



# D6.3: COMPOSITIONAL TECHNIQUE TO VERIFY ADAPTIVE SECURITY AT LOADING TIME ON-DEVICE

Arnaud FONTAINE, Samuel HYM, Isabelle SIMPLOT-RYL (INR); Olga GADYATSKAYA, Fabio MASSACCI, Federica PACI (UNITN); Jan JURGENS, Martin OCHOA (TUD)

## Document Information

| | |
|---|---|
| **Document Number** | D6.3 |
| **Document Title** | Compositional technique to verify adaptive security at loading time on-device |
| **Version** | 2.9 |
| **Status** | Final |
| **Work Package** | WP 6 |
| **Deliverable Type** | Report |
| **Contractual Date of Delivery** | M18 |
| **Actual Date of Delivery** | M18 |
| **Responsible Unit** | INR |
| **Contributors** | INR, UNITN, TUD |
| **Keyword List** | Verification, On-device |
| **Dissemination** | PU |

## Document change record

| Version | Date | Status | Author (Unit) | Description |
|---|---|---|---|---|
| 0.1 | 2010/05/27 | Working | S. Hym, I. Simplot-Ryl (INR) | Plan and document structure, chapters Overview & Notations |
| 0.2 | 2010/06/13 | Working | I. Simplot-Ryl (INR) | Addition of chapter Global policy |
| 0.3 | 2010/06/16 | Working | I. Simplot-Ryl (INR) | Update to chapter Notations |
| 0.4 | 2010/06/16 | Working | O.Gadyatskaya, F. Massacci (UNITN) | Update to the Overview chapter, addition of chapter Rel works, corrections to the chapter Notations |
| 0.5 | 2010/06/17 | Working | O.Gadyatskaya (UNITN) | Update to the Overview chapter: section 2.5.1, some typos fixed |
| 0.6 | 2010/06/18 | Working | O.Gadyatskaya (UNITN) | Glossary and abbreviations updated |
| 0.7 | 2010/06/18 | Working | I. Simplot-Ryl (INR) | Update to chapter Global policy |
| 0.8 | 2010/06/18 | Working | O.Gadyatskaya, F. Massacci (UNITN) | Chapter on direct control flow added |
| 0.9 | 2010/06/24 | Working | A. Fontaine, S. Hym, I. Simplot-Ryl (INR) | Chapter on transitive control flow added |
| 1.0 | 2010/06/24 | Working | A. Fontaine, S. Hym, I. Simplot-Ryl (INR) | Update of chapter Overview |
| 1.1 | 2010/06/27 | Working | I. Simplot-Ryl (INR) | Cleaning of bibliography |
| 1.2 | 2010/06/28 | Working | I. Simplot-Ryl (INR) | Chapter on non-interference added |
| 1.3 | 2010/06/28 | Working | A. Fontaine, I. Simplot-Ryl (INR) | Update of related works |
| 1.4 | 2010/07/02 | Working | O.Gadyatskaya (UNITN) | Update of the Chapter Overview (RelWorks merged, new pictures, cleaning) |
| 1.5 | 2010/07/02 | Working | O.Gadyatskaya (UNITN) | Executive summary and Conclusion drafts added |
| 1.6 | 2010/07/03 | Working | I. Simplot-Ryl (INR) | Update of non-interference |
| 1.7 | 2010/07/04 | Working | S. Hym,I. Simplot-Ryl (INR) | Update of chapter Global policy |
| 1.8 | 2010/07/04 | Working | A.. Fontaine,I. Simplot-Ryl (INR) | Update of chapter Overview |

| | | | | |
|---|---|---|---|---|
| 1.9 | 2010/07/05 | Working | O.Gadyatskaya, F.Paci (UNITN) | Chapter Overview fixed, final draft of Chapter Direct Control Flow submitted |
| 2.0 | 2010/07/05 | Working | I. Simplot-Ryl (INR) | Minor updates in Summary, Chapter non-interference |
| 2.1 | 2010/07/06 | Working | O.Gadyatskaya (UNITN) | General check, typos fixed |
| 2.2 | 2010/07/09 | Working | F. Piessens (KUL) | Review of the document |
| 2.3 | 2010/07/14 | Working | K. Bekoutou, E. Chiarani (UNITN) | First quality check completed, other minor changes requested |
| 2.4 | 2010/07/14 | Working | A. Fontaine, I. Simplot-Ryl (INR) | Style changes, corrections after quality check, section 1.6 added |
| 2.5 | 2010/07/15 | Working | O.Gadyatskaya (UNITN), J.Jurgens, M.Ochoa (TUD) | Link with WP4, threat model updated |
| 2.6 | 2010/07/16 | Working | A. Fontaine, I. Simplot-Ryl (INR) | Check of 1.6, 3.6, corrections of bibliography, related works, style |
| 2.7 | 2010/07/19 | Working | O.Gadyatskaya (UNITN) | Update to Introduction |
| 2.8 | 2010/07/20 | Working | K. Bekoutou, E. Chiarani (UNITN) | Final quality check. Minor changes requested |
| 2.9 | 2010/07/23 | Final | I. Simplot-Ryl (INR) | Corections according to final quality check |

## Executive summary

This document describes compositional techniques to verify evolving security at loading time on small embedded devices (multi-application smart cards). Such small devices have restricted memory and usual run-time monitoring techniques cannot be applied on them. General overview of the targeted systems and possible verification workflows are provided.

Chosen security properties belong to information protection. In order to preserve information exchange security in a dynamic environment, where the applications from different stakeholders can evolve and talk to each other, the device should be able to verify the updates autonomously and in a very lightweight fashion. The proposed techniques cover such parts of the information protection requirement as control flow and special type of non-interference.

Control flow verification provides assurance, that there is no illegal invocation of application services. Chapters 3, 4 and 5 are dedicated correspondingly to direct control flow, transitive control flow and global control flow on-device verification techniques. The chapter 3 also contains the link with the notation provided by WP4.

Non-interference verification is based on the lightweight information flow on-device verification that can be performed with the STAN tool. Chapter 6 describes this approach and enhances it with a compositional verification technique for a security domains hierarchy.

# Index

# List of Figures

# List of Tables

# Abbreviations and Glossary

## Abbreviations

| Abbreviations | References |
|---|---|
| AID | Application IDentifier |
| API | Application Programming Interface |
| CAD | Card Acceptance Device |
| CAP | Converted APplet |
| GP | GlobalPlatform |
| JCRE | Java Card Run-time Environment |
| JVM (JCVM) | Java Virtual Machine (Java Card Virtual Machine) |
| S×C | Security-by-Contract |
| TTP | Trusted Third Party |

**Table 2:** Abbreviations used in the document

## Glossary

| Term | Definition |
|---|---|
| Applet | Java Card application |
| Contract | Formal specification of applet's security relevant behavior |
| GlobalPlatform | Consortium and specification it provides for secure management of multi-application smart card contents |
| Java Card | Hardware-independent technology to run multiple applications on smart cars or other small devices with restricted capabilities |
| OPEN | Card manager on GlobalPlatform |

**Table 3:** Glossary

# 1. Introduction: Overview of the compositional on-device verification

The goal of this deliverable is to define how to verify at loading time on a small device that mobile code respects chosen security properties. This technique should be compositional to allow extensions of the system to be performed without re-verification of already loaded code. In this chapter we provide a general overview of the targeted systems, the chosen security properties and the verification workflows, before going into technical details on the verification methodology.

## 1.1 The system

Open multi-application environments such as PCs and mobile platforms are widespread. The main characteristics of such environments are co-existence of several applications on one single platform and the possibility of this platform to evolve by adding new applications or updating of existing ones in a fully distributed and autonomous way. Without a security policy regulating (unwanted) information exchange the problem is relatively simple.

Modern smart cards could be another example of such a multi-application environment. Everybody has in his wallet half a dozen of plastic cards, mostly smart cards. Would not it be nice to have at least some of them unified into one card? The first research papers on multi-application smart cards appeared more than 10 years ago [23], but the industry has not yet adopted an open multi-application smart card paradigm (in a sense of applications coming and going dynamically from different stakeholders) [40]. Even examples of potentially multi-stakeholders applications are de facto a single one: a loyalty Miles& More Lufthansa credit card [14], could include a Lufthansa application for collecting miles and a Bank application; de facto it is just a credit card and mileage is calculated by the back-end system. Usually the applications on the card are produced by the same provider. Even in the case when the applets are coming from different stakeholders, they were produced accordingly to very precise and restrictive preliminary agreements, that describe the exact structure of the applets and the way they execute and exchange information.

For usual multi-application environments, like PCs, users take decisions what to install and run on their platforms at their own risk. A lot of security solutions, like firewalls, were developed in order to prevent malicious or buggy applications to steal data. In the domain of smart cards the price of data is even higher, since the cards are often used to store sensitive information (personal data, biometrics, bank account details and so on). The restricted memory and computational capabilities of smart cards make them an easier target for frauds, since it is not possible to implement a lot of security mechanisms on smart card platform.

The absence of solutions, when the applets can be provided by different stakeholders and without strict preliminary agreements, can be explained by the gap between the security certification process and the business model behind evolution. The most popular solution for smart card systems now is GlobalPlatofrm (GP) [27] [1] on top of Java Card [32]. Smart card vendors usually develop their own proprietary solutions, implementing some functionality of GP for secure loading of applications, handling keys of security domains, supporting applications life-cycle and unloading of applications if necessary [9, 44]. Currently smart card vendors obtain a compliance certificate w.r.t. the appropriate security standards (e.g. [33]). This certification is essentially an off-line verification of the platform and all the applications on it. If any security-relevant change occurs on the platform, the certificate will no longer be valid. In order to obtain a new certificate the smart card owner has to re-verify from scratch all the platform and the applications. This approach is not compositional and costs a lot. Thus smart card providers might as well develop an open multi-application platform, but, at the end of the day, they prefer to lock the card to forbid addition of new applications, changes in the policy and so on [45].

Nevertheless, the applications' interactions on the card are a very interesting target for smart card users and developers. People would like to be able to install on a single card all transport applets from different cities they visit or loyalty applets of various supermarket chains they shop in. If the applications could exchange information and use services of each other directly on the platform, for example, multiplying royalty points, providing some details of discount rates and so on, the customers would be even more satisfied. But openness of the smart card platform can create certain vulnerabilities, for example, already verified application can be updated in a way that it will break the information exchange security policy of the card. Existing security mechanisms and approaches, which we will describe further in this section, cannot protect the applications' data sufficiently.

In the context of SecureChange, the validation of the platform code is addressed by the Work Package 7. Thus, we consider applications in an open system with the following definitions:

**Application** a coherent code unit, compiled simultaneously by its producer in order to be deployed on a multi-application smart card.

**Open system (smart card platform)** a set of software units which can be dynamically extended with new software units.

The systems can evolve by addition of new applications or elements of applications, thus in the context of security verification, they must fulfill the following requirements:

**Consistency** guarantees the security of already installed applications while installing a new application,

**Incrementality** certifies the security of a newly deployed application.

In order to support evolution, we need a way for the platform to certify that applications arriving on the platform comply with the policy.

If we must combine the security policies of multiple applications from different vendors with application A, saying that application B should not be allowed to call A's favorite service, the task is tricky. It becomes daunting when applications come and go, autonomously, asynchronously and in a distributed fashion. When an application arrives or changes it is necessary to check whether it complies with the security policies of other

[1] GlobalPlatform is a specification for secure management of the card multi-application contents

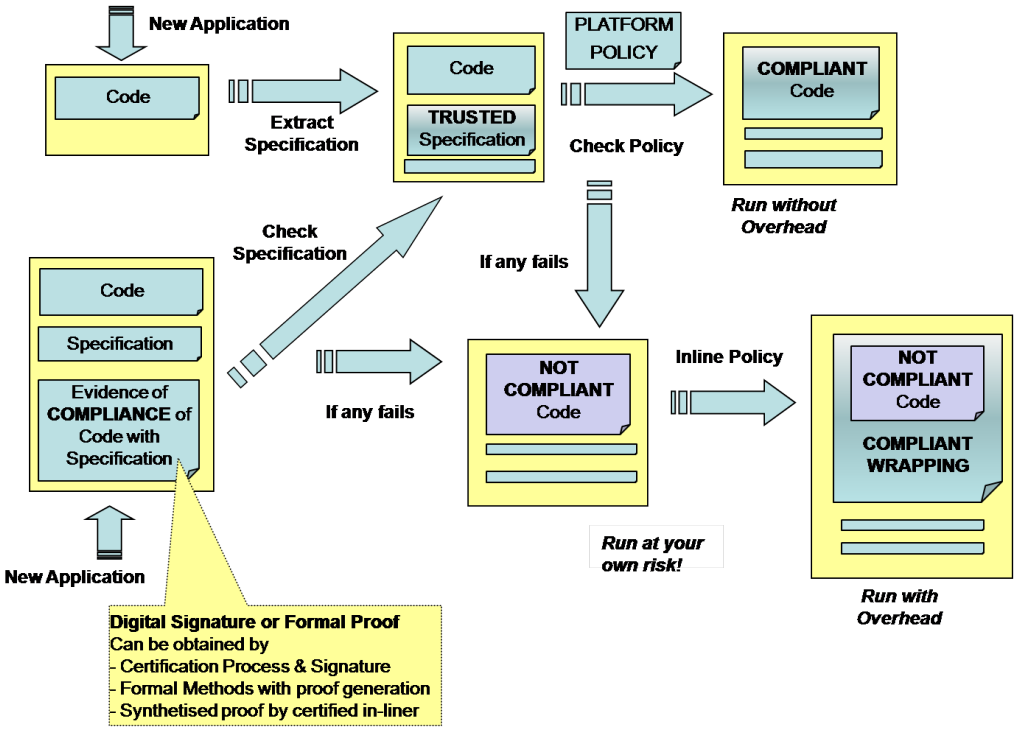


**Figure 1.1:** Verification Workflow for Evolution

applications already on the platform. Systems with enough computation power can use run-time monitors to control the applications [15]. But even for mobile devices the best solution might be to verify or at least certify newcomers at loading time. The idea was investigated by Sekar et al. when the notion of *model-carrying code* was introduced [43], has been successfully demonstrated in the *Security-by-Contract* (S×C) approach for .NET and Java [17, 15], and was later adopted by W. Enck et al. for Android security [19]. Figure 1.1 shows the basic idea. An application arrives with a specification (formal description) of its behavior or this specification is extracted from the code on device. The specification is then checked for compliance with the security policy of the platform.

Why cannot this be done with the approaches for load time evolution just cited?

- In traditional mobile code security *the mobile platform policy dictates to whom an application can talk to*.
- In the multi-application smart card domain *the smart card platform is neutral but the applications dictate by whom they can(not) be called*.

We propose an addition to the smart card security architecture that preserves the security level of the smart card when the content of the card changes, so the card itself can verify security properties of the applications when the following changes happen:

- Loading of a new application on the platform.
- Update of an existing application on the platform.
- Removal of an existing application from the platform.
- Update of the platform security policy.

The changes that we address in this deliverable are **conservative extensions** of the system, that is to say, extensions of the system that do not require deletions or changes of parts of the system. Consequently, we concentrate in this deliverable on the problem of arrival of a new application on the card and incremental updates of the applets on the platform, the other types of changes will be addressed later in the Deliverable 6.4.

## 1.2 Multi-application Smart Card Architecture

Smart cards are strongly constrained in terms of memory and computation resources. Standard Java virtual machines and runtime environments are not suitable for such systems because of mechanisms such as garbage collector or threads that are too greedy for resources, but also because of the very large API that contains unneeded features (AWT, Swing, reflection, etc.).

Java Card is a hardware-independent technology for multi-application smart cards, which allows post-issuance updates of the card contents [32, 13]. It is the most popular solution for smart cards today, because it brings the convenience of Java language to the smart card platform. Java runtime environments and virtual machines deployed on smart cards follow the Java Card specifications [3] from Sun, which is a subset of standard Java with some additional frameworks related to the underlying hardware (APDU, cryptographic abilities, etc.) and to the Java Card application model. There exist two main branches of Java Card specifications, 2.x and 3.x, with strong differences. Specifications of Java Card 3.x virtual machine and environment are closer to standard Java specifications than Java Card 2.x, but Java Card 3.x is quite recent and not yet adopted by manufacturers. For this reason, we chose to focus on Java Card 2.x specifications. Java Card architecture consists of several layers as illustrated in Figure 1.2:

- a device hardware,
- an embedded (native) operating system,
- a Java Card Runtime Environment (JCRE) built on top of the embedded operating system.
- the applications installed on the smart card, that are called *applets*.

Applets, before being loaded on the smart card, are converted on the terminal by Java Card Virtual Machine (JCVM) interpreter into a CAP file, that is a binary executable representation of the Java classes that compose the applet. The JCRE is responsible for managing and executing applets. It is composed by a JCVM interpreter, native API, Java API, and an application installer. The installer downloads and installs the applications on the card. To load an application, the installer interacts with an off-card installation program which transmits the CAP file via a card acceptance device (CAD). Once received the CAP file, the installer, optionally, checks the signature of the file to ensure integrity and to prove the identity of the application provider. Then, the installer saves the content of the file into the card's persistent memory, resolves the links with other applets already present on the card, creates an instance of the applet and registers it to the JCRE. Then on-card part of JCVM, which consists of a bytecode interpreter, executes the code contained in the CAP file.

Java Card applications, *applets*, are identified by a unique *Application IDentifier (AID)* that are assigned by the *International Standards Organization (ISO)* as defined in the ISO-7816 standard.

