework design details, we provide a necessary background on Java Card.

The Java Card platform architecture consists of several layers that include device hardware, a proprietary embedded operating system (Native OS), the JCRE and the installed applications [18]. The JCRE comprises multiple components, some of them belong to the Java Card interface and other belong to the Native interface. The Java Card interface components are the Installer (the entity responsible for the loading and installation processes, it is exposed for communications from the terminal, which is an external device that powers the card up and communicates with it) and the Java Card API; these parts are

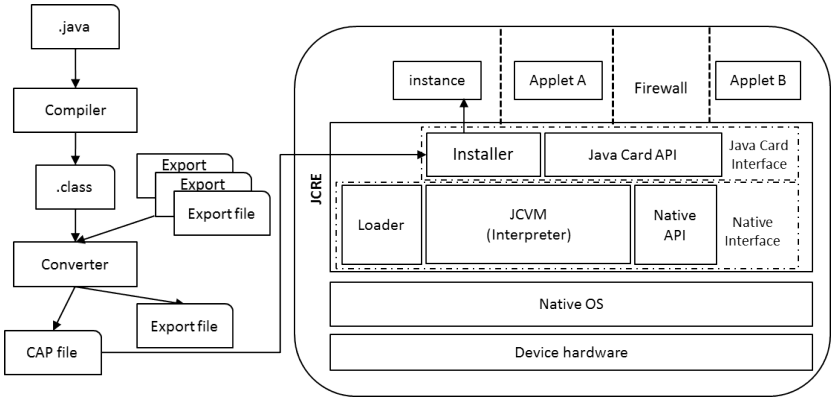


Fig. 7. The Java Card architecture and the applet deployment process.

written in Java Card (subset of Java) and can allocate the EEPROM memory (the persistent modifiable memory).

The Native interface components are the Java Card Virtual Machine (JCVM), the Loader (entity responsible for processing the delivered CAP files) and the Native API. These components are typically written in the native code (C); they are printed in ROM (non-modifiable persistent memory) and are not allowed to allocate the EEPROM memory.

An application package is written in Java, compiled into a set of class files and converted into a CAP (Converted APplet) file, which is delivered on the card. CAP files are optimized by the Converter in order to occupy less space. For instance, a CAP file contains a single Constant Pool component, a single Method component with the full bytecode set of all methods defined in this package, etc. A quite similar approach for optimization of packages loaded on the device is adopted on the Android platform[10]. Each package can contain multiple applications, but interactions inside a package cannot be regulated. Also, each package is loaded in a single pass, and it is not possible to add a malicious applet to the package of an honest applet. Therefore in the sequel we consider that each package contains exactly one application and use words package and applet interchangeably.

The CAP file is transmitted onto a smart card, where it is processed, linked and then an application instance can be created. One of the main technical obstacles for the verifier running on Java Card is unavailability of the application code for reverification purposes after linking. Thus the application policy cannot be stored within the application code itself, as the verifier will not have access to it later.

The Java Card platform architecture and the applet deployment process are summarized in Fig. 7.

[10] `www.android.com`

4.1 Application Interactions

Applications on Java Card are separated by a firewall, and the interactions between applets from different packages are always mediated by the JCRE. If two applets belong to different packages, they belong to different *contexts*. The Java Card firewall confines applet's actions to its designated context. Thus an applet can freely reach only objects belonging to its own context. The only objects accessible through the firewall are methods of *shareable interfaces*, also called *services*. A shareable interface is an interface that extends `javacard.framework.Shareable`.