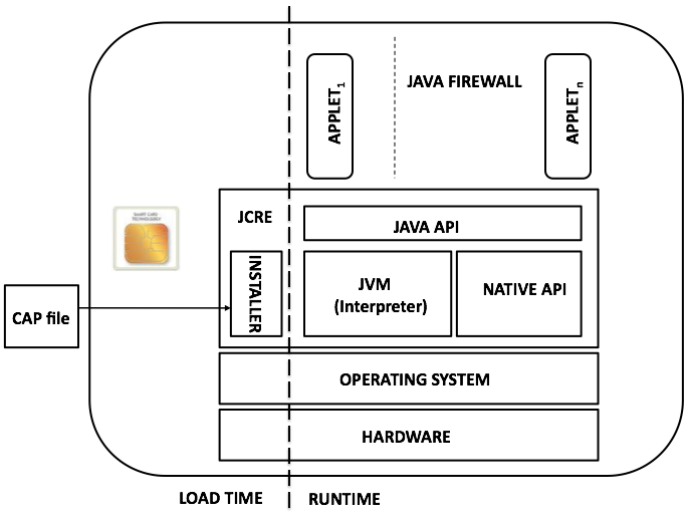


**Figure 1.2:** Java Card Architecture

The applets installed on the smart card belong to different packages (contexts) and they are isolated by the Java Card firewall. Java Card applets can receive commands from outside the card through a specific protocol called APDU, but they cannot directly interact with each other. The firewall allows only applets that belong to the same context to access the respective methods. If an applet (*server*) wants to share data with another applet (*client*) from a different context, it has to implement a *shareable interface* which defines a set of methods that are available to other applets. These methods (they are also called *services*) are the only methods of the server applet that are accessible through the firewall. The firewall will pass the call to the shareable interface method from the client applet to the server applet. In the current security model the server has an access control list of the applications, that are allowed to use this method, embedded into the server code. If the client is in the list, the access is granted and the server passes through the firewall the reference to its service. If the client is not in the list, it only gets `null`. As well, if there is no server applet installed on the card, or the called service is not provided, the firewall will also pass `null` to the client.

Technically speaking, applications developed for Java Card are instances of subclasses of the `Applet` class, and every shareable object is an instance of a class that implements the `Shareable` interface from the Java CardAPI. Access to objects shared by an application is not controlled by the Java Card environment; each application is entirely responsible for the access control to its shared objects relying on AIDs of requesters.

This security mechanism is not very flexible, as any changes to the security policy require to change the code of the applet (potentially introducing new bugs) and can be easily compromised by the evolution of the smart card platform. The JCRE allows to install and update the applets on the card even after the issuance. If the client has been updated, the server will still grant it an access, though now it cannot really be sure of the trustworthiness of the client. Also, if the server really needs to update its access control list, the server's owner can do it only at a very high price of full card recertification.

GlobalPlatform specifications [2] are generally used to enhance interoperability and security of Java Card environments deployed on open (for post-issuance updates) smart cards, especially those dealing with sensitive information such as SIM cards or credit cards. Indeed, GlobalPlatform proposes standardized and secure features for post-

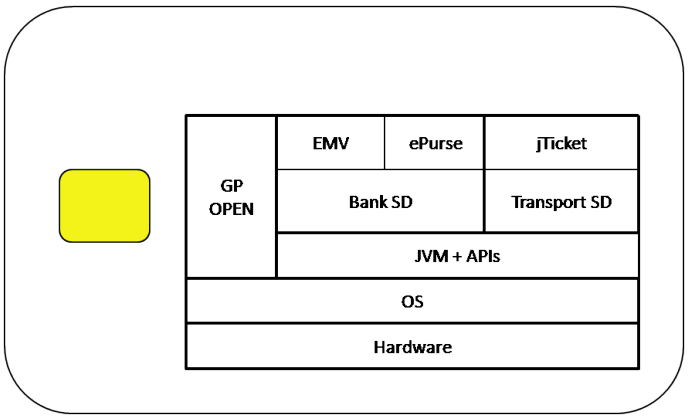


**Figure 1.3:** GlobalPlatform + Java Card Architecture

issuance (un)installation of applications through encrypted communication channels as well as secure application management thanks to the *security domains* architecture. A security domain is a privileged application, that stores cryptographic keys of its provider and manages some applications of the same provider. All the communications of these applications with the terminal are held through the security domain. Typically, each stakeholder on the platform has its own security domain. Security domains can be as well organized hierarchically. In the same way as Java Card applets, each security domain is uniquely identified by an AID, and its access is governed by a *public key infrastructure*. The set of initially available security domains is defined by the card manufacturer. GlobalPlatform specifications also include the possibility for the card issuer to create supplementary security domains after issuance. Intentionally, GlobalPlatform specifications do not clearly define permitted/forbidden interactions between applications according to their installation domains in order to let implementations fit specific needs and objectives.

**Example 1.2.1** *Typical GlobalPlatform on top of Java Card architecture is presented in Figure 1.3. There are two security domains installed on the platform, namely Bank security domain and Transport security domain. Bank security domain belongs to a stakeholder* $\mathtt{Bank}$*, who is also the card issuer, and manages two applications:* $EMV$*, that is a Europay – MasterCard – VISA standard banking applet, and* $ePurse$*, which is an electronic purse for small transactions. Transport security domain belongs to a stakeholder* $\mathtt{Transport}$ *and it manages* $jTicket$ *application.*

## 1.3 Security properties

### 1.3.1 Security Properties of the Case Study

Our targeted scenario of evolution is software update, specified in the POPS scenario description. General security property that we address is Information protection, as described in [6]:

**Citation 1.3.1 (Information Protection)** *The applications on the card must be "isolated" (segregation), that means no illegal access to the data from one application to another. For that several security policies are described and assumed to be implemented on the card, like the Java Card firewall (access control implemented by the virtual machine) or the security domains of GlobalPlatform. Therefore, some properties must be verified,*

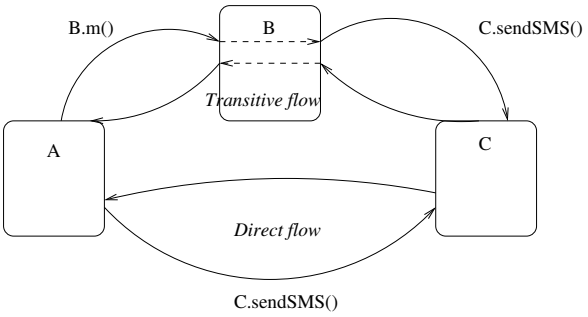


**Figure 1.4:** Control flow properties

*when an applet is added on the card, like the consistency of the security domain hierarchy, the non-violation of the information flow policy implemented on the card, etc.*

In this deliverable, we focus mainly on general information flow properties, that can be of two categories:

**Control flow properties** that can informally be expressed as:

- "*An application $A$ is allowed to invoke the operation $m_1$ but not the operation $m_2$*".

These properties express who is allowed to call "dangerous" methods such as access to operating system properties or sending of SMS for example.

**Non-interference properties** that can informally be expressed as:

- "*secrets of applications cannot leak to public observers*"

This non-interference property is often called information flow in the literature (see [39] for a survey). We add on top of the non-interference property some rules to control exchange of secret informations, like for example:

- "*secrets of applications can be transmitted to authorized parties*".

Chapters 3, 4, and 5 will focus on control flow property, whereas Chapter 6 will consider non-interference.

### 1.3.2 Control flow property

Control flow properties have been widely studied for many years, in fields like distributed computations [20] or for behavioral specification in VHDL [34]. More recently, they have been used to express security properties and in a recent work they were applied to small autonomous devices like Android phones [19].

We distinguish three different kinds of control flow that can be desired:

**Direct control flow** allows to verify that an application (or an element of a system) only calls services which it is allowed to call;

**Transitive control flow** adds to the previous one the transitivity property: an application that is not allowed to call a service cannot delegate the call to another application.

**Global control flow** adds to the previous ones a notion of interaction between services: an application can be allowed to call a service but not allowed to call a sequence of services.

The first one is sufficient to check in a very light way the use of services that are offered by trusted applications. For example, in a Unix system, a user is not allowed to access the properties of the system, but he is allowed to call services offered by the system that will provide him some answers. Direct control flow is the property that Java Card firewall enforces, but, as we have shown, without big flexibility for evolution. The second property is necessary in a multi-applicative context to avoid collusion between applications. The third one is necessary to express restriction on the usage of the device, for example: it is not allowed to send more than three SMS, or it is not allowed to use camera and send `http` requests.

Let us consider example of Figure 1.4: application $A$ is not allowed to call the service `sendSMS` of application $C$. Then, there is an illegal direct control flow if $A$ tries to call $C$, and an illegal transitive control flow if $A$ calls the service `m` of $B$, which calls the service `sendSMS` of application $C$.

### 1.3.3 Non-interference property

*Information flow analysis*, as defined in [39], detects how data may flow between variables, focusing on data manipulations of primitive types. This analysis is used to check data propagation in programs with regard to security levels and aims to avoid that programs leak confidential information: observable/public outputs of a program must not disclose information about secret/confidential values manipulated by the program. *Non-interference* [24] defines the absence of illicit information flows by stating that public outputs of a program remain unchanged if the secret inputs are modified. Data are labeled with security levels, usually $high$ for secret/confidential values and $low$ for observable/public variables.

Some information flows arise from assignments (direct flow), others from the control structure of a program (implicit flow). For example, the code `l=h` generates a direct flow from `h` to `l`, while `if(h) then l=1 else l=0` generates an implicit flow also from `h` to `l`. If `h` has a security level $high$ and `l` $low$, then the examples are insecure, as secret data can be inferred by the reader of `l`.

## 1.4 Verification techniques properties

To achieve the above-mentioned goals, we provide on-device verification techniques that must ensure the next results:

**Autonomous on-device verification** : devices must be able to verify autonomously the mobile code they receive,

**Minimal re-verification load** : additions to the system must be handled without re-verifying the already loaded code (or with a minimal re-verification).

Moreover, as we want to be able to address small devices, the memory footprint of the verification algorithms has to be considered as an important criterion:

**Low footprint** depending on their footprint, verification algorithms will be available for different kinds of devices.

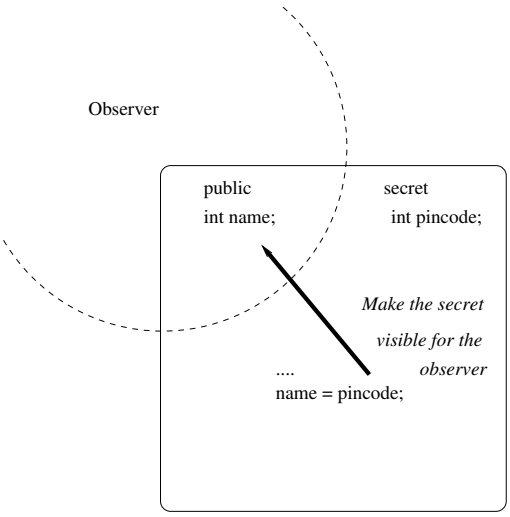


**Figure 1.5:** Non-interference properties

Thus the proposed techniques must be at least incremental and compositional, and if possible use algorithms with low complexity and small memory needs.

Security properties enforcement can be achieved by:

- *verification at compilation time*, which is not compatible with our target environment, as for Java Card compilation is done on the terminal and the deployment context is unknown,
- *testing*, which is not suitable on device when new mobile code arrives,
- *policy in-lining*, which is impossible to perform on a smart card,
- *monitoring at execution time*, which would penalize the performance,

thus, we use in the following :

**Verification at loading time** that is before running the new code. Load time verification will only increase the time to make an application available on the system, but not the runtime performance.

## 1.5 Verification Workflows

Depending on the verified properties and on the device itself, the device may be able to establish the properties or not. We will see in the following chapters that the control flow properties can be established by devices as small as a smart card, whereas non-interference properties need algorithms including fixed point computation that are too heavy for such systems. In this section we provide descriptions of the workflows for verification being performed only by device and by device and terminal together.

### 1.5.1 On-device Verification

The best solution for security is to enable the card with the means to verify new application only by itself and independently from every stakeholder. For reasonably small security properties, such as control flow, the card security mechanisms can indeed be extended with load time application verification.

We have already discussed, that smart cards with GlobalPlatform on top of Java Card have certain security mechanisms: firewall between applications, signature-checking mechanism, it can be as well enhanced with partial bytecode verification on device [16, 30]. In order to perform loading time verification, these mechanisms have to be extended. One can consider delegating the verification to the card manager, that is to OPEN (card manager on GP) or JCRE, for example, installer or JCVM interpreter.

Extension of the card entities can be done in different ways depending on trust assumptions and approach chosen by smart card vendor. For example, GP implementation may be proprietary or it can be formally proved and certified, so the vendor will not want to touch it. It can be also the other way around, that Java Card implementation, or parts of it, cannot be changed. Thus it is not possible to propose one single answer for the question "which entity on the card will perform the verification process?".

In this deliverable for on-device verification process we propose different approaches for security mechanisms' extension. These approaches assume different trust and availability assumptions and have their own pros and cons.

All the solutions assume to extend the loading mechanisms of the platform in a way which ensures that verifications cannot be bypassed. Using full Java, or Java Card 3.0, we can add our verification algorithm in a user defined classloader. Using Java Card 2.x, we can use the JCRE specific entity, namely, installer, that performs the installation of the applications, resolves links with other applications and writes the CAP file contents into the persistent memory. It seems natural to add the loading time verification responsibilities to the installer. If Java Card implementation cannot be modified, we have identified two ways for integrating our verification algorithms in GlobalPlatform that fulfill these requirements. A rough solution is to modify the GlobalPlatform implementation, and more precisely the LOAD command implementation. It therefore requires access to the source code of the GlobalPlatform implementation to be deployed. As an alternative to this invasive solution, GlobalPlatform includes a way to define *Controlling Authorities* in charge of enforcing the security policy on all application code loaded into the card. Each Controlling Authority gets a special security domain in which it can install its own verification processes called by the GlobalPlatform framework during the loading process. These security domains need to have certain privileges in order to have access to loaded applications and perform the verification process.

The improved loading mechanism can analyze the bytecode, going line by line, and make a decision if this applet can be installed on the platform or not. If at some point it will meet a potentially problematic instruction (e.g. an unauthorized call to a method of another applet), the mechanism can stop the installation process and make a roll-back to the previous card state (the contents of ROM memory cannot be erased, but all the links to the unwanted applet data are removed from the registry [13]).

If the GlobalPlatform implementation cannot be modified, we can propose to modify the Java Card installer, in order to include our verification algorithms. The platform loading mechanism cannot be bypassed and, consequently, the improved one can indeed guarantee that all the applets on the card will be verified.

Still it might happen, that it is not possible to include verification process into Java Card installer, or the installer is not trusted. In this case we propose another approach.

Two components: ClaimChecker and PolicyChecker are added to the JCRE. This architecture is provided in Figure 1.6. The ClaimChecker checks that the code of the new applet is compliant with its contract (which is a formal specification of applet security-relevant behavior). The PolicyChecker, instead, verifies, that the contract is compliant with the security policy of the smart card system. Separation of these steps gives the platform

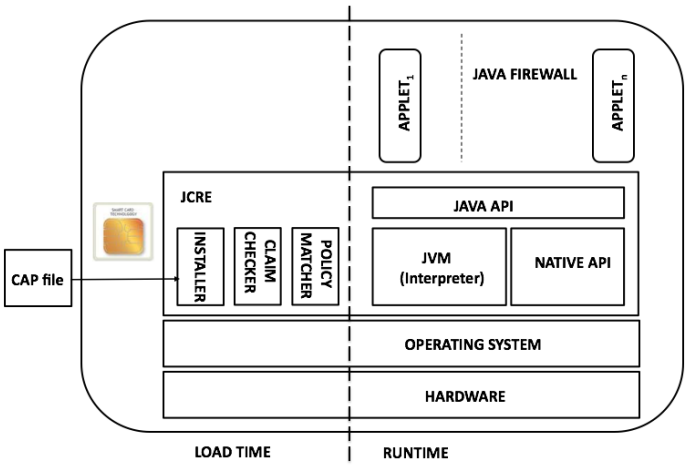


**Figure 1.6:** Security-by-Contract Architecture

more flexibility in implementation of the security policy and updates of it. We describe this approach in more details in Chapter 3.

### 1.5.2 Lightweight Verification

Since the smart card has very restricted memory, a lot of interesting and important security properties, including non-interference, cannot be verified only by the card. In this case, we propose to use a technique known as "Lightweight bytecode verification" that has been developed in [38] for Java bytecode type verification. This technique, closely related to proof-carrying code [35], is interesting because:

1. it provides the developers with tools, that help them to test the security of their software unit before loading it on the embedded environment,

2. it allows an open system to verify code received from an untrusted source without relying on a third party, even if it does not have enough power to compute the proof itself.

Lightweight verification relies on the simple idea that it is easier to verify a result already computed. It consists of two phases, as depicted in Figure 1.7:

**Offboard phase** (on the producer side) which is assumed to have access to infinite resources (typically a personal computer compared to a small device), which computes the type correctness and annotates the bytecode with some proof elements,

**Onboard phase** (on the consumer side) which, at loading time, verifies the proof annotations obtained during the first phase. The annotations are embedded within the code and verified. The verification operation is linear in the code size and uses constant memory.

The first phase is performed by a *prover*, while a *verifier* embedded on the device (again into JCRE or OPEN) certifies the second phase. This technique has been developed for type checking and it relies on the lattice structure of types and on unification operations on this lattice. To be used in other contexts, the technique must be extended and adapted to the desired properties.

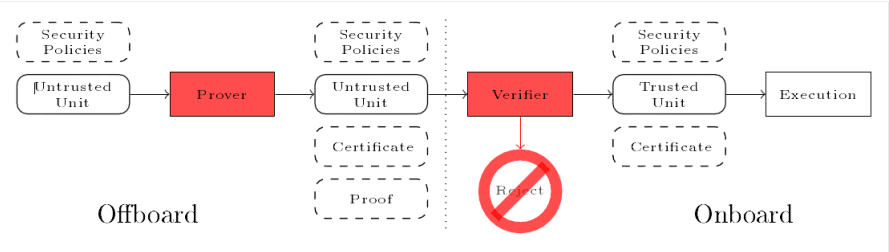


**Figure 1.7:** Lightweight verification

## 1.6 Threat model

In this deliverable, we consider open systems supporting multiple applications from different providers and control flow and non-interference properties as desired properties. Thus, threats are personified by malicious applications that try to be installed on the system (or to be updated) and to break the information protection property in one of the following forms:

- invoking services (directly or indirectly) that they are not allowed to invoke,
- relaying service invocations from A to a service of B when A is not allowed to call this service (acting as a malicious proxy),
- stealing confidential information of other applications,
- obtaining (even with authorization) confidential information of an application A to give the information to an application B that is not allowed to access information of A.

All these threats can result either from a malicious application, or from a buggy application. A buggy application can contain functional bugs, or for example be an application that has been developed for another system with a different policy. In this case, the bug is not in the application itself, but the policy is not compliant with the system. On the other hand, a malicious application is an application that has been developed in order to attack a system. In the general case, a malicious application can be an honest application that has been tampered by an attacker after it has been sent by the stakeholder and before it is loaded on the system: this cannot occur in our case since we are relying on the GlobalPlatform mechanism that are tested by Work Package 7.

## 1.7 Related Works

Several methods have been proposed to strengthen the security of open multi-application systems [46], but few are devoted to small devices such as mobile phones or smart cards. The work of Huisman *et al.* [25, 26] is close to the control flow policies mentioned in this deliverable. They have presented a formal framework and a tool set for compositional verification of application interactions on a multi-application smart card. Their method was based on construction of maximal applets, w.r.t structural safety properties, simulating all the applets respecting these properties. Model checking techniques were then used to check whether a composition of two applets $A$ and $B$ respects some behavioral safety property. They proposed a compositional verification for control-flow-based properties for Java Card applets. Their model was proved to be sound and complete for properties

expressed in simulation logic, and was implemented in a tool that accepts Java bytecode as input. However, their proposal was not dedicated to be embedded on small devices, and it lacks concrete means for ensuring deployment of verified applets only, especially in a context where it is crucial that composition is done on-device. Barthe *et al.* [10] had a similar approach to Huisman *et al.* that relies on temporal logic and lacks the same features. Stack inspection mechanisms [11] could also be used to enforce control flow policies but this monitoring approach cannot be concretely applied without significant runtime speed penalty.

Concretely, few proposals have been made for static on-device verification of some properties, especially security properties. Leroy [31] has exhibited problems encountered to verify properties, like type-checking or stack under/over-flow, on-device on Java bytecode. In fact this approach is analogous to the one described in this chapter: some transformations done off-device help on-device verification at installation-time in order to allow secure execution without runtime speed penalty of non-signed applets. Ghindici *et al.* [21, 22] have followed a similar approach for secure information flow purposes. They were relying on a sound and complete model to compositionally verify on-device proofs (inspired by proof-carrying code methods [35]) computed off-device by the STAN tool and then embedded in Java bytecode. As in Leroy's work, this does not require code signing.

Avvenuti et al. in [8] have proposed a tool for off-card verification of Java bytecode files, that might be later installed on the platform. Their method explored the multi-level security policy model and the theory of abstract interpretation.

Girard in [23] have suggested to associate security levels (clearances) to application attributes and methods, using traditional Bell/La Padula model. Bieber et al. have adopted this approach in [12] and have proposed a technique based on model checking for verification of actual information flows. The same approach was used by Schellhorn et al. in [41] for their formal security model for operating systems of multi-application smart cards.

Certification based-mechanisms were in fact widely used in small devices. Code signing is the simplest kind of such mechanisms. This process is used in several existing devices such as smart cards (SIM cards, credit cards, etc.) but also iPhone, Android and JavaME. Although code signing provides a simple but strong way to ensure both origin and integrity of code, it is not designed to deal with application behaviors, especially their interactions with each other.

Relying on code signing, many systems restrict authorized behaviors according to the origin of the application, as it is for instance recommended by the *Application Security Framework* of the *Open Mobile Terminal Platform* [4]. A first simple example comes from the Android platform where an application can specify that its shared services must only be invoked by code signed with the same signature.

*Security-by-contract* approaches [17, 18, 28] rely on code signing to bind application code with some *contracts*. A contract is a set of relevant features of an application (*e.g.*, use only HTTPS connections, use a shared service of application $A$, etc.) that can include interactions with other applications and/or its host platform. Developers must generally write their application contracts since features described in contracts are generally very difficult to infer automatically from application code. In practice, contracts have to be enforced on-device by additional mechanisms whose complexity depends on the expressiveness of supported features. Security-by-contract main principle is for instance used in the Android platform where some irrevocable permissions can be granted at installation-time: to use the camera, for example, an application needs to have the corresponding permission which is granted if and only if it is mentioned in its manifest (contract) so that

the user can decide to grant it or not until application is uninstalled.

McDaniel *et al.* [36] have introduced more refined features in Android contracts along with the SAINT framework to enforce them at run-time, which is not appropriate for constrained devices. This approach was far more sophisticated than the model of control flow policies presented in this chapter, but suffers from a strong drawback, as it is also the case for Kirin rules [19] for detecting malwares on Android: they are not sensitive to collusion of applications and can thus be cheated. Since the Android platform is mostly Java-based, our model could easily be instantiated and integrated in this platform to complement these techniques and to provide the anti-collusion feature.

# 2. Notations

## 2.1 Mathematical notations

We denote by $\wp(S)$ the powerset of $S$.

### 2.1.1 Functions

For a function $f$, we denote by $dom(f)$ the domain of $f$, by $f[x \mapsto e]$ the function $f'$ such that $f'(y) = f(y)$ if $y \neq x$ and $f'(x) = e$, and by $f_{|S}$ the function $f$ whose domain has been restricted to $S$.

For a set of sets $C$ and two functions $f_1 : A \longrightarrow C$ and $f_2 : B \longrightarrow C$, $f_1 \sqcup f_2$ is defined by:

$$\begin{array}{rlcl} f_1 \sqcup f_2 : & A \cup B & \longrightarrow & C \\ & x & \longmapsto & f_1(x) \text{ if } x \in A \text{ and } x \notin B \\ & x & \longmapsto & f_2(x) \text{ if } x \in B \text{ and } x \notin A \\ & x & \longmapsto & f_1(x) \cup f_2(x) \text{ if } x \in A \cap B \end{array}$$

### 2.1.2 Graphs

We consider finite directed graphs with edges labeled by elements of a set $L_E$ and vertices labeled by elements of a set $L_V$. A graph $G$ is given by a triple $(V, E, \varsigma)$ where $V$ is its set of vertices, $E \subseteq V \times V \times L_E$, its set of labeled edges, and $\varsigma$ is a mapping from $V$ to $L_V$.

In the graph $G = (V, E, \varsigma)$, the edge from the vertex $u$ to $v$, labeled by $l$ is denoted by $(u, v, l)$ or $u \xrightarrow{l} v$. We write $adj_G(u, l)$ to denote the set of adjacent vertices of $u$ in $G$, reached by an outgoing edge labeled by $l$. A vertex $u$ is a *leaf* (respectively, a *root*) if for any vertex $v$ and label $l$, $(u, v, l) \notin E$ (resp. $(v, u, l) \notin E$). Graphs may also have unlabeled vertices ($\varsigma$ is undefined for all vertices). In this case, we simply write graphs as a pair $(V, E)$. Graphs may also have unlabeled edges, which are edges for which labeling is irrelevant and can be ignored (we may assume that all edges are labeled with the same dummy symbol from $L_E$). In this case, edges are pairs of vertices $(u, v)$.

A vertex $v$ is reachable from $u$ in a graph $G$ if there is a path (a sequence of edges, of length greater or equal to 0, leading) from $u$ to $v$; we denote by $Reach_G(u)$ the set of vertices reachable from $u$ in $G$. We define $G\lfloor u \rfloor$ as the subgraph of $G = (V, E)$ as $(Reach_G(u), E \cap (Reach_G(u) \times Reach_G(u) \times \mathcal{L}, \varsigma_{|Reach_G(u)}))$.

The union of two graphs is a partial operation defined only when their vertex labellings are compatible; let $G_1 = (V_1, E_1, \varsigma_1)$ and $G_2 = (V_2, E_2, \varsigma_2)$. Then, the union $G_1 \cup G_2$ is defined only if for all vertices $v$ from $V_1 \cap V_2$, either $\varsigma_1(v)$ or $\varsigma_2(v)$ is undefined, or both are defined and $\varsigma_1(v) = \varsigma_2(v)$. When $G_1 \cup G_2$ is defined, it is equal to the graph $(V_1 \cup V_2, E_1 \cup E_2, \varsigma_{1\cup 2})$ where for all $v$ in $V_1 \setminus V_2$ (respectively in $V_2 \setminus V_1$), $\varsigma_{1\cup 2}(v)$ is equal

to $\varsigma_1(v)$ (resp. to $\varsigma_2(v)$) and for all $v$ in $V_1 \cap V_2$, $\varsigma_{1\cup 2}(v)$ is equal to $\varsigma_1(v)$ if $\varsigma_2(v)$ is undefined and to $\varsigma_2(v)$ otherwise. Note that when considering graphs with unlabeled vertices, the union of two of such graphs is always defined.

The graph inclusion ($G_1 \subseteq G_2$) is defined accordingly, that is, $V_1 \subseteq V_2$, $E_1 \subseteq E_2$ and for all $v$ in $V_1$, either $\varsigma_1(v)$ is undefined or $\varsigma_1(v) = \varsigma_2(v)$.

For a graph $G = (V, E, \varsigma)$, the graph $G[(u, l) \mapsto v]$ agrees with $G$ except that all the edges of the form $(u, u', l)$ in $E$ are replaced by a unique edge $(u, v, l)$, and the graph $G[u \mapsto l]$ denotes the graph $(V, E, \varsigma[u \mapsto l])$.

For a graph $G = (V, E)$, $V' \in V$ and $V'' \subseteq V$:

$$path(G, V', V'') = \{v_0 v_1 \dots v_n \in V^* \mid v_0 \in V', v_n \in V'' \wedge \forall 0 < i \leq n, (v_{i-1}, v_i) \in E\}.$$

A strongly connected component of a graph $G = (V, E)$ is a subgraph $G' \subseteq G$, $G' = (V', E')$, where $V'$ is a maximal subset of $V$ such that for each pair $v_1$, $v_2$ of vertices of $V'$, there exists a word $v_1 v_{i_0} \dots v_{i_n} v_2$ in $path(G, V', V')$ and each $v_{i_k} \in V'$ (and symmetrically from $v_2$ to $v_1$).

## 2.2 Notations for programs

### 2.2.1 Object-oriented notations

In this deliverable, we consider on-device Java systems that support several applications. As in any Java system, an application is implemented by a set of classes. We write $Classes$ the set of all classes names.

The services which we wish to control access to will be implemented as methods. We will write $\mathcal{M}$ the set of all methods names and $meth(A)$ the set of methods of some application $A$, that is to say the set of all the methods available for objects of its classes. Thus, the set $meth(A)$ contains fully qualified names of methods, namely elements of the form $A.C.m$ to denote the method $m$ that is available for objects of class $C$ in application $A$.

Note that the method $A.C.m$ might be defined and implemented in a super-class of $A.C$ – which might itself be in some other application – and simply inherited. Later on in this work, the distinction must be made. In order to avoid confusion, we use the following notations for inheritance.

The set $Classes$ is equipped with an inheritance relation $\leq$; $C_1 \leq C_2$ means that the class $C_1$ inherits from $C_2$ or is $C_2$ (and so we use $C_1 < C_2$ when $C_1$ is a strict sub-class of $C_2$). The set $\mathcal{M}$ is also equipped with an inheritance relation, also written $\leq$. When a class $C_1$ of application $A_1$ inherits from some class $C_2$ in application $A_2$ and redefines a method $m$, we write $A_1.C_1.m < A_2.C_2.m$. We use $\leq$ whenever the method is either inherited or redefined (*i.e.* basically whenever one class inherits from the other). So the method $A.C.m$ of $meth(A)$ is actually defined in the class $C$ of the application $A$ when there does not exist any $A'.C'.m$ such that $A.C.m \leq A'.C'.m$ unless $A.C.m < A'.C'.m$ or $A.C = A'.C'$.

### 2.2.2 Application level definitions

Applications can make methods available through offered interfaces, in the sense of shareable interfaces of Java Card or for example provided interfaces of the Corba Component Model. Methods that do not belong to these interfaces can only be called inside the application. Thus methods of the application are partitioned into two groups: offered

methods and internal methods. By analogy to the GlobalPlatform denomination, we consider a set $Domains$ of security domains of the system, even if it is not related to any precise technology. By analogy to the GlobalPlatform denomination, we also use the term "interface" in the specific meaning of Java Card shareable interfaces. Thus, for an application $A$, $shareable(A)$ denotes the subset of methods of $meth(A)$ that are offered in shareable interfaces.

### 2.2.3 Graphs of the programs

For a method $m$, $P_m$ is its instruction list. We assume this list to be indexed from $0$ to $|P_m| - 1$, where $|P_m|$ is its size. Hence, we denote by $P_m[i]$ the $i+1$-th bytecode of the method $m$.

Let us first define the call graph of a set of methods.

**Definition 2.2.1 (Call graph for a set of methods)** *Let $M$ be a set of methods. A call graph for $M$ is a finite graph $CG = (V, E, \varsigma)$ with $\varsigma$ from $V$ to $M$, $E \subseteq V \times V \times \mathbb{N}$, such that: for each $v_1 \in V$, for each instruction $P_{\varsigma(v_1)}[i] =$ `invoke` [1] $m$, for all $m' \leq m$ that might be called at runtime by this instruction, there exists a $(v_1, v_2, i) \in E$ such that $\varsigma(v_2) = m'$.*

**Definition 2.2.2 (The context-insensitive call graph)** *The* context-insensitive call graph *of a set of methods $M$ is a call graph $(V, E, \varsigma)$ such that $\varsigma$ is a bijection.*

**Definition 2.2.3 (Intraprocedural control flow graph)** *The* intraprocedural control flow graph *of a method $m$ is an unlabeled graph $CF_m = (P_m, E)$ such that $(i, i')$ belongs to $E$ if either $(1)$ $i' = i + 1$ and $P_m[i]$ is different from a* `return` *bytecode and from* `goto` $a$*, or $(2)$ $i' = a$ and $P_m[i]$ is equal to* `goto` $a$ *or to a comparison*[2] *bytecode* `if`cmp $a$*. Hence, there is no edge going out of the vertex $i$ if $P_{C.m}[i]$ is a* `return` *bytecode. An exit-point in a control flow graph is a vertex without successor.*

To deal with control flow properties, we have to be able to compute for a given method, the set of methods it invokes directly of indirectly. The set is statically computed and thus collects methods that are invoked on static types.

**Definition 2.2.4 (Invoked methods)** *Let $m$ be a method. Then the set of methods that $m$ invokes is the smallest set $\mathcal{I}(m)$ such that:*

$$\mathcal{I}(m) = \mathcal{I}^{direct}(m) \cup \mathcal{I}^{indirect}(m)$$

*with*

- $\mathcal{I}^{direct}(m) = \{m' \mid \exists 0 \leq i < |P_m|, P_m[i] =$ `invoke` $m'', m' \leq m''\}$ *is the set of methods invoked in the bytecode of $m$,*
- $\mathcal{I}^{indirect}(m) = \bigcup_{m' \in \mathcal{I}^{direct}(m)} \mathcal{I}(m')$*,i.e. the set of methods (directly and indirectly) invoked by methods directly invoked by $m$.*

It is fairly easy to see that $\mathcal{I}(m)$ is exactly the set of all the methods that are reachable (by 1 or more transitions) from $m$ in the call graph of the system (Definition 2.2.2).

[1] Where `invoke` stands for any invocation bytecode like `invokevirtual`, `invokestatic`, ...

[2] Or similar cases.

## 2.3 On-device system

### 2.3.1 Definition of the system

**Definition 2.3.1 (On-device system)** *An on-device system $\mathcal{S} = (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P})$ is a tuple where:*

- *$\mathcal{D}$ is a finite set of domains,*
- *$\mathcal{A}$ is a finite set of applications,*
- *$\mathcal{U}$ is the subset of $\mathcal{A}$ of applications with* unsolved *dependencies,* i.e.*, applications that cannot run because they need to use some services that are not available for the moment, the applications of $\mathcal{A} \setminus \mathcal{U}$ are said* selectable*, meaning that all their requirements are met and they can be run,*
- *$\delta$ is the deployment function from $\mathcal{A}$ to $\mathcal{D}$ that defines where the applications are deployed,*
- *$\mathcal{P}$ is the security policy of the system.*

**Remark.** The function $\delta$ described in this definition can naturally be extended from applications to methods:

$$\begin{array}{rccc} \delta: & \mathcal{M} & \longrightarrow & \mathcal{D} \\ & A.C.m & \longmapsto & \delta(A) \end{array}$$

### 2.3.2 Changes: addition of new elements in a system

A system $\mathcal{S} = (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P})$ can be updated by the installation of new applications:

$$install(A, d, \mathcal{S}) = (\mathcal{D}, \mathcal{A} \cup \{A\}, \mathcal{U}', \delta[A \mapsto d], \mathcal{P}')$$

with $\mathcal{P}'$ ($\mathcal{P} \sqsubseteq \mathcal{P}'$) the new policy that takes into account the new application.

The application will become *selectable* as soon as all its dependencies will be resolved, this means that all the called methods must be methods of selectable applications, and that all the inherited classes must also be classes of selectable applications. Thus,

$$\begin{array}{lll} \mathcal{U}' = \mathcal{U} & \text{if} & \forall A.C.m \in \mathcal{M}_A, \forall A'.C'.m' \in \mathcal{I}(A.C.m), A' \in \mathcal{A} \setminus \mathcal{U} \\ & \text{and} & \forall A.C.m \in \mathcal{M}_A, (A.C \leq A'.C') \Rightarrow (A' \in \mathcal{A} \setminus \mathcal{U}) \\ = \mathcal{U} \cup \{A\} & & \text{otherwise} \end{array}$$

A system can also be updated by addition of a new domain:

$$add(d, \mathcal{S}) = (\mathcal{D} \cup \{d\}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P}).$$

At last a system can be updated by addition of new policy elements. This will be described in the following chapters.

# 3. Direct Control Flow

In this chapter we consider a verification of a direct control flow for service invocations of applets on Java Card. In the proposed model we do not consider GlobalPlatform security domains hierarchy, but the model can be easily extended to include them. The considered smart card system (platform) is based only on Java Card concepts and the verification process is based on two new components added to JCRE. The mechanism of verification implements Security-by-Contract approach, in which every application has a contract, that is a formal specification of its security-relevant (in our case direct control flow-relevant) behavior.

## 3.1 Introduction

To identify the requirements for an extension of the security mechanisms on Java Card in presence of evolution we look at the GlobalPlatform specification [27]. In essence, GP is a specification for the middleware, that can run on top of Java Card and provide more security mechanisms for application management (for inter-application communications GP relies on JCRE).