An applet A implementing some services is called a *server*. An applet B that tries to call any of these services is called a *client*. A typical scenario of service usage starts with a client's request to the JCRE for a reference to A's object (that is implementing the necessary shareable interface). The firewall passes this request to application A, which decides if the reference can be granted or not based on its access control rules. The current method for services access control implementation on Java Card is the list of trusted clients embedded into the applet code. The caller can be identified through the Java Card `getPreviousContext` API. If the decision is positive, the reference is passed through the firewall and the client can now invoke any method declared in the shareable interface which is implemented by the referenced object. During invocation of a service a *context switch* will occur, thus allowing invocation of a method of the application A from a method of the application B. A call to any other method, not belonging to the shareable interface, will be stopped by the Java Card firewall; while the calls from A to its own methods or the JCRE entry points are allowed by the firewall rules [18]. Notice that with the S×C framework on board the access control policies embedded into the applet code will become obsolete.

In order to realize the interaction scenario the client has necessarily to import the shareable interface of the server and to obtain the *Export file* of the server, that lists shared interfaces and services and contains their *tokens*. Tokens are used by the JCRE for linking on the card similarly as Unicode strings are used for linking in standard Java class files. A service s can be uniquely identified as a tuple $\langle A, I, t\rangle$, where A is a unique application identifier (AID) of the package that provides the service s, that is assigned according to the standard ISO/IEC 7816-5, I is a token for a shareable interface where the service is defined and t is a token for the method in the interface I. Further, in case the origin of the service is clear, we will omit the AID and will refer to a service as a tuple $\langle I, t\rangle$.

The server's Export file is necessary for conversion of the client's package into a CAP file. In a CAP file all methods are referred to by their tokens, and during conversion from class files into a CAP file the client needs to know correct tokens for services it invokes from other applications. Shareable interfaces and Export files do not contain any implementation, therefore it is safe to distribute them.

5 Design and Implementation Details

Let us present the details of the S×C framework implementation.

5.1 Contracts

The contract of an application is defined as follows. AppClaim of an application specifies provided (the Provides set) and invoked (the Calls set) services. Let A be an application and $A.s$ be its service. We say that the service $A.s$ is *provided* if applet A is loaded and it has service s. Service $B.m$ is *invoked* by applet A if A may try to call $B.m$ during its execution. The AppClaim will be verified for compliance with the bytecode (the CAP file) by the ClaimChecker.

The application policy AppPolicy contains *authorizations for services access* (the sec.rules set) and *functionally necessary services* (the func.rules set). We say a service is necessary if a client will not be functional without this service on board. The AppPolicy lists applet's requirements for the smart card platform and other applications loaded on it.

Thus the application contract is: $\mathsf{Contract} = \langle \mathsf{AppClaim}, \mathsf{AppPolicy} \rangle$, where $\mathsf{AppClaim} = \langle \mathsf{Provides}, \mathsf{Calls} \rangle$ and $\mathsf{AppPolicy} = \langle \mathsf{sec.rules}, \mathsf{func.rules} \rangle$.

Writing and Delivering Contracts A provided service is identified as a tuple $\langle A, I, s \rangle$, where A is the AID of package A (as the AID of A is explicit in the package itself, it can be omitted from the provided service identification tuple), and I and s are the tokens of the shareable interface and the method that define the service. Correspondingly, a called service can be identified as a tuple $\langle B, I, s \rangle$, where B is the AID of the package containing the called service.

An authorization rule is a tuple $\langle B, I, s \rangle$, where B is the AID of package B that is authorized to access the provided service with interface token I and method token s. Notice that A is not specified in its authorization rules because the rules are delivered within the A's package and the service provider is implicitly identified. Later, when the authorization rules will be added to the platform security policy, the service provider will be explicitly specified.

$\mathsf{func.rules_A}$ is a set of functionally necessary services for A, we consider that without these services provided, an application A cannot be functional. Thus we can ensure availability of necessary services. A functionally necessary service can be identified in the same way as a called service. Moreover, we insist that $\mathsf{func.rules_A} \subseteq \mathsf{Calls_A}$, as we cannot allow to declare arbitrary services as necessary, but only the ones that are at least potentially invoked in the code.