GlobalPlatform specification has explicit requirements for maintaining security and functionality of the applications on the card during evolution:

- [27, p.92], Card and Application Management: Only code and data not referenced by another entity on the card may be deleted.
- [27, p.24] Trusted Framework security requirements: Each Trusted Framework on the card shall check the application access rules of the inter-acting Applications according to their respective privileges;

The security mechanisms on the card have to support these requirements and maintain security and functionality of the applications.

The requirement for deletion of only not referenced entities, however, can be overridden for services on Java card, because Java Card specification allows the calls to the services to be checked only at run-time. Still, this requirement shows, that the applications shall be functional and their functionality can include services provided by another applications. Thus, before deleting an application, we need to check if some other applications rely on the services it provides. For example, for electronic purse – transport application smart card, transport applet is useless without the payment application. But, instead, for bank – mileage applets tandem, if the bank applet will be removed from the card, the mileage application will still work perfectly, because its core functionality is to count miles during check-ins in the airports, and accumulating miles for banking transactions is a “bonus” functionality. Java Card technology allows applets to be functional even if they have embedded calls to the services of other applets and these services are not

provided on the card. But some services, as we have shown, can actually be necessary and we are capturing this property in our model.

**Example 1** *As an example we can consider the multi-application smart card described in the Deliverable 1.1 [7], it is implemented according to the specifications of Java Card and GP. We assume two security domains on the platform:* $Bank$ *and* $Transport$ *with applications* $EMV@Bank$, $ePurse@Bank$ *and* $jTicket@Transport$. *The architecture is provided in Figure 1.3. Application* $EMV$ *has one service* `transaction`, $ePurse$ *has one service* `payment`, $jTicket$ *has no services.*

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

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

The basic idea of our proposal is that we will add a contractual component (contract) to each application detailing its security policies and its claim on the usages of other resources on the platform (the latter can be also extracted from the bytecode). Two new components of JCRE, as illustrated in Figure 1.6, namely, the ClaimChecker and the PolicyChecker will perform correspondingly the comparison of the applets contract with its bytecode and the comparison of the contract with the smart card platform security policy, that will is related to direct control flow. Our extension to Java Card security mechanisms can help both stakeholders in the example 1 to make sure that their security and functionality requirements are met when changes occur.

**Example 2** *Due to a termination of the agreement, the* $Bank$ *security domain may update the policy forbidding access to the* $Transport$ *owner. In the new security architecture this change is much more easier to implement than before, since the security policy is no longer embedded in the application code, but is provided separately in the* PolicyChecker.

Further details on architecture are provided in Section 3.5.

## 3.2 Formal Security Model

One can provide various formal security models for specification of security-relevant and functionality-relevant behavior of the applications and for the platform security policy. We present in this deliverable a formal model based on the shareable interfaces of the applications.

We now introduce a definition of a smart card platform, that is a similar concept of Definition 2.3.1 of the on-device system. As well we will further introduce a notion of evolved (updated) platform that is similar to the notion of update of a system of the Sec. 2.3.2

**Definition 3.2.1 (Platform)** *Platform* $\Theta$ *is a tuple* $\langle \Delta_{\mathcal{A}}, \Delta_{\mathcal{S}}, \mathcal{A}, \mathsf{shareable}(), \mathsf{invoke}(), \mathsf{sec.rules}(), \mathsf{func.rules}() \rangle$, *where* $\Delta_{\mathcal{A}}$ *is a domain of applications;* $\Delta_{\mathcal{S}}$ *is a domain of services;* $\mathcal{A} \subseteq \Delta_{\mathcal{A}}$ *is a set of selectable applications installed (deployed) on the platform;* $\mathsf{shareable}(), \mathsf{invoke}() : \Delta_{\mathcal{A}} \to \wp(\Delta_{\mathcal{S}})$ *are association function for the services on the platform;* $\mathsf{sec.rules}() : \Delta_{\mathcal{A}} \times \Delta_{\mathcal{S}} \to \wp(\Delta_{\mathcal{A}})$ *and* $\mathsf{func.rules}() : \Delta_{\mathcal{A}} \to \Delta_{\mathcal{S}}$ *define security policy of the applications.*

In order to simplify the notations, we use the application as an index of the function instead of being an argument. So we write $\text{shareable}_\text{A}$ instead of shareable(A); $\text{invoke}_\text{A}$ instead of invoke(A); $\text{sec.rules}_\text{A}(\text{s})$ instead of sec.rules(A, s); $\text{func.rules}_\text{A}$ instead of func.rules(A). The set of all services installed on the platform will be denoted as $\mathcal{S} = \bigcup_{A \in \mathcal{A}} \text{shareable}_\text{A}$.

The last two functions of the platform, sec.rules() and func.rules, define respectively the security policy and the functionality policy of each application. For every application $A \in \mathcal{A}$ sec.rules(A, s) = $\text{sec.rules}_\text{A}(\text{s})$ is a mapping, that defines for each service of application $A$ which other applications on the platform are authorized to call it. The functional rule func.rules(A) = $\text{func.rules}_\text{A}$ specify the set of services on the platform that $A$ needs in order to be functional. If some service, that is declared in $\text{func.rules}_\text{A}$ is absent on the platform, the firewall will return `null`, thus making $A$ crash or malfunction.

We assume that $\text{shareable}_\text{A} \cap \text{shareable}_\text{B} = \emptyset$ for all $A, B \in \Delta_\mathcal{A} : A \neq B$. Also sec.rules() is a partial function, so that sec.rules(A, s) is only defined if $s \in \text{shareable(A)}$. In other words the security policy of an application can only talk about the application's own services. Also we assume that $\text{shareable}_\text{A} \cap \text{func.rules}_\text{A} = \emptyset$ for all $A \in \mathcal{A}$.

We denote applications already installed on the platform as $A$ or $A_i$ (elements of $\mathcal{A}$) and the application that is affected by a change (or new applet being installed) as $B$. We use notation $A.s$ for service $s$ of application $A$. Also we assume that application and service names are unique.

Let us now define an evolution of the platform, for the types of changes we have chosen (see Section 1.1 for details):

**Definition 3.2.2 (Evolved Platform)** *Let $B$ be an application affected by change, an evolved platform $\Theta'$ from a platform $\Theta = \langle \Delta_\mathcal{A}, \Delta_\mathcal{S}, \mathcal{A}, \text{shareable}(), \text{invoke}(), \text{sec.rules}(), \text{func.rules}() \rangle$ is defined as follows:*

1. *Addition of a new application $B$: $\mathcal{A}' = \mathcal{A} \cup B$.*
2. *Update of an installed application $B \in \mathcal{A}$. This update can be expressed as a sequence of simpler updates. For all applications $A \in \mathcal{A}$, $A \neq B$ holds:*
   (a) *Addition of a service $s$ to $\text{shareable}_\text{B}$: $\text{shareable}'_\text{A} = \text{shareable}_\text{A}$ and $\text{shareable}'_\text{B} = \text{shareable}_\text{B} \cup \{\text{s}\}$;*
   (b) *Addition of a service $s$ to $\text{invoke}_\text{B}$: $\text{invoke}'_\text{A} = \text{invoke}_\text{A}$ and $\text{invoke}'_\text{B} = \text{invoke}_\text{B} \cup \{s\}$;*
   (c) *Addition of an access authorization to service $s$ for application $C$ by $B$: $\text{sec.rules}'_\text{A}() = \text{sec.rules}_\text{A}()$ and $\text{sec.rules}'_\text{B}() = \text{sec.rules}_\text{B}() \cup \{\text{C}\}$;*
   (d) *Addition of a necessary service $s$ to $\text{func.rules}_\text{B}$: $\text{func.rules}'_\text{A} = \text{func.rules}_\text{A}$ and $\text{func.rules}'_\text{B} = \text{func.rules}_\text{B} \cup \{\text{s}\}$;*

*Unless specified above, other components of the platform are unchanged.*

Our goal is to keep the platform secure after every update and to this extent we need to define the notion of executions of applications on the platform. We approximate executions of applications as sequences of service invocations. So we denote by $A :: s$ an invocation of service $s$ by application $A$, where $s$ is a service of $A$. $A :: B.t$ is an invocation of service $B.t$ by application $A$. We also assume that in each single moment only one application is active and combine the traces of all applications synchronously. We use $\sigma$ for an execution's trace and $\sigma_1 \cdot \sigma_2$ for a concatenation of executions $\sigma_1$ and $\sigma_2$.

**Definition 3.2.3 (Possible executions on the platform)** *Set $\Sigma_\Theta$ of all possible executions of $\Theta$ is constructed as follows:*

- ***Axiom*** $\epsilon \in \Sigma_\Theta$*, where $\epsilon$ is an empty trace.*
- ***Rule 1*** $\frac{\sigma \in \Sigma_\Theta, \quad A \in \mathcal{A}, s \in \mathsf{shareable_A}}{\sigma \cdot A :: s \in \Sigma_\Theta}$
- ***Rule 2*** $\frac{\sigma \in \Sigma_\Theta, \quad A,B \in \mathcal{A},\ B.s \in \mathsf{invoke_A} \cap \mathsf{shareable_B}}{\sigma \cdot A :: B.s \cdot B :: s \in \Sigma_\Theta}$

The intuitive meaning is that if $A$ is an applet on the platform $\Theta$, then it can become an active applet and invoke its service $s \in \mathsf{shareable_A}$. If there is a service $B.s$ such that some application $A$ invokes this service during its work, then a sequence '$A$ calls $B.s$' – '$B.s$ is invoked' can appear in the execution of $\Theta$. We can now define the security properties to be preserved across updates.

**Definition 3.2.4 (Secure Platform)** *Let $\Theta$ be a platform, then for all applications $A,\ B \in \mathcal{A}$ and all services $s \in \mathcal{S}$*

**Security in traces** *: if $B \notin \mathsf{sec.rules_A(s)}$, then for all executions $\sigma, \sigma'$ an execution $\sigma \cdot B :: A.s \cdot A :: s \cdot \sigma' \notin \Sigma_\Theta$.*

**Functionality in traces** *: if $B.s \in \mathsf{func.rules_A}$, then exists an execution $\sigma \cdot A :: B.s \cdot B :: s \in \Sigma_\Theta$.*

*$\Theta$ is **secure** if security and functionality are maintained across updates.*

## 3.3 Specifications of Contract and Components

The contract in a S×C approach for mobile platforms is a specification of an application that will help honest application providers to describe what their application is going to do and the platform owner to check if these applications are not malicious [15]. In Figure 1.1 the contract (specification) provided by application developer should be checked for compliance against the application code, so it seems natural to extract it from the code, like in [22], or use proof-carrying-code techniques [35]. Here, we need also to give the application owners the possibility to impose restrictions on the callers of their services. This information cannot be extracted from/compared to the code, since it is not about the application itself but about other external applications. So we extend the S×C definition from [17] by dividing the contract $\mathsf{Contract_A}$ of application $A$ into two parts:

- $\mathsf{Claim_A}$ describes the actual behavior of application $A$. $\mathsf{Claim_A}$ contains two parts: the services that are called by $A$ and the published services of $A$, essentially this is an information that can be automatically extracted from the code. For instance, if one has $A$'s source code, then the list of called services can be comprised from getShareableInterfaceObject methods, and the list of $A$'s services – from the methods that implement Shareable interface.
- $\mathsf{AppPolicy_A}$ is declared by application providers, security domain owner or controlling authority about the desirable behavior of other applications on the platform w.r.t. this application.

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

**Definition 3.3.1 (Contract)** *For application $A$ its* $\text{Contract}_\text{A}$ *is a pair* $\langle \text{Claim}_\text{A}, \text{AppPolicy}_\text{A} \rangle$*, where*

- $\text{Claim}_\text{A} = \langle \text{Provides}_\text{A}, \text{CallsA} \rangle$*, and* $\text{Provides}_\text{A}, \text{Calls}_\text{A} \in \wp(\mathcal{S})$*;*
- $\text{AppPolicy}_\text{A} = \langle \text{sec.rules}_\text{A}, \text{func.rules}_\text{A} \rangle$*, and* $\text{sec.rules}_\text{A} : \mathcal{S} \rightarrow \wp(\text{A})$*,* $\text{func.rules}_\text{A} \in \wp(\mathcal{S})$*.*

*We assume dom(*$\text{sec.rules}_\text{A}$*)* $\subseteq \text{Provides}_\text{A}$ *and* $\text{func.rules}_\text{A} \subseteq \text{Calls}_\text{A}$*.*

$\text{Provides}_\text{A}$ is a declared set of services of application $A$ and $\text{Calls}_\text{A}$ is a declared set of services of another applets that $A$ calls into its code. The security and functional rules are now instantiated in the AppPolicy of each application i.e. the policy of the platform is broken down to each individual application and then written into its Contract.

$\text{Contract}_\text{A}$ provided according to Definition 3.3.1 is *well-formed*. We can expect that there might be some malicious applications with not well-formed contracts, where they could try to declare, for example, sec.rules for services, that are provided by other applications. So the verification process normally has to include a check that provided Contract is well-formed.

We now enlarge the generic platform security policy given as a set of rules on the platform $\Theta$, adding to the policy also Claim of all applications on the $\Theta$.

**Definition 3.3.2 (Platform Security Policy)** *Security policy of the platform $\mathcal{P}$ consists of the contracts of all the applications $\mathcal{A}$ on the platform:* $\mathcal{P} = \bigcup_{A \in \mathcal{A}} \text{Contract}_\text{A}$

Thus policy $\mathcal{P}$ is compositional and it is jointly defined by all the stakeholders. Any accepted update of the platform $\Theta$ will influence $\mathcal{P}$.

Now we will specify the behavior of the ClaimChecker and the PolicyChecker for various types of updates that we consider in this deliverable.

**Definition 3.3.3 (Claim Checker)** *A* ClaimChecker *algorithm for a new (updated) application $B$ is a PTIME algorithm that returns true iff the conditions below are true:*

1. $\text{shareable}_\text{B} \subseteq \text{Provides}_\text{B}$.
2. $\text{invoke}_\text{B} \subseteq \text{Calls}_\text{B}$.

ClaimChecker *is precise if* $\text{shareable}_\text{B} = \text{Provides}_\text{B}$ *and* $\text{invoke}_\text{B} = \text{Calls}_\text{B}$*.*

Intuitively, the ClaimChecker has to check that application code and Claim are compliant: all services that $B$ has are declared in $\text{Provides}_\text{B}$ and all services that $B$ actually invokes are declared in $\text{Calls}_\text{B}$. If we want to capture functional dependencies we have to require the ClaimChecker to be precise.

We do not show in this deliverable how to construct the ClaimChecker. One can use, for example, the techniques for parsing the bytecode on device that are provided in Chapter 4.

**Definition 3.3.4 (Static Policy Checker)** *A* StaticPolicyChecker *algorithm for platform $\Theta$ and changed application $B$ is a PTIME algorithm, that returns true iff for all applications $A, B \in \mathcal{A}$ and services $s \in \mathcal{S}$*

- ***Security on contract level****: if* $B.s \in \text{Calls}_\text{A}$ *then* $A \in \text{sec.rules}_\text{B}(\text{s})$*.*
- ***Functionality on contract level****: if* $B.s \in \text{func.rules}_\text{A}$ *then* $s \in \text{Provides}_\text{B}$*.*

Intuitively, the StaticPolicyChecker checks that $\text{AppPolicy}_\text{A}$ for every application $A \in \mathcal{A}$ is satisfied on the platform.

We can now define an optimized checker that works for verification of an update for an application $B$. The update is reflected in the $\text{Contract}_\text{B}$ if there if a change in the set of called or provided services, or there is a change in security or functional rules. Such an update can be broken down into atomic updates, that correspond the targeted evolutions of the platform.

**Definition 3.3.5 (Optimized Contract-Policy Compliance Checker (Policy Checker))** *An Optimized* PolicyChecker *(or just* PolicyChecker*) is a PTIME algorithm for certification of change in the application $B$, that returns true iff the conditions below are true for all applications $A \in \mathcal{A}$ on the platform:*

1. *Installation of a new applet $B$ on the platform.*
   - $B \in \text{sec.rules}_\text{A}(\text{Provides}_\text{A} \cap \text{Calls}_\text{B})$*;*
   - $\text{func.rules}_\text{B} \subseteq \bigcup_{\text{A} \in \mathcal{A}} \text{Provides}_\text{A}$*;*
   - $A \in \text{sec.rules}_\text{B}(\text{Provides}_\text{B} \cap \text{Calls}_\text{A})$*;*
2. *An update of some application $B \in \mathcal{A}$.*
   - *(a) Addition of a service $s$ to* $\text{Provides}_\text{B}$*:* $A \in \text{sec.rules}_\text{B}(\text{s} \cap \text{Calls}_\text{A})$*;*
   - *(b) Addition of a service $s$ to* $\text{Calls}_\text{B}$*:* $B \in \text{sec.rules}_\text{A}(\text{Provides}_\text{A} \cap \text{s})$*;*
   - *(c) Addition of an authorization rule for some application $C$ to a service $s$ of $B$ to* $\text{sec.rules}_\text{B}$*: return true;*
   - *(d) Addition of a service $s$ to* $\text{func.rules}_\text{B}$*:* $s \in \bigcup_{A \in \mathcal{A}} \text{Provides}_\text{A}$

The PolicyChecker rejects any change if security or functionality can be broken.

After an accepted change, when both the ClaimChecker and the PolicyChecker returned true, the policy $\mathcal{P}$ has to be adjusted. The adjustment is defined accordingly to what the PolicyChecker did. For example, for installation of a new application $B$ it is $\mathcal{P}'$= $\mathcal{P} \cup \text{Contract}_\text{B}$.

### 3.3.1 Different approaches for accepting updates

Another benefit of separating the ClaimChecker and the PolicyChecker from any component of JCRE and from each other is more flexibility in adopting various strategies for accepting updates. The PolicyChecker we describe in this deliverable has the strategy to maintain stable security and functionality on the smart card platform and to reject all potentially threatening updates (principle "already installed applets have more privileges than a new one"). So if an update of some applet $B$ can potentially break security (for example, $B$ wants to add a call to some applet $A$, and this call is not authorized by $A$), the card will reject this update.

A different approach might assume a hierarchy of applications so that a change in a "more important" application can still go ahead even if less important applications break down. This is a reasonable strategy for current smart card business model, where the card issuer is always more privileged than any other stakeholder on the card. Consequently, the card issuer wants always to be able to update her applets, even though they might break security or functionality of other applets on the card. In this case, the

PolicyChecker has to apply a different algorithm, in which it will identify the applications that cause the problem and make them unselectable (unavailable to be executed). This may start a process of cascade deselection of applications on the platform, but this is a price of maintaining security and functionality.

We also point here to potential problems for resolving the conflicts between applications. For example, the applet $jTicket$ from our running example 1 is preventing the application $ePurse$ of another stakeholder from being removed from the card (if the PolicyChecker adopts the strategy from Definition 3.3.5), essentially, locking it. Consequently, this is a conflict of interests and the card (or at least the business model) should have a way to resolve such conflicts. As a solution, the PolicyChecker can have a special algorithm with a support for resolution of such conflicts, again searching for the applet to blame in the problem and evaluating both parties' positions, with a priority given to the owner of the conflicted service. For example, in the aforementioned case of $jTicket$ and $ePurse$, the $ePurse$ is allowed to be removed, because its owner can take decisions on her services, and the $jTicket$ is made unselectable, since it will not be functional anyway.

## 3.4 Validation of the Security-by-Contract Approach

We now state that our approach to verification of updates does actually deliver the right security guarantees.

**Lemma 3.4.1** *If* $\exists \sigma \cdot B :: A.s \cdot A :: s \in \Sigma_\Theta$*, then* $s \in \text{shareable}_A \cap \text{invoke}_B$.

**Proof.** The only possible rule to obtain such an execution is Rule 2 of Definition 3.2.3. By condition of this rule, $s \in \text{shareable}_A \cap \text{invoke}_B$. □

**Theorem 3.4.2** ClaimChecker *and* StaticPolicyChecker *are sound and they returned true for* $\forall A \in \mathcal{A}$*, then* $\Theta$ *is secure.*

**Proof.** Platform $\Theta$ is secure if security and functionality in traces are maintained. If the StaticPolicyChecker is sound and returned true, then security and functionality on contract level are maintained. We need to proof that security and functionality on contract level imply security and functionality in traces.
Proof by contradiction. Let security on contract level holds, but security in traces is not maintained. Then $\exists A \in \mathcal{A}, \exists s \in \text{shareable}_A, \exists B \in \mathcal{A}$ such that $B \notin \text{sec.rules}_A(s)$ AND there exists $\sigma \cdot B :: A.s \cdot A :: s \in \Sigma_\Theta$. Then by Lemma 3.4.1 $A.s \in \text{invoke}_B$. According to definition of the ClaimChecker, the previous fact implies, that $\exists A, B \in \mathcal{A}, A.s \in \mathcal{S}$ $A.s \in \text{Provides}_A$and$A.s \in \text{Calls}_B$. Application B $\notin$ $\text{sec.rules}_A$A.s by our assumption, but it is a contradiction with the definition of security on contract level, that is maintained on the platform.
Let functionality on contract level holds, but functionality in traces is broken. Then $\exists A, B \in \mathcal{A}, \exists A.s \in \text{func.rules}_B$ such that $\forall \sigma$ execution $\sigma \cdot B :: A.s \cdot A :: s \notin \Sigma_\Theta$. Analogously to the proof of security in traces, we can make a conclusion that either $A.s \notin \text{invoke}_B$ or $s \notin \text{shareable}_A$. According to the definitions of application Contract and the ClaimChecker, $\text{func.rules}_B \subseteq \text{Calls}_B = \text{invoke}_B$, under assumption that the ClaimChecker is precise, and, consequently, $A.s \in \text{invoke}_B$. Since functionality on contract level is maintained, it implies (from the fact $A.s \in \text{func.rules}_B$) that $A.s \in \text{Provides}_A$. Hence, under assumption that the ClaimChecker is precise, $s \in \text{shareable}_A$. □

**Theorem 3.4.3** *If* $\Theta$ *is secure,* ClaimChecker *and* PolicyChecker *are sound and they returned true, then the evolved platform* $\Theta'$ *is secure after targeted types of changes.*

**Proof.** If the StaticPolicyChecker returns true on the evolved platform $\Theta'$, then by the Theorem 3.4.2 $\Theta'$ is secure.
In order to show that the StaticPolicyChecker returns true on $\Theta'$, we have to reason by cases for all possible $\Theta'$ described in Definition 3.2.2. For each type of evolved platform and since $\Theta$ was secure before the change, it is easy to show that the StaticPolicyChecker run on $\Theta'$ is equivalent to the PolicyChecker run, performed the case correspondingly the type of change occurred. Thus if the PolicyChecker returned true, the StaticPolicyChecker will return true on $\Theta'$. □

If we look at Figure 1.1, Theorem 3.4.2 tells us about the initial status of the card. The soundness of the ClaimChecker is needed for a successful first step. The PolicyChecker's soundness is required for the second transition. The Java Card firewall reliability assumption (blocking non-shareable interface calls) is necessary as an underlying assumption to show that the two checks are sufficient. We do not deal with the problem of inlining the policy, because it is unfeasible on the card.

## 3.5 Security-by-Contract Architecture

Let us discuss now more precise workflow for an extension of the Java Card security mechanisms that we have proposed in this chapter. The architecture extends the existing Java smart card architecture with two additional components: the ClaimChecker and the PolicyChecker (see Figure 1.6).

When a new applet has to be loaded on the card, the terminal sends the CAP file of the applet to the installer. The CAP file contains the binary code of the applet and its Contract. When the installer receives the CAP file of the new applet, it saves the file in the card's memory. The installer might also perform an optional signature check for verification of the source of an update. If the signature is valid, then the ClaimChecker extracts the set of claims $\text{Claim}_A$ from the contract, and verifies that the claims are compliant with the applet's code. If this is the case (the ClaimChecker returned true), the PolicyChecker checks the applet's contract against the platform policy $\mathcal{P}$ that is stored in the PolicyChecker. If the applet's contract complies with the platform policy (the PolicyChecker returned true), then the PolicyChecker updates the security policy and the installer creates an instance of the applet and makes it selectable. Otherwise, the applet is rejected and, if possible, erased from the memory. This is the workflow our formal model supports. In the S×C-enhanced security architecture, when a method of the applet is called by another applet, the Java firewall simply checks that the method belongs to the shareable interface of the applet and does not perform run-time checks of the applet's privileges for an access to the services.

In general, it is possible to implement the ClaimChecker and the PolicyChecker components as off-card entities, using some Trusted Third Party (TTP) computational capabilities, though it will require reconsideration of the approach as a lightweight verification. But it may be interesting, for example, for a complicated specification of application behavior (contract), if the PolicyChecker will be implemented on the side of the issuer or mobile phone platform (for SIM applications), and the JCRE will provide a cryptographic support for communications with TTP. Another solution is to replace the ClaimChecker with a digital signature verification for some trusted application providers. In this case signature of an applet has a concrete semantics, represented as an application contract.

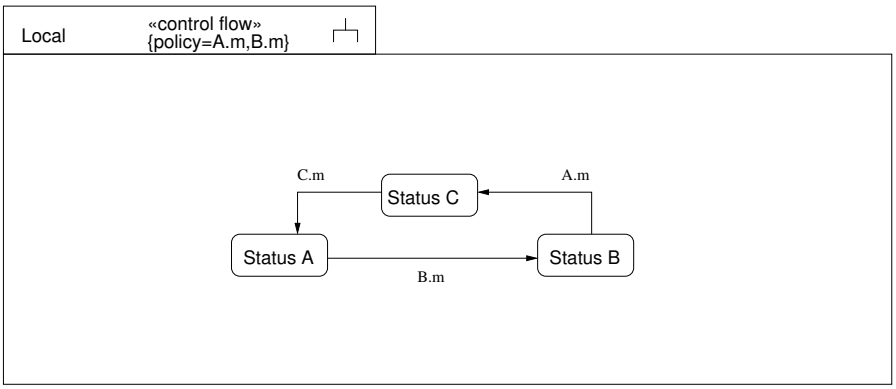


**Figure 3.1:** Example of a diagram annotated with $<<$ control flow $>>$

In a full industrial setting it could be possible to incorporate the components into Java Card run-time environment. Such approach will increase the speed of verification.

## 3.6 Link with WP4 Notation

In model-driven security engineering, the main goal is to detect unwanted behavior of a system before its actual implementation. Prudent secure engineering techniques to achieve this are encapsulated for example in the UML security extension UMLsec [29]. To deal with evolution, WP4 has introduced a further extension to UMLsec called UMLseCh [42]. The techniques described in this chapter to deal with direct control flow can be complemented at design time by means of the stereotype $<<$ control flow $>>$ and the UMLseCh stereotypes to represent evolution.

The stereotype $<<$ control flow $>>$ can be applied to a package containing a state-chart diagram. It has an attached tag $\{\mathsf{policy} = \mathsf{list\ of\ methods}\}$, specifying a policy for the application described within the diagram. Semantically, $<<$ control flow $>>$ checks if there are transitions within states where a method not contained in $\{$policy$\}$ is called. In this case, the diagram fails the direct control flow requirement. The $\{$policy$\}$ for an application $B$ can be represented in terms of Contract: it is exactly the set $\bigcup_{A \in \mathcal{A}} \mathsf{sec.rules}_{\mathsf{A}}^{-1}(\mathsf{B})$. The transitions within the states have to correspond the set of actually called services $\mathsf{invoke}_{\mathsf{B}}$.

For example, in Figure 3.1, there is an example of a diagram violating the requirements of $<<$ control flow $>>$, since the application considered is only allowed to call services $A.m$ and $B.m$, and it calls $C.m$. In this figure Status A, Status B and Status C are some statuses of the modeled application and $A.m$, $B.m$, $C.m$ are some services of another applications on the platform.

In the case of having a policy respecting UML diagram representing the behavior of the application, this proof obligation for the implementation can be checked using the on-device verification techniques presented in this chapter. WP4 plans to provide verification results at the UML model level for the preservation of $<<$ control flow $>>$ under evolution by M24. The evolution stereotypes included in the UMLseCh extension to UMLsec allow to specify changes in the system and in the policy. The expected result is that given a model respecting the $<<$ control flow $>>$ semantics and the evolution `delta` as specified in the UMLseCh stereotypes, a decision about the security preservation of the evolved model can be made without re-running the verification from scratch.

The approaches for on-device direct control flow verification we suggest in this chapter and the model level evolution verification that will be provided in WP4 for the $<<$ control flow $>>$ stereotype are complimentary. At design time with the help of UMLseCh models one can verify that the model of an application respects certain (also evolving) policies. However, the ClaimChecker and the PolicyChecker together make sure, that the application's code indeed respects these policies.

## 3.7 Conclusions

In this chapter we have proposed an extension of the Java Card security mechanisms for open multi-application smart cards that makes it possible to perform verification of direct control flow security property on the card. This extension adds two components to the JCRE, the ClaimChecker and the PolicyChecker. In a nutshell, all applications are arriving with specifications of their behavior and their requirements on other applications on the platform. These requirements merged together create platform security policy. The card can verify autonomously whether they are acceptable and then either reject or accept the change.

We have shown that the S×C approach can be used in a dynamic environment when applications come and go and the security policy can be changed, and it can help solving security and functionality challenges. The direct control flow properties were specified in terms of services used and provided by applications on the platform.

The framework we have proposed can check the absence of non-authorized accesses to the application services (direct control flow) and the functional stability of the applications on the card in terms of necessary services provided by other applets. Another contribution of the framework is the flexibility in specification and updates of the platform security policy. In today's security model access control is embedded into the application code and it is very costly to update it. Placing it separately from the code allows the stakeholders to deploy the updates more easily.

The S×C approach for on-device verification can easily be extended for removal of applications or decremental updates (removal of services) and it will make a part of the Deliverable 6.4 that is due at M24.

# 4. Transitive Control Flow

**Preliminary remark** This chapter has been submitted for publication in a conference. The content of the article has been only slightly modified in order to use uniform notations across the report and to merge related works.

In this chapter, we tackle the issue of controlling application interactions including collusion in Java-based systems running on open, constrained devices such as smart cards or mobile phones. We present a model specially designed to be embedded in constrained devices to verify at loading time that interactions between applications abide by the security policies of each involved application without resulting in run-time computation overheads; this models deal with application (un)installations and policy changes in an incremental fashion. We propose some algorithms, fit to be embedded, to ensure the security of a system is preserved along those updates and we show how this framework can be instantiated in GlobalPlatform with Java Card platform, widely used by smart card manufacturers to illustrate the security enhancements provided by our framework on a multi-application use case. The example used as support follows the requirements of the POPS use case as described in the deliverable D1.1, but is not directly the POPS use case, the basic idea has been enriched, keeping all its key elements but focusing on the very topic of the chapter.

## 4.1 Introduction

Ubiquitous devices such as mobile phones and smart cards tend to be multi-application devices and to support post–issuance installation of applications. Applications are also evolving to take advantages of these new features: they are shifting from standalone designs to a collaborative model where they provide services to other applications. This new shape of ubiquitous devices causes major *security* concerns for end-users and applications vendors.

In a multi-application environment, the system generally provides some basic mechanisms to application developers for controlling access to services they provide to other applications. In open contexts, *i.e.*, when some applications can be added (or changed) to the system post issuance, *access control* mechanisms are however very limited and insufficient to fulfill requirements of international security agreements such as Common Criteria Recognition Agreement [1]. Actually, formal methods are required to reach the highest certification levels of these agreements.

Formal methods, that are currently applied to verify some security properties (confidentiality, integrity, application interactions, safety, etc.), have been developed to analyze a finite set of applications known before issuance of the device. These methods are not appropriate in open contexts even if they can be embedded on-device. Indeed it could lead to useless resource consumption simply because security models and algorithms have not been tuned for this purpose.

In this chapter, we focus on the design of on-device verification of absence of undesired control flow between applications. For open multi-application devices, it is mandatory to verify that each new incoming application will not create some undesirable control flow paths because of application-specific restrictions. We describe a model and some efficient algorithms to enforce control flow policies for open multi-application Java-based systems. The model is designed to avoid collusion between applications to exploit services provided by other applications. The main originality of this model is to define an incremental verification process at installation-time with few memory requirements and computation overheads in order to be fully embeddable in constrained devices.

We first introduce the model of application interactions we use to formally define control flow policies (Section 4.2), and algorithms for efficient verification of these policies on-device at installation time without any third party (Section 4.3). We then describe a concrete integration of our model in Java-based smart cards (Section 4.4), and illustrate the security enhancements achieved on such devices with a concrete use case (Section 4.5).

## 4.2 Model for control of service calls between applications

We first give a high level abstraction of an open multi-application Java-based system. This model is not directly related to any specific technology and is abstract enough to be adapted to any Java environment, as it is illustrated later in this chapter.

Informally, we consider a system composed of several *domains*. A domain can harbor *applications*, and it is assumed that access to domains is controlled by the underlying system. An application is a bunch of classes that can provide some *shared services*. Basically, a shared service is just a method. In the system, applications residing in different domains can only call each other through their *shared services*. An application provides, for each of its shared services, the exhaustive list of all the domains from which this service might be used. The exhaustiveness of this list is here to be understood even up to relays: in cases where two applications act in collusion, we might see a situation in which an application *Bad* wants to call a service *S* but resides in some domain from which calling *S* is prohibited; in order to do so, it might use some application *Rel*, be it renegade or simply buggy, that resides in some domain *S* grants access to, to relay calls from *Bad*.

To sum up, the model we propose here to secure calls of shared services is based on those domains which simply define a partition of applications such that:

- classes of applications that belong to the same domain can share information and call each other's methods,
- while communications between classes of applications that belong to different domains must be explicitly allowed for the code to be permitted to run.

### 4.2.1 Systems and security policies

A security policy for a system (or a subsystem) is a set of security rules in which every method of the system (resp. the subsystem) is given exactly one security rule and where security rules define from which domains a given method can be called.

**Definition 4.2.1 (Security policy)** *Let $M$ be a set of methods and $D$ be a set of domains. A security policy for the set of methods is*

$$p = (M, D, rules),$$

*where* $rules$ *is a mapping* $rules : M \longrightarrow \wp(D) \cup \{\top\}$*, where* $\top$ *stands for "*any domain*".*

The value $\top$ is needed to allow evolutions of the system after deployment: the difference between $\top$ and $D$ is that $D$ is the set of all the *known* domains. When the system evolves, some new domains might be introduced from which some previously deployed methods might want to accept calls without any extra update of its security policy. $\top$ conveys that meaning of all domains, even yet-unknown ones.

The security concerns of the system are expressed in terms of authorizations to invoke methods of the system.

**Definition 4.2.2 (Security policy of the system)** *The security policy of the system, denoted by* $\mathcal{P}$ *is a security policy* $(\mathcal{M}, \mathcal{D}, rules)$*.*

### 4.2.2 Semantics of the security policy

We build on those definitions of security policies in order to define formally the security property we want to assert on systems. Basically, in secure systems, an application in some domain $d$ will be enabled to invoke indirectly (*i.e.* trigger an invoke from a method in some other domain) or directly only services explicitly expecting calls from $d$. The semantics must take into account dynamic dispatch, where the method of any sub-class of the static type can be actually called.

**Definition 4.2.3 (The context-insensitive call graph of a system)** *The* context-insensitive call graph *of a system* $\mathcal{S} = (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P})$ *is denoted by* $CG(\mathcal{S})$ *and is the context-insensitive call graph of the set of methods of the system.*

**Definition 4.2.4 (Secure system)** *A system* $\mathcal{S} = (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P})$ *is* secure *if and only if for all* $(A_1.C_1.m_1, A_2.C_2.m_2) \in \mathcal{M} \times \mathcal{M}$*, there exists a path from* $A_1.C_1.m_1$ *to* $A_2.C_2.m_2$ *in* $CG(\mathcal{S})$ *only if* $\delta(A_1.C_1.m_1) \in rules(A_2.C_2.m_2)$*.*

### 4.2.3 Generic security policies

When implementing an application $A$, the programmer has to decide which methods are critical and must therefore get a security policy rule. But the programmer may not be aware of the precise configuration on which the application will be deployed, in particular we do not want the set of domains to be fixed once and for all. So we allow programmers to define, for an application $A$, a set of security levels $\mathcal{L}_A$, that represents relevant security access levels for the application like *public*, *applications of the same domain*, *applications of commercial partners*, etc. The policy $P_A$ defines the access rules to the methods of the application, *i.e.*, the set of security levels from which this method can be called: $P_A = (\mathcal{M}_A, \mathcal{L}_A, rules_A)$.