The provided service tokens can be found in the Export file, where the fully-qualified interface names and method names are present. The called services can be determined from the Export files of the desired server applets, which are consumed for conversion. More details on the token extraction from the Export files can be found in [6, 7]. However, the contract-code matching will be performed on card with the CAP file, as the Export files are not delivered on board.

An important problem is contract delivery on the platform. The Java Card specification, besides the standard CAP file components, allows Custom components to be present in CAP files [18]. We have organized the contract delivery within a specific Contract Custom component. In this way the contract can be securely attached to the bytecode and sealed by the provider's signature. The

standard Java Card tools do not include means to provide Custom components, so for the proof-of-concept implementation we have designed a simple CAP Modifier tool with a GUI that provides means to write contracts and add them as a Custom component to standard CAP files. More details are available in [7].

5.2 The S×C Components

We now specify the design of each component of the S×C framework.

The ClaimChecker The ClaimChecker is responsible for the contract-code matching step. It has to retrieve the contract from the Custom component and verify that the AppClaim is faithful (the application policy part is not matched with the code). For every service from the Provides set the ClaimChecker will find its declaration in the Export component of the CAP file; and it will ensure no undeclared services are present in the Export component. For every called service from the Calls set the ClaimChecker will identify the point in the code when this service is invoked.

Specifically, the ClaimChecker will parse the CAP file and find all the service invocation instructions (the `invokeinterface` opcode). From the operands of this instruction we can identify the invoked method token and the pointer to the Constant Pool component, from which we can resolve the needed invoked interface token and the AID of the called applet. Concrete details of this procedure are available in [7]. To implement the CAP file processing the ClaimChecker has to be integrated with the platform Loader component, as only this component has direct access to the loaded code.

The PolicyChecker The PolicyChecker component is responsible for ensuring compliance of the platform security policy and the contract for the loading protocol. Namely, it will check that 1) for all called services from Calls, their providers have authorized the interaction in the contracts; 2) for any provided service from Provides if there is some applet on the platform that can try to invoke this service, there is a corresponding authorization rule for these service and applet in sec.rules; 3) all services in func.rules are present on the platform (provided by some applets). We have integrated the PolicyChecker with the ClaimChecker, to ease the contract delivery. The PolicyChecker is a part of the SxCInstaller component that is the main verification interface with the Loader.

For the applet removal scenario, the PolicyChecker has to retrieve the platform policy, identify the contract of the applet to be removed and check if this applet provides any service that is listed as functionally necessary by some other applet. This is the only incompliance problem, because removal of an applet cannot introduce unauthorized service invocation. If the applet is not needed by the others, the PolicyChecker will remove its contract from the platform policy and send the new policy to the PolicyStore.

If any compliance check performed by the ClaimChecker or the PolicyChecker has failed, the components signal to the Loader, which will stop the executed

change on the platform (due to the transaction mechanism on Java Card the platform will return to the previous secure state).

The PolicyStore The PolicyStore is used to maintain the security policy across the card sessions. As the policy of the platform is dynamic in nature and cannot be static throughout the card lifecycle, it has to be stored in the EEPROM. Therefore, as we have specified in §4, the PolicyStore cannot be implemented as a part of the Native interface of the JCRE, but instead it should be the part of the Java Card interface. We have integrated the PolicyStore with the Installer component. However, the ClaimChecker needs to be integrated with the native Loader, thus the S×C framework has to be divided across the Native and the Java Card interfaces. In the same time, we need to enable the communication between these parts, in order to retrieve the current card policy and update it after changes. This communication is realized using a new dedicated Native API.

The security policy data structures were designed to be memory-saving. For example, the applet AID can occupy up to 16 bytes, therefore each called service can occupy up to 18 bytes (the interface and method tokens each occupy 1 byte). We decided to store the policy in the bit array format that allows to speed up the policy matching operations. The platform policy data structure currently supports up to 10 loaded applets, each containing up to 8 provided services; but these applets are not pre-defined and any AID can be listed in the policy.

The final implementation of the S×C framework delegates the validation for the scenario of application policy update to the PolicyStore. The reason for this decision is the fact, that it is integrated with the platform Installer, which is already exposed to the communications with the outside world. In this way we do not need to identify new protocols for the policy update, what would be necessary if we needed to invoke the PolicyChecker, and hence - the Loader. The application policy update is atomic: each time only one authorization rule can be added or removed, or one functionally necessary service can be added or removed. For the authorization service addition and functionally necessary service removal the update can be executed directly, as these changes cannot introduce inconsistency. For the removal of an authorization the PolicyStore will ensure the de-authorized client does not call this service. For the addition of a functionally necessary service the PolicyStore will check this service is actually called and is provided on the platform. If the check is successful, the update is applied, otherwise the policy is not modified.

5.3 Integration with the Java Card Platform

Fig. 8 presents the Java Card architecture extended with the S×C framework and the new steps in the applet development and deployment process. The S×C framework is fully backward-compatible with the existing Java Card platforms. The platforms that do not know about the framework will be able to process applets with contracts, because unknown Custom components are just ignored by default. The applets without the contract can be still deployed on the platform:

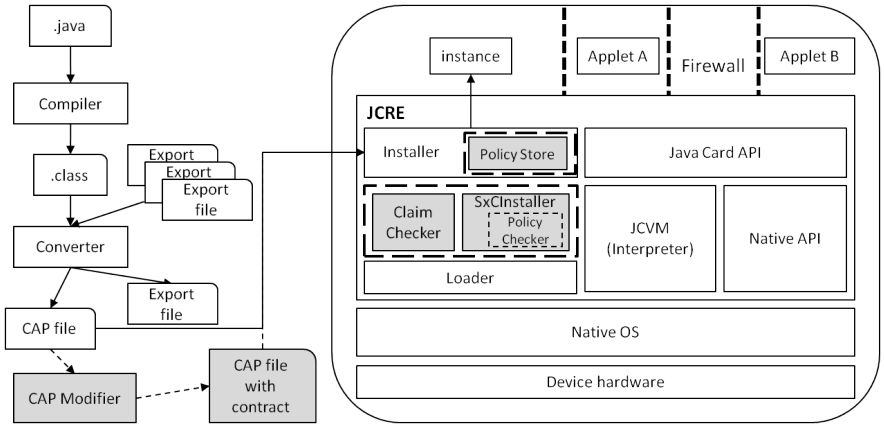


Fig. 8. The Java Card architecture and the applet deployment process in presence of the S×C framework. The grey components are introduced by the S×C approach, the dashed lines denote new steps in the development and deployment process.

they will be treated as providing and calling no services. We did not modify the standard loading protocol or the JCVM. The interested reader can find more information on the S×C design challenges and the implementation details in [7].

6 Application to the Use Case Scenarios

Let us now present concrete contracts we devised for the motivating scenarios introduced in §§2.1-2.2.

6.1 The NFC-Enabled Phone and the New Applet Installation

We consider the payment functionality of the payWave application to be implemented in the shareable `PaymentInterface` and the service `payment`. Let the Touch&Travel applet has the AID `0xAA01CA71FF10` and the payWave applet has the AID `0x4834D26C6C6F4170`[11].

We can notice in Tab. 1, which presents the contracts of the scenario applets, that one of the presented contracts of Navigo (Non-Compliant) is not compliant with the contract of the payWave application (specifically, Navigo calls the payment service, but is not authorized to do so). Therefore in our scenario, when the device holder requests the loading of Navigo with this contract, the loading will be rejected by the S×C framework. If the device holder installs the Navigo version without the non-authorized call to payWave (the Compliant option), then it will be installed without any problems.

[11] The chosen AIDs are fictional, but they are compliant with the ISO/IEC 7816-5 standard to give an idea how an actual contract can look like.

Table 1. Contracts of the payWave, Touch&Travel and Navigo applets.