To impose as little constraints as possible on the set of levels $\mathcal{L}_A$, the only requirement is the availability of an instantiating function mapping those levels to actual domains when the application is deployed:

$$inst_{A,d,(\mathcal{D},\mathcal{A},\mathcal{U},\delta,\mathcal{P})} : \quad \mathcal{L}_A \quad \longrightarrow \quad \wp(\mathcal{D})$$

As said at the beginning of this section, the intuition is to always allow communications inside one security domain. To that end, we require the images by $inst_{A,d,\mathcal{S}}$ to always include $d$[1].

[1] So the co-domain of the mapping really is $\wp(\mathcal{D}) \setminus \wp(\mathcal{D} \setminus \{d\})$.

That mapping is further extended to:

$$\begin{array}{rccl} inst_{A,d,(\mathcal{D},\mathcal{A},\mathcal{U},\delta,\mathcal{P})}: & \wp(\mathcal{L}_A)\cup\{\top\} & \longrightarrow & \wp(\mathcal{D})\cup\{\top\} \\ & L\in\wp(\mathcal{L}_A) & \longmapsto & \bigcup_{l\in L} inst_{A,d,(\mathcal{D},\mathcal{A},\mathcal{U},\delta,\mathcal{P})}(l) \\ & \top & \longmapsto & \top \end{array}$$

Then the policy of $A$ deployed on $\mathcal{S}=(\mathcal{D},\mathcal{A},\mathcal{U},\delta,\mathcal{P})$ is $P_A^i=(\mathcal{M}_A,\mathcal{D},inst_{A,d,\mathcal{S}}\circ rules_A)$.

**Remark.** The rules of the policy are a mapping from all the methods to the security levels. Obviously, giving a security policy for each method of the application can become a boring task. Thus it is possible to provide a tool that allows the programmer to give policies for methods of shareable interfaces and that computes the closure of the least relation $rules_A$ such that $rules_A(m)\subseteq rules_A(m')$ whenever $m$ invokes $m'$.

## 4.3 On-device algorithms

We devise now efficient embeddable algorithms to ensure that the system stays secure along its evolutions.

### 4.3.1 Addition of a new application

When a new application $A$ with the instantiated policy $P_A^i=(\mathcal{M}_A,\mathcal{D},inst_{A,d,\mathcal{S}}\circ rules_A)$ is loaded on the device in some domain $d$, the device has to verify that $A$ respects the policy of the system and, if the application successfully passes the verification, to update the system $\mathcal{S}=(\mathcal{D},\mathcal{A},\mathcal{U},\delta,\mathcal{P})$.

To accept the application $A$, we verify it class by class, method by method, following the standard Java loading mechanism. Thus, for each method $m$, we have to check that:

1. $m$ invokes only methods that grant access to applications in domain $d$,
2. every method of other applications that asks access to $m$ resides in some domain from which the rules for $m$ grant access.

Let us recall that $\mathcal{I}^{direct}(m)$ is the set of methods that $m$ directly invokes, namely the $m'$ such that an instruction `invoke` $m'$ appears in the code of $m$. To perform the verification, we have to be able to take into account, for a given method, the set of methods it invokes directly of indirectly (that is $\mathcal{I}(m)$). So all the rules and the algorithms we propose hereafter pay special attention to preserve the transitivity of the property they ensure.

**Definition 4.3.1 (On-device method verification)** *A method $m$ of a class $C$ of an application $A$ with a policy $P_A=(\mathcal{M}_A,\mathcal{L}_A,rules_A)$ passes the verification to be loaded on the system $\mathcal{S}$ in the domain $d$ if:*

1. $\forall A'.C'.m'\in\mathcal{I}^{direct}(A.C.m),\{d\}\cup inst_{A,d,\mathcal{S}}\circ rules_A(A.C.m)\subseteq rules(A'.C'.m')$
2. $\forall A'.C'.m', A.C.m\in\mathcal{I}^{direct}(A'.C'.m')\Rightarrow\{\delta(A'.C'.m')\}\cup\ rules(A'.C'.m')\subseteq inst_{A,d,\mathcal{S}}\circ rules_A(A.C.m)$
3. $\forall A'.C', A.C.m\leq A'.C'.m\Rightarrow rules(A'.C'.m)\subseteq inst_{A,d,\mathcal{S}}\circ rules_A(A.C.m)$

Those rules are dictated by the requirement of consistency of the system:

- Rules 1 ensures that all the methods invoked by the incoming method grant access to the domain in which the incoming application is being installed and to all the domains from which the incoming method will be expecting calls,
- Rules 2 ensures that all the methods calling the incoming method are in domains from which this is explicitly permitted,
- Rules 3 ensures that, when the incoming method overrides some inherited method, it still grants access to all the domains the overridden method did: this check handles in a simple way dynamic dispatch while still allowing the system to be open and further extended with applications unknown when the first applications are checked.

As mentioned above, the transitivity of the operations are ensured because the relation $\subseteq$ is obviously transitive. Those rules are translated into the Algorithm 4.3.1. As previously mentioned, Java loading mechanism works one class at a time. Since applications (and classes, for that matter) may depend on each other, the algorithm must keep some information about unsatisfied dependencies ($wait(A)$ is the set of all the applications $A$ is waiting for to become selectable) and collect all the constraints imposed on methods until they are actually loaded and the constraints can be checked ($unsolved(A.C.m)$ is the set of all the domains from which already-loaded methods call $A.C.m$). We also use another temporary function $ruleswait$, similar to $rules$, to store all the rules for the methods in applications waiting for some dependency.

Rule 1 is checked by the code in Lines 9 to 18: as stated before, the verification is performed only on directly invoked methods; if the invoked method is already loaded the consistency can be checked right away; if not, the check is postponed by using $unsolved$. Rule 2 is easier to check, on Line 6: all the already-loaded methods invoking the incoming one have registered their requirements to $unsolved$. Finally Rule 3 is tested on Line 1. In the event of failure of this check when the method is loading, we decide to resolve the conflict between already installed applications and the new one by rejecting the incoming code. This strategy could be refined by considering application removals in order to make way for the new one.

The Algorithm 4.3.2 wraps the complete verification of an incoming application. The main task to perform is to mark newly selectable applications as such, when they were waiting for the newly-loaded one, on Lines 10 to 16. The final operation of cleaning-up temporary data structures is not detailed here.

Once the methods of $A$ are verified, the application can be securely installed on the system. Thus the system has to be updated by the installation of the new application with the following function:

$$install((A, P_A), d, \mathcal{S}) = (\mathcal{D}, \mathcal{A} \cup \{A\}, \mathcal{U}', \delta[A \mapsto d], \mathcal{P}')$$

with

$$\mathcal{P}' = (\mathcal{M} \cup \mathcal{M}_A, \mathcal{D}, rules \cup inst_{A,d,\mathcal{S}} \circ rules_A).$$

The application will become *selectable* as soon as all its dependencies will be resolved, this means that all the called methods must be methods of selectable applications, and that all the inherited classes must also be classes of selectable applications. Thus,

$$\begin{array}{lll} \mathcal{U}' = \mathcal{U} & \text{if} & \forall A.C.m \in \mathcal{M}_A, \forall A'.C'.m' \in \mathcal{I}^{direct}(A.C.m), A' \in \mathcal{A} \setminus \mathcal{U} \\ & \text{and} & \forall A.C.m \in \mathcal{M}_A, (A.C \leq A'.C') \Rightarrow (A' \in \mathcal{A} \setminus \mathcal{U}) \\ = \mathcal{U} \cup \{A\} & & \text{otherwise} \end{array}$$

1: **if** $A.C$ inherits $A'.C'$ and $\exists A'.C'.m$ and $(rules \cup ruleswait)(A'.C'.m) \not\subseteq rules^i_A(A.C.m)$ **then**
2: **return** FAIL
3: **else if** $A.C$ inherits $A'.C'$ and $wait(A') \neq \emptyset$ **then**
4: $wait \leftarrow wait[A \mapsto wait(A) \cup \{A'\}]$
5: **end if**
6: **if** $unsolved(A.C.m) \not\subseteq rules^i_A(A.C.m)$ **then**
7: **return** FAIL
8: **end if**
9: **for all** `invoke` $A'.C'.m'$ bytecode **do**
10: **if** $A'.C'$ is already loaded **then**
11: **if** $\{d\} \cup rules^i_A(A.C.m) \not\subseteq (rules \cup ruleswait)(A'.C'.m')$ **then**
12: **return** FAIL
13: **end if**
14: **else**
15: $unsolved \leftarrow unsolved[A'.C'.m' \mapsto unsolved(A'.C'.m') \cup \{d\} \cup rules^i_A(A.C.m)]$
16: $wait \leftarrow wait[A \mapsto wait(A) \cup \{A'\}]$
17: **end if**
18: **end for**
19: **return** SUCCESS

**Algorithm 4.3.1:** Verification of the method $A.C.m$ loaded in the domain $d$.

1: $rules^i_A \leftarrow inst_{A,d,\mathcal{S}} \circ rules_A$
2: **for all** class $C$ of application $A$ **do**
3: **for all** method $m$ defined in class $C$ **do**
4: **if** verify_method(A.C.m) = FAIL **then**
5: **return** FAIL
6: **end if**
7: **end for**
8: **end for**
9: $ruleswait \leftarrow ruleswait \cup rules^i_A$
10: **for all** $B$ such that $wait(B) = \emptyset$ and $rules(B) = \emptyset$ **do**
11: /** Note that $A$ is the only possible $B$ at the first iteration **/
12: $rules \leftarrow rules[B \mapsto ruleswait(B)]$
13: **for all** $B'$ such that $B \in wait(B')$ **do**
14: $wait \leftarrow wait[B' \mapsto wait(B') \setminus \{B\}]$
15: **end for**
16: **end for**
17: clean $unsolved$ and $ruleswait$
18: **return** SUCCESS

**Algorithm 4.3.2:** Loading of an application $A$ with policy $(\mathcal{M}_A, \mathcal{L}_A, rules_A)$ in a domain $d$ of $\mathcal{S}$.

**Lemma 4.3.2** *Let $(A_i)_i$ be a list of applications and $\mathcal{S}$ be the system*

$$\mathcal{S} = install((A_n, P_{A_n}), d_n, install((A_{n-1}, P_{A_{n-1}}), d_{n-1}, \dots install((A_0, P_{A_0}), d_0, \mathcal{S}_\emptyset)))$$

*where $\mathcal{S}_\emptyset = (\mathcal{D}, \emptyset, \emptyset, \delta_\emptyset, (\emptyset, \mathcal{D}, rules_\emptyset))$ ($\delta_\emptyset$ and $rules_\emptyset$ are defined only on $\emptyset$) and each application passes the verification to be loaded in its domain. Then $\mathcal{S}$ is secure.*

**Proof.** We prove by induction on the number of methods installed that, for any path from some $m_1$ (we abbreviate full names here) to some $m_2$ in the call graph for $\mathcal{S}'$, $\{\delta(m_1)\} \cup rules(m_1) \subseteq rules(m_2)$ and that for all methods $m_1$ and $m_2$ such that $m_1 \leq m_2$, $rules(m_2) \subseteq rules(m_1)$, which entails the result. This is obviously for $\mathcal{S}_\emptyset$. Let us consider a method $m$ that is added to the system in domain $d$ after passing the verification. And let us consider two methods $m_1$ and $m_2$ such that there is a path $m_1 \dots m_{i-1}.m.m_{i+1} \dots m_2$ in the call graph. By the induction hypothesis, we get $\{\delta(m_1)\} \cup rules(m_1) \subseteq rules(m_{i-1})$ and $\{\delta(m_{i+1})\} \cup rules(m_{i+1}) \subseteq rules(m_2)$.
By definition of the call graph, there is some method $m' \in \mathcal{I}^{direct}(m_{i-1})$ with $m \leq m'$. If $m' = m$, by Rule 2, we get $rules(m_{i-1}) \subseteq rules(m)$. Otherwise, there is a one-arrow path from $m_{i-1}$ to $m'$ so we get $rules(m_{i-1}) \subseteq rules(m')$ by the induction hypothesis. And by Rule 3, we get $rules(m') \subseteq rules(m)$.
We also know from the call graph that there is some method $m'' \in \mathcal{I}^{direct}(m)$ with $m_{i+1} \leq m''$. By Rule 1, we get $rules(m) \subseteq rules(m'')$ and by induction hypothesis we know $rules(m'') \subseteq rules(m_{i+1})$. Summing up those results, we get $\{\delta(m_1)\} \cup rules(m_1) \subseteq rules(m_2)$.
Let us consider any two methods $m_1$ and $m_2$ such that $m_1 \leq m_2$. If both methods are different from $m$, $rules(m_2) \subseteq rules(m_1)$ entails from the induction hypothesis. If $m_1 = m$, we get the result by Rule 3. We cannot have $m_2 = m$ and $m_1 \neq m$ since the Java-loading mechanism ensures that $m_1$ gets loaded after $m$.

□

**Removal.** Note that removing an application does not break the consistency of the properties our algorithms ensure. So no particular additional checking is necessary on a JVM able to unload code.

### 4.3.2 Addition of new domains

Designers of applications may want to change the application by adding new domains, for example to offer services to new partners.

**Definition 4.3.3 (On-device verification of access from new domains)** *Let $M$ be a set of methods of an application $A$ and $D$ a set of domains. Access to the methods $M$ can be granted from the domains $D$ if for each method $A.C.m \in M$:*

$$\forall A'.C'.m' \in \mathcal{I}^{direct}(A.C.m), D \subseteq rules(A'.C'.m') \tag{4.1}$$

$$\forall A'.C', A'.C'.m \leq A.C.m \Rightarrow D \subseteq rules(A'.C'.m) \tag{4.2}$$

Note that (4.1) and (4.2) of Definition 4.3.3 correspond respectively to (1) and (3) of Definition 4.3.1. We do not need to recheck (2) since the set of domains from which calls to $A.C.m$ are granted can only grow.

Once the extended policy is verified, we update the system thus:

$$add((M, D), A, (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, (\mathcal{M}, \mathcal{D}, rules))) = (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, (\mathcal{M}, \mathcal{D}, rules \cup (m \in M \mapsto D))).$$

This extension of the set of domains from which access is granted can be checked for consistency following the Algorithm 4.3.3.

```
1: for all A.C.m of M do
2:   if A.C inherits A'.C' and ∃A'.C'.m and D ⊈ rules(A'.C'.m') then
3:     return FAIL
4:   end if
5:   for all invoke A'.C'.m' bytecode do
6:     if D ⊈ rules(A'.C'.m') then
7:       return FAIL
8:     end if
9:   end for
10: end for
11: rules ← rules ∪ (A.C.m ∈ M ↦ D)
12: return SUCCESS
```

**Algorithm 4.3.3:** Verification of the consistency when granting access to $M$ from $D$.

## 4.4 Implementation on Java-enabled smart cards

The model described in previous sections is abstract enough to be integrated in any loading process of Java-based systems. Since this model has been specially designed to be embedded in very constrained systems, Java-enabled smart cards appear to be appropriate targets for an integration example. We first introduce the technologies considered to deploy Java applications on smart cards, namely Java Card and GlobalPlatform. Then we describe how entities from our model are interpreted in this context, and how control flow policies are encoded and embedded for on-device verification. Finally we discuss about integration of verification algorithms within the virtual machine.

### 4.4.1 Concrete instantiation of the model

The control flow model described in Section 4.2 can be easily instantiated in a context as Java Card&GlobalPlatform. Domains from the model are mapped to GlobalPlatform security domains, applications from the model are mapped to Java Card applets and their related classes, and the shared methods declared by objects that implement the `Shareable` interface from the Java Card API. This mapping is coherent since it satisfies assumptions and hypotheses of the model. In the model it is assumed that admittance in a domain is controlled by the underlying system, which is the case for security domains of GlobalPlatform. It is also assumed in the model that all applications can potentially interact with each other, whatever their domains, and that application interactions are made only through methods of shared objects. This hypothesis is also satisfied by the proposed mapping since it corresponds to the least constrained implementations of GlobalPlatform specifications, *i.e.*, when domains are not isolated from each other. Naturally this hypothesis remains valid on more constrained implementations since it takes into account all the concretely feasible interactions.

The case of the control flow policy instantiation function $inst$, and the related security levels, is trickier since it depends on what the target device is devoted to. Two solutions are conceivable: to implement this function inside or outside the device. When implemented on-device this function requires a predefined set of security levels that can be difficult to maintain after issuance of the card. Implementation outside the device permits to have a more versatile set of security levels but can complicate post-issuance deployment of applications by third-parties. For the purpose of this chapter, we chose the outside implementation because it does not constrain us to define a set of security levels

without a relevant context.

### 4.4.2 Encoding control flow policies

A technical but important point in the concrete instantiation of our model for GlobalPlatform with Java Card is how to encode and attach control flow policies to applications. Security domains of GlobalPlatform are uniquely identified by an AID that is a sequence from five to sixteen bytes. Since an instantiated control flow policy of a method consists of a set of domains authorized to invoke it, it is natural in practice to use a set of AIDs as an instantiated control flow policy. A null AID is simply used to map the *any domain* value ($\top$) from the model. For an application $A$ installed in a domain $d$, it is not mandatory to add $d$ to the set of domains authorized to call shared methods from $A$ because it is automatically added on-device at installation.

Keeping sets of AIDs once applications have been installed on-device is useless: the number of domains and applications installed on one device will always be small because of memory constraints. Indeed, a more efficient binary encoding can be reached to keep policies in memory if the maximal number of domains is fixed to a multiple of eight, and if one index for the whole system is maintained to map a bit position to a domain. A bit set to zero in a policy of a shared method denotes that the corresponding domain is not authorized to call the method, and conversely for a bit set to one. The special value where all bits are set to one is reserved for the “any domain” value of the model. At loading time, each time a policy involves a domain never seen before, a free position is simply assigned to this domain. As a consequence, policies of already loaded shared methods do not have to be updated when a new domain is recorded since the bit corresponding to this domain is already set to zero in their policies.

Thanks to this bit-wise encoding, the memory required to keep control flow policies in the system is greatly reduced since only one or two bytes are needed for each method of installed applications. In addition to minimize overhead memory requirements, such an encoding permits to test very efficiently inclusion between sets of domains in verification algorithms, and consequently to reduce overhead computations.

### 4.4.3 Integration of on-device algorithms

The algorithms described in Section 4.3 have several practical requirements:

1. to have access to the code of an application before it is installed and selectable;
2. to know the control flow policies attached to shared methods of an application before its code is analyzed;
3. to keep persistent data related to control flow policies of already loaded applications across installation of new applications;
4. to prevent an installation or an update if it would break the security policy of already loaded applications.

For requirements 1 and 4 to be met, it is mandatory to integrate the algorithms in the application loading process. This can be done using the techniques described in Section 1.5.1.

Satisfaction of the requirement 2 is intimately linked to the specific development and deployment processes of Java Card. Development of Java Card applets relies on the classical Java compiler to produce standard class files. However, these class files cannot

be loaded directly in a Java Card virtual machine. All class files required to instantiate an applet, except those from the runtime environment, have to be bundled in a special file, the CAP file. The conversion involves many tasks such as bytecode verification (types, operand stack, valid instruction set, etc.), bytecode transformation (typed instructions), instantiation of the static containers, and bytecode reordering for efficient loading. A CAP file is in fact organized to be loaded in card memory in one pass. Our algorithms, especially the one described in Section 4.3.1, are designed to deal with such a constraint, but they need the policy of a method to be known before its bytecode is analyzed. To meet the loading process constraints of CAP files, encoded control flow policies have to be placed before method bytecodes within CAP files. This can be achieved thanks to special containers, called *custom components*, included in Java Card virtual machine specifications. As an alternative, or if custom components are already used for other purposes by the card manufacturer, it is also conceivable to embed encoded control flow policies as traditional values into static components of CAP files. In both cases the CAP file conversion process is amended to include control flow policies specified in parameters thanks to a very simple DSL. As remarked in Section 4.2.3, it is also possible to include automatic propagation of control flow policies of shared methods in this process in order to avoid manual assignment of control flow policies to internal methods (*i.e.*, not part of `Shareable` object) (in)directly called from shared methods; this is simply a union of sets of domains.

To deal with addition of new domain(s) to the policy of a method, we provide APDU commands implemented in a simple applet installed in the controlling authority domain (or the card issuer domain) to load the policy update. We rely on PKI mechanism of security domains to establish a secure channel with the domain that hosts the shared method whose policy has to be updated. If it succeeds, we check whether this update would break or not the control flow policy of loaded applications thanks to the algorithm of Section 4.3.1.

## 4.5 Multi-application use case on smart card

The control flow policy model introduced in Section 4.2 as well as on-device verification algorithms detailed in Section 4.3 have been instantiated on Java-enabled smart cards in Section 4.4. We now illustrate the security benefits from using this model on such devices with a concrete multi-application use case. We first give an overview of the use case with security issues to be tackled. Then we describe how applications from this use case are implemented and then deployed. We finally describe how control flow policies are enforced step by step according to the deployment scenario, and discuss some points related to this process.

### 4.5.1 Overview of the use case

This use case follows the style of previously existing use cases from the literature [37, 22] with addition of GlobalPlatform considerations, support for changes and enhanced deployment scenarios in order to include post-issuance (un)installation of applications.

The use case consists of a Java Card&GlobalPlatform smart card hosting applications from different stakeholders as depicted on Figure 4.1. At the beginning, the card is simply issued with an electronic purse from Bank that includes a service offered to future applications to debit and to credit the purse. In agreement with the bank, the card issuer can install applications from other stakeholders if and only if he guarantees that stakeholders

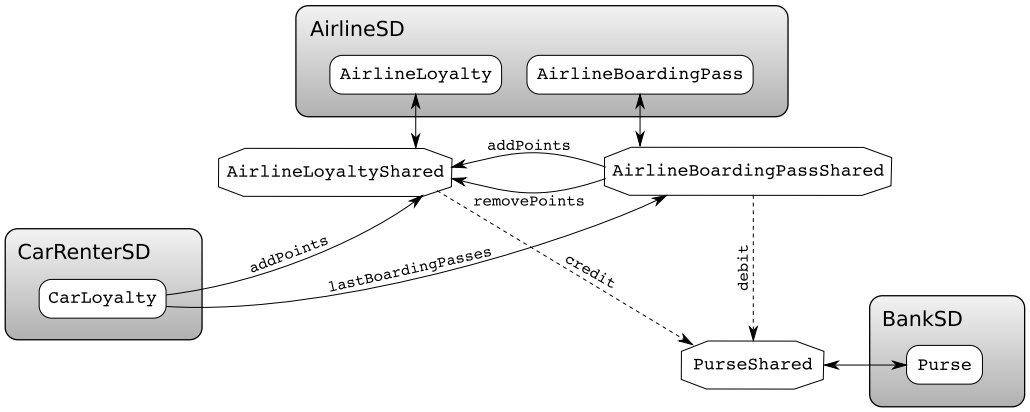


**Figure 4.1:** Overview of application interactions on the use case.

who have paid to use bank services will not relay requests from stakeholders that do not pay for these services.

Later, the card issuer allows two companies, CarRenter and Airline, to install some applications on the card. CarRenter installs a loyalty application, while Airline installs two applications: a loyalty application and an application to manage electronic boarding passes. The electronic boarding pass application interacts with the loyalty application to grant some points when a boarding pass is "consumed", and to debit some points to pay additional charges at check-in, for example in case of excess luggage. The loyalty application of Airline provides a shared service to receive some loyalty points from its partner CarRenter. The loyalty application of CarRenter also checks last recorded boarding passes of Airline to offer discount prices to their customers.

Finally, Airline decides to subscribe to bank services and consequently updates its applications. The purse is now debited by the electronic boarding pass application when the amount of loyalty points is too low to pay additional charges at check-in and to buy duty-free goods according to the current boarding pass. On the contrary, the purse is credited by the loyalty application when the gathered loyalty points exceed some predefined amount. If the installation of this new loyalty application succeeds, then CarRenter can use its own loyalty application to credit the purse via the loyalty application of Airline although it has not payed to use the services supplied by Bank.

### 4.5.2 Implementation and deployment considerations

Each application of the use case is implemented as a Java Card applet. The electronic purse application of Bank is implemented in the class `Purse` that registers a shared instance of `PurseShared` to make its services available to other applications. Thus, the `PurseShared` defines two methods: `debit(int)` and `credit(int)`. The loyalty application of Airline is implemented in the class `AirlineLoyalty` that registers a shared instance of `AirlineLoyaltyShared` with three methods: `getPoints()`, `addPoints(int)` and `removePoints(int)`. The electronic boarding pass application is implemented in the class `AirlineBoardingPass` that registers a shared instance of `AirlineBoardingPassShared` with one method: `lastBoardingPasses()`. The loyalty application of CarRenter is implemented in the class `CarLoyalty` class.

When the card is issued, three domains exist: the BankSD security domain with the `Purse` application installed, the card issuer domain and control authority domain where our algorithms are deployed. Supplementary security domains are created by the card

issuer for each company after issuance of the card to the end-user: AirlineSD domain for Airline company and CarRenterSD domain for CarRenter company. These domains, as well as the BankSD domain, are granted the "delegated" privilege by the card issuer which allows them to freely (un)install their applications in their respective domains. Each company installs its applications in its domain.

### 4.5.3 Enforcement of control flow policies

When the card is issued, only the `Purse` is installed in the BankSD. The control flow policy of the methods declared in `PurseShared` is simply empty since neither Airline nor CarRenter has paid to access its services. One can however imagine that some other domains (not mentioned in this example) corresponding to other entities who have paid are authorized to call them.

The card issuer now creates the two security domains AirlineSD and CarRenterSD, and `CarLoyalty` is installed by CarRenter in CarRenterSD. As `CarLoyalty` uses shared services of `AirlineLoyalty` and `AirlineBoardingPass` that are not yet installed, it is obvious that `CarLoyalty` is not in the *selectable* state and thus cannot be used until all its dependencies are fulfilled. It is important to notice that this behavior, enforced through GlobalPlatform, has nothing to do with functional dependencies but that it is induced by our incremental analysis (Algorithm 4.3.2 in Section 4.3.1).

In this state, Airline can only install its applications that do not use the `Purse`. Airline applications have to allow CarRenterSD to invoke the shared methods `addPoints` of `AirlineLoyaltyShared` and `lastBoardingPasses` of `AirlineBoardingPassShared` otherwise installation of `AirlineLoyalty` and `AirlineBoardingPass` will be rejected and `CarLoyalty` will remain not selectable.

According to the scenario described previously, Airline decides to subscribe to Bank services. Bank consequently updates the control flow policy of the shared methods `debit` and `credit` of `PurseShared` registered by `Purse`. This update is straightforward since no application installed on the card actually uses them and these methods do not invoke methods (directly or indirectly) from other domains. To update its applications, Airline has to first uninstall the ones existing on the card. This operation is permitted by GlobalPlatform if and only if no instance of their shared objects are retained by `CarLoyalty`. This behavior is the expected one and we assume this is the case. After uninstallation of `AirlineLoyalty` and `AirlineBoardingPass`, `CarLoyalty` remains in the selectable state. In fact, this is a convenience to avoid a useless re-analysis of all applications that recursively depend on services of `AirlineBoardingPass` and `AirlineLoyalty` in order to mark them as not selectable. Since control flow policies of selectable applications are satisfied when `AirlineBoardingPass` and `AirlineLoyalty` are installed, they cannot be broken by their uninstallation.

Airline installs its new `AirlineBoardingPass` application that remains not selectable because of unresolved dependencies with `AirlineLoyalty`. However, installation of `AirlineLoyalty` fails because its shared method `addPoints` does not respect the control flow policy attached to the method `credit` of the `PurseShared`. The control flow policy of the method `credit` states that it can be called only from the security domain AirlineSD, but the method `addPoints` invokes the method `credit` and the policy of `addPoints` permits CarRenterSD to call it. It is not possible for Airline to install its `AirlineLoyalty` application even if CarRenterSD is not authorized to call `addPoints` since `CarLoyalty` is already installed.

## 4.6 Conclusion

In this chapter we have presented a control flow policy model for open systems that support dynamic loading of applications. It is appropriate for autonomous systems because everything is computed on-device without any third party. It also fits requirements of small constrained devices, as it has been described for Java-enabled smart cards because verification is performed at loading time in an incremental way.

Future works will offer support for deeper requirements modifications on-device, after issuance, especially modifications of control flow policies for already loaded applications.

# 5. Global Policy

## 5.1 Introduction

In the previous chapters, we have studied the policies of applications in terms of access to their methods. We now add the notion of global policy of a system that defines the control of interactions applied by the system on applications. The global policy of a system defines the sequences of method calls that are forbidden by the system. This type of policy can be used for example to limit the access of applications to the system APIs, or for example for the card issuer to limit access to her own services. This policy is not in conflict with the policies of applications, since policies of applications describe who is authorized to call methods, and the policy of the system adds further restrictions regarding the sequences of calls that are forbidden.

## 5.2 Notations and definitions

We give in this section some additional notations that are used in this chapter, in particular for finite automata and languages.

**Definition 5.2.1 (Finite automaton)** *A* finite automaton *is a tuple* $A = (\Sigma, S, s_0, \zeta, S_F)$ *where:*

- $\Sigma$ *is the input alphabet,*
- $S$ *is the set of states,*
- $s_0 \in S$ *is the initial state,*
- $\zeta : S \times \Sigma \longrightarrow S$ *is the transition function,*
- $S_F \subseteq S$ *is the set of final states.*

**Definition 5.2.2 (Trimmed automaton)** *We say that an automaton* $A = (\Sigma, S, s_0, \zeta, S_F)$ *is* trimmed *if for all state* $s \in S$*, there exist two words* $u$ *and* $v$ *such that* $\zeta(s_0, u) = s$ *and* $\zeta(s, v) \in S_F$*.*

**Definition 5.2.3 (Language)** *Let* $A = (\Sigma, S, s_0, \zeta, S_F)$ *be a finite automaton. The* language *of* $A$*, denoted by* $\mathcal{L}(A)$ *is defined by:*

$$\mathcal{L}(A) = \{a_0 \dots a_n \in \Sigma^* \mid \begin{array}{l} \exists (s_{i_0}, a_0, s_{i_1})(s_{i_1}, a_1, s_{i_2}) \dots (s_{i_n}, a_n, s_{i_{n+1}}) \\ \forall 0 \leq k \leq n, (s_{i_k}, a_k, s_{i_{k+1}}) \in \zeta, s_{i_0} = s_0, s_{i_{n+1}} \in S_F\}. \end{array}$$

*Note that* $\varepsilon$ *denotes the empty word.*

We now define the factors of a language.

**Definition 5.2.4 (Factors)** *Let $L$ be a language of $\Sigma^*$. Then, the* left factors*, (regular)* factors*, and* right factors *of $L$ are respectively defined by:*

$$\begin{array}{lcl} lf(L) & = & \{u \in \Sigma^* \mid \exists w \in \Sigma^*, uw \in L\}, \\ fact(L) & = & \{u \in \Sigma^* \mid \exists v, w \in \Sigma^*, vuw \in L\}, \\ rf(L) & = & \{u \in \Sigma^* \mid \exists v \in \Sigma^*, vu \in L\}. \end{array}$$

**Definition 5.2.5 (Projection)** *Let $\Sigma$ and $\Xi$ be two alphabets. Let $L \subseteq \Sigma^*$ be a language. The* projection onto $\Xi$ *is the alphabetical morphism $\Pi_\Xi$ from $\Sigma$ to $\Xi$ such that for each $x$ of $\Sigma$, $\Pi_\Xi(x) = x$ if $x \in \Xi$ and $\Pi_\Xi(x) = \varepsilon$ otherwise.*

## 5.3 Definition of the global policy

At the verification level, the forbidden sequences of calls are described by a finite automaton.

**Definition 5.3.1 (Global policy)** *Let the* global control flow policy *of a system $\mathcal{G}$ be a finite trimmed automaton $\mathcal{G} = (\Sigma, S, s_0, \zeta, \{s_F\})$ with $\Sigma \subseteq \mathcal{M}$ and such that:*

- *there is no $(s, u) \in S \times \Sigma^*$ with $\zeta(s, u) = s_0$,*
- *there is no $(s, u) \in S \times \Sigma^*$ with $\zeta(s_F, u) = s$.*

**Definition 5.3.2 (Conformity to a policy)** *We say that a system conforms to the policy $\mathcal{G}$ if for each execution trace $t \in \mathcal{M}^*$, $conform(t, \mathcal{G})$ holds with:*

$$conform(t, \mathcal{G}) \Leftrightarrow (\forall v \text{ such that } \Pi_\Sigma(t) = uvu', v \notin \mathcal{L}(\mathcal{G})).$$

Note that we prohibit any transition incoming into the initial state and outgoing of the final state in our global policies. Since a trace is not conforming as soon as it contains a word of the language of the global policy, transitions starting in a final state or leading to the initial state are useless to detect non-conforming traces: on the one hand, the trace was already recognized as invalid the first time $s_F$ was reached and on the other hand we can simply chop off the prefix corresponding to the loop from $s_0$ to $s_0$ while keeping an invalid trace.

Also note that, since the final state has no outgoing transitions, only one final state is needed: if there were various such final states they would be equivalent and could be merged.

## 5.4 Global policy footprint of a method

In this work, we consider open systems that support dynamic application loading. Thus, we aim at a compositional model in which methods can be verified one by one and systems can be extended with new methods without re-verification of already loaded code. In this system, a method $m$ can be valid for a global policy but still contain a part of an invalid trace. If this method is invoked by another one that produces the beginning of a forbidden trace, invokes $m$, and produces the end of a forbidden trace, then $m$ might participate in the construction of an invalid trace even if all the traces of $m$ are allowed. To track this kind of behaviors, we define in this section the contribution of a method to the current execution with respect to the global policy of a system; we call that contribution the *footprint* of the method.

### 5.4.1 Definition of the footprint of a method

**Definition 5.4.1 (Interprocedural control flow graph)** *The interprocedural control flow graph of a set of methods $M$ is the graph $ICFG_M = (V, E)$ such that:*

- $V = \bigcup_{i \in V_m | m \in M, CF_m = (V_m, E_m)} \{m.i\}$,
- $E = \begin{array}{ll} & \{(m.i, m.i') \mid \quad m \in M \wedge CF_m = (V_m, E_m) \wedge (i, i') \in E_m \\ & \qquad \wedge P_m[i] \neq \texttt{invoke}\ m' \text{ with } m' \in M\} \\ \cup & \{(m.i, m''.0) \mid \quad P_m[i] = \texttt{invoke}\ m' \wedge m'' \leq m'\} \\ \cup & \{(m.i, m'.i') \mid \quad P_m[i] = \texttt{return} \wedge P_{m'}[i'-1] = \texttt{invoke}\ m'' \wedge m \leq m''\} \end{array}$

Let $m$ be a method. Let us remind that $\mathcal{I}_m$ is the set of methods called by a method $m$, directly or indirectly. Then the interprocedural control flow graph of $m$ is $ICFG_m = ICFG_{\{m\} \cup \mathcal{I}_m}$.

In general, it is not possible to compute statically the exact set of traces of a system or a subsystem, so we compute an over-approximation of that set. In particular, we put in the interprocedural control flow graph the edges between an `invoke` instruction and all the methods that could be actually invoked, due to method overloading.

As we only consider method calls in the policy, we define a morphism that allows us to restrict the traces to method calls:

$$\begin{array}{rlll} calls: & V & \longrightarrow & \mathcal{M} \\ & m'.i & \longmapsto & m'' \text{ if } P_{m'}[i] = \texttt{invoke}\ m'' \\ & m'.i & \longmapsto & \varepsilon \text{ otherwise} \end{array}$$

We now define the set of traces of a method $m$, that is an over-approximation of its set of execution traces as:

$$traces(m) = \{m.calls(t) \mid t \in paths(ICFG_m, \{m.0\}, \{m.i \mid P_m[i] = \texttt{return}\})\}.$$

We can now define the $\mathcal{G}$-footprint of a method $m$ for a policy $\mathcal{G}$ that describes the factors (left, regular and right) of the traces of the policy language that may result from the execution of this method.