Contract structure	Fully-qualified names	Token identifiers
payWave		
Provides	`PaymentInterface.payment()`	⟨0, 0⟩
Calls		
sec.rules	Touch&Travel is authorized to call `PaymentInterface.payment()`	⟨ `0xAA01CA71FF10`, 0, 0⟩
func.rules		
Touch&Travel		
Provides		
Calls	payWave.`PaymentInterface.payment()`	⟨`0x4834D26C6C6F4170`, 0, 0⟩
sec.rules		
func.rules		
Navigo – Non Compliant		
Provides		
Calls	payWave.`PaymentInterface.payment()`	⟨`0x4834D26C6C6F4170`, 0, 0⟩
sec.rules		
func.rules		
Navigo – Compliant		
Provides		
Calls		
sec.rules		
func.rules		

6.2 The Telecommunications Hub and the Application Policy Update

For the application policy update scenario we present the contracts of the Consumption Optimization applet before and after the update. We consider the consumption optimzation functionality to be implemented in the shareable `OptimizationInterface` and the service `optimization`. Let the Electricity applet has the AID `0xEE06D7713386`, the Consumption Optimization applet has the AID `0xEE06D7713391`,the Gas applet has the AID `0xGG43F167B2890D6C` and the Telecare applet has the AID `0x4D357F82B1119AEE`.

Tab. 2 presents contracts the Electricity, Gas, Telecare and Consumption Optimization applets. Notice that the application policy update for addition of the authorization for Telecare is possible and will be executed by the S×C framework. The Telecare applet can later be updated and in the new version the call to the `optimization` service will appear.

7 Related Work

Fontaine et al. [5] design a mechanism for implementing transitive control flow policies on Java Card. These policies are stronger than the access control policies provided by our framework, because the S×C approach targets only direct service invocations. However, the S×C approach has the advantage of the openness of the policy to any applet AID. The main limitation of [5] is the focus on ad-hoc

Table 2. Contracts of the Electricity, Consumption Optimization, Gas and Telecare applets.

Contract structure	**Fully-qualified names**	**Token identifiers**
Consumption Optimization – Before		
Provides	`OptimizationInterface.optmization()`	⟨0, 0⟩
Calls		
sec.rules	Electricity applet is authorized to call `OptimizationInterface.optmization()`	⟨`0xEE06D7713386`, 0, 0⟩
	Gas applet is authorized to call `OptimizationInterface.optmization()`	⟨`0xGG43F167B2890D6C`, 0, 0⟩
func.rules		
Electricity		
Provides		
Calls	ConsumptOptim.`OptimizationInterface.optimization()`	⟨`0xEE06D7713391`, 0, 0⟩
sec.rules		
func.rules		
Gas		
Provides		
Calls	ConsumptOptim.`OptimizationInterface.optimization()`	⟨`0xEE06D7713391`, 0, 0⟩
sec.rules		
func.rules		
Telecare		
Provides		
Calls		
sec.rules		
func.rules		
Consumption Optimization – After		
Provides	`OptimizationInterface.optmization()`	⟨0, 0⟩
Calls		
sec.rules	Electricity applet is authorized to call `OptimizationInterface.optmization()`	⟨`0xEE06D7713386`, 0, 0⟩
	Gas applet is authorized to call `OptimizationInterface.optmization()`	⟨`0xGG43F167B2890D6C`, 0, 0⟩
	Telecare applet is authorized to call `OptimizationInterface.optmization()`	⟨`0x4D357F82B1119AEE`, 0, 0⟩
func.rules		

security domains, which are very coarse grained administrative security roles (usually a handful), typically used to delegate privileges on GlobalPlatform. As a consequence we can provide a much finer access control list closer to actual practice.

An information flow verification system for small Java-based devices is proposed by Ghindici et al. [9]. The system relies on off-device and on-device steps. First, an applet *certificate* is created off device (contains information flows within the applet). Then 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. However, we believe the on device information flow verification for Java Card is not yet practical due to the resource and architecture limitations. The proposed system cannot be implemented for Java Card version 2.2 because the latter does not allow custom class loaders,

and even implementation for Java Card version 3.0 may not be effective due to significant amount of memory required to store the information flow policies.

There were investigations [2, 3, 10, 11, 16] of static scenarios, when all applets are known and the composition is analyzed off-device. For example, Avvenuti et al. [2] have developed the JCSI tool which verifies whether a Java Card applet set respects pre-defined information flow policies. This tool runs off-card, so it assumes an existence of a controlling authority, such as a telecom operator, that can check applets before loading.

The investigation of the Security-by-Contract techniques for Java Card is carried out in [4, 8, 6, 7] targeting dynamic scenarios when third-party applets can be loaded on the platform.

Dragoni et al. [4] and Gadyatskaya et al. [8] propose an implementation of the PolicyChecker component as an applet. While very appealing due to avoiding the JCRE modification, it has not solved in any way the actual issue of integration with a real platform. This solution could only work if the authors of [4, 8] had access to the full Java-based JCRE implementation, because only in this way the Loader can be implemented as a part of the Java Card interface. The Java Card specifications do not prohibit this, but in practice full Java-based implementations do not exist.

8 Conclusions and the Future Work

The S×C framework enables load time bytecode validation for multi-application Java cards. Now each application provider can independently deploy her applications and update the application policies. The proposed approach fits on a real smart card, it enables the backward compatibility and is not very invasive, as the changes to the platform are kept at minimum.

The main benefit of the proposed solution is the validation of the code on card. In this way each card is independent in the decision it takes; we can envisage that (U)SIM cards from the same telecom operator can contain different application sets, depending on the needs of the phone holder. The telecom operator now does not need to verify security for each possible set of applets; therefore the costs of managing the device are lower. We can envisage that the S×C approach will be quite efficient for less expensive applets (like the already mentioned messages from Santa) that do not provide and do not call any services. This fact can be easily ensured on the card itself, and these applets do not need to pass the costly certification process.

Our framework performs the load time on-card checks of Java Card bytecode. The restrictions of the Java Card platform (the dedicated service invocation instruction and the static invoked class binding in the CAP file) allow our framework to efficiently analyze the sets of invoked and provided services in the code and match them with the contract. We can notice that our bytecode analysis techniques will have to be improved, for example, following [19], before application to full Java, because the inference of method invocation targets will be more complicated. However, the idea of performing load time on-device

checks on the bytecode is promising for computationally-restricted devices, e.g. the Android phones. The users expect certain delay when an application is being installed, but they will not tolerate any runtime lags. Therefore, we think that the Security-by-Contract idea is very promising for Android and other constrained devices.

We have chosen the conservative approach for verification: if the change can lead to an insecure state it is rejected. However, this might not be acceptable in the business community. For instance, we can envisage that application providers would like to be able to revoke the access to their sensitive service at any time. The S×C framework currently does not provide this option: the access to a service for a specific client can be revoked only if this client does not actually invoke this service. To be more practical, the card should be able to perform some conflict resolution. For instance, one can choose an approach in which the application provider is always able to revoke access to her service, and the client applet will become locked until the new version without service invocation is deployed. Another possibility is to explore more the centralized policy management facilities offered by the next generations of the Java Card platform. We expect a lot of interesting research challenges in this direction.

Acknowledgements

This work was partially supported by the EU under grants EU-FP7-FET-IP-SecureChange and FP7-IST-NoE-NESSOS. We thank Eduardo Lostal for implementing the first version of the S×C prototype and Boutheina Chetali and Quang-Huy Nguyen for evaluation of the prototype and valuable information on the platform internals. We also thank Davide Sangiorgi and Elena Giachino for the invitation to give the S×C tutorial at the HATS-2012 Summer School.

References

1. J. Aljuraidan, E. Fragkaki, L. Bauer, L. Jia, K. Fukushima, S. Kiyomoto, and Y. Miyake. Run-time enforcement of information-flow properties on Android. Technical Report CMU-CyLab-12-015, Carnegie Mellon University, `http://repository.cmu.edu/cgi/viewcontent.cgi?article=1110&context=cylab`, accessed on the web in Dec. 2012.
2. M. Avvenuti, C. Bernardeschi, N. De Francesco, and P. Masci. JCSI: A tool for checking secure information flow in Java Card applications. *J. of Systems and Software*, 85(11):2479–2493, 2012.
3. P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J-L. Lanet. Checking secure interactions of smart card applets: Extended version. *J. of Comp. Sec.*, 10(4):369–398, 2002.
4. N. Dragoni, E. Lostal, O. Gadyatskaya, F. Massacci, and F. Paci. A load time Policy Checker for open multi-application smart cards. pages 153–156. IEEE Press.
5. A. Fontaine, S. Hym, and I. Simplot-Ryl. On-device control flow verification for Java programs. In *Proc. of ESSOS'2011*, volume 6542 of *LNCS*, pages 43–57. Springer-Verlag, 2011.

6. O. Gadyatskaya, E. Lostal, and F. Massacci. Load time security verification. In *Proc. of ICISS'2011*, volume 7093 of *LNCS*, pages 250–264. Springer-Verlag, 2011.
7. O. Gadyatskaya, F. Massacci, Q.-H. Nguyen, and B. Chetali. Load time code validation for mobile phone Java Cards. Technical Report DISI-12-025, University of Trento, 2012.
8. O. Gadyatskaya, F. Massacci, F. Paci, and S. Stankevich. Java Card architecture for autonomous yet secure evolution of smart cards applications. In *Proc. of NordSec'2010*, volume 7127 of *LNCS*, pages 187–192. SV.
9. D. Ghindici and I. Simplot-Ryl. On practical information flow policies for Java-enabled multiapplication smart cards. In *Proc. of CARDIS'2008*, volume 5189 of *LNCS*, pages 32–47. Springer-Verlag, 2008.
10. P. Girard. Which security policy for multiapplication smart cards? In *Proc. of USENIX Workshop on Smartcard Technology*. USENIX Association, 1999.
11. M. Huisman, D. Gurov, C. Sprenger, and G. Chugunov. Checking absence of illicit applet interactions: a case study. In *Proc. of FASE'04*, volume 2984 of *LNCS*, pages 84–98. Springer-Verlag, 2004.
12. Project Hydra. Project Hydra - Smart care for smart meters. Final report, October 2012. `http://projecthydra.info/wp-content/uploads/2012/10/Hydra_Final_Report.pdf`. Accessed on the web in Dec. 2012.
13. J. Koo, X. Lin, and S. Bagchi. Privatus: Wallet-friendly privacy protection for smart meters. In *Proc. of ESORICS 2012*, volume 7459 of *LNCS*, pages 343–360. SV, 2012.
14. J. Langer and 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 December 2012.
15. A. Rial and G. Danezis. Privacy-preserving smart metering. In *Proc. of the ACM WPES '11*, pages 49–60. ACM, 2011.
16. G. Schellhorn, W. Reif, A. Schairer, P. Karger, V. Austel, and D. Toll. Verification of a formal security model for multiapplicative smart cards. In *Proc. of ESORICS'00*, volume 1895 of *LNCS*. Springer-Verlag, 2000.
17. B. Sullivan. What will talking power meters say about you? `http://redtape.nbcnews.com/_news/2009/10/09/6345711-what-will-talking-power-meters-say-about-you`. Accessed on the web in Dec. 2012.
18. Sun. Runtime environment and virtual machine specifications. Java Card^{TM} platform, v.2.2.2. Specification, 2006.
19. V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *In Proc. of OOPSLA-2000*, pages 264–280. ACM Press, 2000.