**Definition 5.4.2 ($\mathcal{G}$-footprint of a method)** *The $\mathcal{G}$-footprint of a method $m$ for a policy $\mathcal{G} = (\Sigma, S, s_0, \zeta, s_F)$ is $foot_{\mathcal{G}}(traces(m))$ with*

$$foot_{\mathcal{G}}(L) = \begin{cases} \text{✠} & \text{if } fact(L) \cap \mathcal{L}(\mathcal{G}) \neq \emptyset \\ (\ lf(\Pi_\Sigma(L)) \cap rf(\mathcal{L}(\mathcal{G})), & \\ \quad \Pi_\Sigma(L) \cap fact(\mathcal{L}(\mathcal{G})), & \text{otherwise} \\ \quad rf(\Pi_\Sigma(L)) \cap lf(\mathcal{L}(\mathcal{G}))\ ) & \end{cases}$$

*We denote by $\mathcal{F}_{\mathcal{G}}$ the set of footprints for a policy $\mathcal{G}$.*

This definition allows us to keep information about the contribution of a method $m$ to creation of forbidden sequences as factors. That contribution is written ✠ whenever the method contains an execution trace that is prohibited. Otherwise, it is described as a tuple: the first element of the tuple contains the possible ends of forbidden traces (left factors of complete execution traces of the method); the second element contains the full execution traces of the method that are middle elements of forbidden traces, and the last element of the tuple contains the possible beginnings of forbidden traces. All these factors will be later aggregated with the beginnings and ends of traces of a method that invokes $m$ in order to produce the footprint of this calling method.

### 5.4.2 Properties of the operations on footprints

In this section, we present the main operations on footprints and their properties, that will be useful to obtain compositionality.

**Definition 5.4.3 (Union of $\mathcal{G}$-footprints)** *Let $\mathcal{G} = (\Sigma, S, s_0, \zeta, s_F)$ be a global policy and $f_1$ and $f_2$ be two $\mathcal{G}$-footprints. Then the* union of $f_1$ and $f_2$ *is defined depending on their forms:*

$$\begin{array}{rcl} \maltese \cup \maltese & = & \maltese \\ \maltese \cup (LF_2, F_2, RF_2) & = & \maltese \\ (LF_1, F_1, RF_1) \cup \maltese & = & \maltese \\ (LF_1, F_1, RF_1) \cup (LF_2, F_2, RF_2) & = & (LF_1 \cup LF_2, F_1 \cup F_2, RF_1 \cup RF_2) \end{array}$$

**Lemma 5.4.4 ($foot_{\mathcal{G}}$ distributes over union)** *Let $L_1$ and $L_2$ be two languages, then we have $foot_{\mathcal{G}}(L_1 \cup L_2) = foot_{\mathcal{G}}(L_1) \cup foot_{\mathcal{G}}(L_2)$.*

**Proof.** If one of the languages contains a word with a prohibited factor, its footprint will be $\maltese$, as will be the union of the footprints and the footprint of the union.
Otherwise, the result follows from the fact that the set of factors (resp. left or right) of a language is the set containing all the factors (resp. left or right) of its words. □

**Definition 5.4.5 (Concatenation of $\mathcal{G}$-footprints)** *Let $\mathcal{G} = (\Sigma, S, s_0, \zeta, s_F)$ be a global policy. We define the* concatenation of two $\mathcal{G}$-footprints *depending on their forms.*

$$\begin{array}{rcl} \maltese.\maltese & = & \maltese \\ (LF_1, F_1, RF_1).\maltese & = & \maltese \\ \maltese.(LF_2, F_2, RF_2) & = & \maltese \\ (LF_1, F_1, RF_1).(LF_2, F_2, RF_2) & = & \begin{cases} \maltese & \text{if } RF_1.LF_2 \cap \mathcal{L}(\mathcal{G}) \neq \emptyset \\ (\ LF_1 \cup (F_1.LF_2 \cap rf(\mathcal{L}(\mathcal{G}))), & \\ \quad F_1.F_2 \cap fact(\mathcal{L}(\mathcal{G})), & \\ \quad RF_2 \cup (RF_1.F_2 \cap lf(\mathcal{L}(\mathcal{G}))) \ ) & \\ & \text{otherwise} \end{cases} \end{array}$$

Obviously, the concatenation is not commutative. It has also the following property:

**Lemma 5.4.6** *The concatenation of footprints is associative.*

**Proof.** Let us consider three footprints. If one of them is $\maltese$, the result is immediate. Otherwise, we want to show that

$$((L_1, F_1, R_1).(L_2, F_2, R_2)).(L_3, F_3, R_3) = (L_1, F_1, R_1).((L_2, F_2, R_2).(L_3, F_3, R_3)).$$

Let us give names to the two sides:

$$\begin{array}{rcl} Res_1 & = & ((L_1, F_1, R_1).(L_2, F_2, R_2)).(L_3, F_3, R_3) \\ Res_2 & = & (L_1, F_1, R_1).((L_2, F_2, R_2).(L_3, F_3, R_3)) \end{array}$$

We have $(L_1, F_1, R_1).(L_2, F_2, R_2) = \maltese$ only when $R_1.L_2 \cap \mathcal{L}(\mathcal{G}) \neq \emptyset$. On the other side, if $(L_2, F_2, R_2).(L_3, F_3, R_3) = \maltese$, the results will obviously coincide. Otherwise, the left factor of $(L_2, F_2, R_2).(L_3, F_3, R_3)$ will contain $L_2$ so $(L_1, F_1, R_1).((L_2, F_2, R_2).(L_3, F_3, R_3))$ will be forced to be $\maltese$ too. The converse is similar.

Let us develop further if $(L_1, F_1, R_1).(L_2, F_2, R_2) \neq \maltese$ and $(L_2, F_2, R_2).(L_3, F_3, R_3) \neq \maltese$.

$$\begin{array}{rcl} Res_1 & = & (L_1 \cup (F_1.L_2 \cap rf(\mathcal{L}(\mathcal{G}))), F_1.F_2 \cap fact(\mathcal{L}(\mathcal{G})), R_2 \cup (R_1.F_2 \cap lf(\mathcal{L}(\mathcal{G}))) \\ & & .(L_3, F_3, R_3) \\ Res_2 & = & (L_1, F_1, R_1). \\ & & (L_2 \cup (F_2.L_3 \cap rf(\mathcal{L}(\mathcal{G}))), F_2.F_3 \cap fact(\mathcal{L}(\mathcal{G})), R_3 \cup (R_2.F_3 \cap lf(\mathcal{L}(\mathcal{G})))) \end{array}$$

If $Res_1 = \maltese$ then $(R_2 \cup R_1.F_2).L_3 \cap \mathcal{L}(\mathcal{G}) \neq \emptyset$. Since $(L_2, F_2, R_2).(L_3, F_3, R_3) \neq \maltese$, that would mean $R_1.F_2.L_3 \cap \mathcal{L}(\mathcal{G}) \neq \emptyset$. This would mean that $R_1.(F_2.L_3 \cap rf(\mathcal{L}(\mathcal{G}))) \cap \mathcal{L}(\mathcal{G}) \neq \emptyset$ so $Res_2 = \maltese$. The converse is similar.
Assuming that $Res_1 \neq \maltese$ and $Res_2 \neq \maltese$, we finish developing the two terms.

$$\begin{array}{rcl} Res_1 & = & ((L_1 \cup (F_1.L_2 \cap rf(\mathcal{L}(\mathcal{G})))) \cup ((F_1.F_2 \cap fact(\mathcal{L}(\mathcal{G}))).L_3 \cap rf(\mathcal{L}(\mathcal{G}))), \\ & & (F_1.F_2 \cap fact(\mathcal{L}(\mathcal{G}))).F_3 \cap fact(\mathcal{L}(\mathcal{G})), \\ & & A) \\ Res_2 & = & (L_1 \cup (F_1.(L_2 \cup (F_2.L_3 \cap rf(\mathcal{L}(\mathcal{G})))) \cap rf(\mathcal{L}(\mathcal{G}))), \\ & & F_1.(F_2.F_3 \cap fact(\mathcal{L}(\mathcal{G}))) \cap fact(\mathcal{L}(\mathcal{G})), \\ & & B) \end{array}$$

We first consider factors (second component) of $Res_1$ and $Res_2$: $(F_1.F_2 \cap fact(\mathcal{L}(\mathcal{G}))).F_3 \cap fact(\mathcal{L}(\mathcal{G}))$ and $F_1.(F_2.F_3 \cap fact(\mathcal{L}(\mathcal{G}))) \cap fact(\mathcal{L}(\mathcal{G}))$. In both cases, if $u$ is a word of the language, $u$ is a word of $fact(\mathcal{L}(\mathcal{G}))$ that is composed of three factors: $u = u_1.u_2.u_3$ with $u_i \in F_i$, as $fact(\mathcal{L}(\mathcal{G}))$ is closed by $fact$, we get the equality.

We now consider the left factors (first component) of $Res_1$ and $Res_2$:

$$\begin{array}{rcl} & & (L_1 \cup (F_1.L_2 \cap rf(\mathcal{L}(\mathcal{G})))) \cup ((F_1.F_2 \cap fact(\mathcal{L}(\mathcal{G}))).L_3 \cap rf(\mathcal{L}(\mathcal{G}))) \\ & = & L_1 \cup (F_1.L_2 \cap rf(\mathcal{L}(\mathcal{G}))) \cup ((F_1.F_2 \cap fact(\mathcal{L}(\mathcal{G}))).L_3 \cap rf(\mathcal{L}(\mathcal{G}))) \end{array}$$

and

$$L_1 \cup (F_1.(L_2 \cup (F_2.L_3 \cap rf(\mathcal{L}(\mathcal{G})))) \cap rf(\mathcal{L}(\mathcal{G})))$$

So we have to show that

$$\begin{array}{rcl} & & (F_1.L_2 \cap rf(\mathcal{L}(\mathcal{G}))) \cup ((F_1.F_2 \cap fact(\mathcal{L}(\mathcal{G}))).L_3 \cap rf(\mathcal{L}(\mathcal{G}))) \\ & = & ((F_1.L_2) \cup ((F_1.F_2 \cap fact(\mathcal{L}(\mathcal{G}))).L_3)) \cap rf(\mathcal{L}(\mathcal{G})) \end{array}$$

and

$$\begin{array}{rcl} & & F_1.(L_2 \cup (F_2.L_3 \cap rf(\mathcal{L}(\mathcal{G})))) \cap rf(\mathcal{L}(\mathcal{G})) \\ & = & (F_1.L_2 \cup F_1.(F_2.L_3 \cap rf(\mathcal{L}(\mathcal{G})))) \cap rf(\mathcal{L}(\mathcal{G})) \end{array}$$

are equal. That can be reduced to show that

$$((F_1.F_2 \cap fact(\mathcal{L}(\mathcal{G}))).L_3) \cap rf(\mathcal{L}(\mathcal{G}))$$

and

$$(F_1.(F_2.L_3 \cap rf(\mathcal{L}(\mathcal{G})))) \cap rf(\mathcal{L}(\mathcal{G}))$$

are equal. In both cases if $u$ is a word of the language, it is composed of three factors: $u = u_1.u_2.u_3$ with $u_1 \in F_1$, $u_2 \in F_2$, and $u_3 \in L_3$. As $u$ belongs to $rf(\mathcal{L}(\mathcal{G}))$, obviously $u_2.u_3 \in rf(\mathcal{L}(\mathcal{G}))$, and $u_1u_2 \in fact(\mathcal{L}(\mathcal{G}))$.

□

We will now show that $foot_{\mathcal{G}}$ is an endomorphism, *i.e.*, it somehow "distributes" over concatenation.

**Lemma 5.4.7 (Footprint is a morphism)** *Let $L_1$ and $L_2$ be two languages, then we have $foot_{\mathcal{G}}(L_1.L_2) = foot_{\mathcal{G}}(L_1).foot_{\mathcal{G}}(L_2)$.*

**Proof.** If $foot_{\mathcal{G}}(L_1) = \maltese$ or $foot_{\mathcal{G}}(L_2) = \maltese$, then $foot_{\mathcal{G}}(L_1.L_2) = \maltese$ because $fact(L_i) \subseteq fact(L_1.L_2)$.
Otherwise, let $foot_{\mathcal{G}}(L_1) = (LF_1, F_1, RF_1)$ and $foot_{\mathcal{G}}(L_2) = (LF_2, F_2, RF_2)$. If we have $foot_{\mathcal{G}}(L_1.L_2) = \maltese$, we know that there is a factor of $L_1.L_2$ in $\mathcal{L}(\mathcal{G})$. That factor must be of the form $u_1.u_2$, with $u_1 \in rf(\Pi_\Sigma(L_1))$, $u_2 \in lf(\Pi_\Sigma(L_2))$. Obviously, $u_1 \in lf(\mathcal{L}(\mathcal{G}))$ and $u_2 \in rf(\mathcal{L}(\mathcal{G}))$. So $u_1 \in RF_1$ and $u_2 \in LF_2$ which implies that $RF_1.LF_2 \cap \mathcal{L}(\mathcal{G}) = \emptyset$ so $foot_{\mathcal{G}}(L_1).foot_{\mathcal{G}}(L_2) = \maltese$.
If $foot_{\mathcal{G}}(L_1.L_2) \neq \maltese$, we have:

$$\begin{array}{rl} foot_{\mathcal{G}}(L_1.L_2) = & (lf(\Pi_\Sigma(L_1.L_2)) \cap rf(\mathcal{L}(\mathcal{G})), \\ & \Pi_\Sigma(L_1.L_2) \cap fact(\mathcal{L}(\mathcal{G})), \\ & rf(\Pi_\Sigma(L_1.L_2)) \cap lf(\mathcal{L}(\mathcal{G}))) \end{array}$$

Let us first consider the factors :

$$\begin{array}{rl} & \Pi_\Sigma(L_1.L_2) \cap fact(\mathcal{L}(\mathcal{G})) \\ = & (\Pi_\Sigma(L_1).\Pi_\Sigma(L_2)) \cap fact(\mathcal{L}(\mathcal{G})) \\ = & ((\Pi_\Sigma(L_1) \cap fact(\mathcal{L}(\mathcal{G}))).(\Pi_\Sigma(L_2) \cap fact(\mathcal{L}(\mathcal{G})))) \cap fact(\mathcal{L}(\mathcal{G})) \end{array}$$

For left factors we have:

$$\begin{array}{rl} & lf(\Pi_\Sigma(L_1.L_2)) \cap rf(\mathcal{L}(\mathcal{G})) \\ = & lf(\Pi_\Sigma(L_1).\Pi_\Sigma(L_2)) \cap rf(\mathcal{L}(\mathcal{G})) \\ = & (lf(\Pi_\Sigma(L_1)) \cup \Pi_\Sigma(L_1).lf(\Pi_\Sigma(L_2))) \cap rf(\mathcal{L}(\mathcal{G})) \\ = & (lf(\Pi_\Sigma(L_1)) \ \cap \ rf(\mathcal{L}(\mathcal{G}))) \ \cup \ (\Pi_\Sigma(L_1).lf(\Pi_\Sigma(L_2)) \ \cap \\ & rf(\mathcal{L}(\mathcal{G}))) \\ = & LF_1 \cup (\Pi_\Sigma(L_1).lf(\Pi_\Sigma(L_2)) \cap rf(\mathcal{L}(\mathcal{G}))) \end{array}$$

As all the words of $rf(\mathcal{L}(\mathcal{G})))$ are concatenations of factors of $\mathcal{L}(\mathcal{G}))$

$$\begin{array}{rl} = & LF_1 \cup ((\Pi_\Sigma(L_1) \cap fact(\mathcal{L}(\mathcal{G}))).lf(\Pi_\Sigma(L_2)) \cap rf(\mathcal{L}(\mathcal{G}))) \\ = & LF_1 \cup (F_1.lf(\Pi_\Sigma(L_2)) \cap rf(\mathcal{L}(\mathcal{G}))) \end{array}$$

Obviously, we have:

$$\begin{array}{rl} = & LF_1 \cup (F_1.(lf(\Pi_\Sigma(L_2)) \cap rf(\mathcal{L}(\mathcal{G}))) \cap rf(\mathcal{L}(\mathcal{G}))) \\ = & LF_1 \cup (F_1.LF_2 \cap rf(\mathcal{L}(\mathcal{G}))) \end{array}$$

and we can obtain the right factors in the same way. □

From Lemma 5.4.4 and Lemma 5.4.7, we get the following proposition:

**Proposition 5.4.8** *For a language $L \subseteq \mathcal{M}^*$ and a global policy of the system $\mathcal{G} = (\Sigma, S, s_0, \zeta, s_F)$, we have*

$$foot_{\mathcal{G}}(L) = \bigcup_{\substack{m_0 m_1 \dots m_n \in L \mid \\ \forall 0 \leq i \leq n, m_i \in \mathcal{M}}} foot_{\mathcal{G}}(\{m_0\}).foot_{\mathcal{G}}(\{m_1\}) \dots foot_{\mathcal{G}}(\{m_n\}).$$

### 5.4.3 Compositionality of the footprint computation

The $\mathcal{G}$-footprints of methods can be computed in compositional way except in the presence of mutual recursive methods that have to be analyzed together. For a method $m$, we compute the strongly connected components of the graph $CG_m$. Starting from $m$, we have a partial order of components as the transitive closure of the relation saying that a component $c_1$ is lower than a component $c_2$ if there exists an edge from $c_1$ to $c_2$. Then, the methods of each strongly connected component have to be analyzed together, when all the greater components have been analyzed (*i.e.* when their footprints are available).

**Proposition 5.4.9 (Compositionality)** *Let us consider a method $m$ and a global policy of the system $\mathcal{G} = (\Sigma, S, s_0, \zeta, s_F)$, and $M$ the set of methods of the strongly connected component of $CG_m$ that contains $m$. Then,*

$$foot_{\mathcal{G}}(traces(m)) = compose_{\mathcal{G},M}(calls(paths(ICFG_M, m.0, \{m.i \mid P_m[i] = \texttt{return}\})))$$

*with the morphism*

$$\begin{array}{rccll} compose_{\mathcal{G},M} : & \mathcal{M} & \longrightarrow & \mathcal{F}_{\mathcal{G}} & \\ & m' & \longmapsto & foot_{\mathcal{G}}(traces(m')) \textit{ if} & m' \notin M \\ & m' & \longmapsto & foot_{\mathcal{G}}(\{m'\}) \textit{ if} & m' \in M \end{array}$$

**Proof.** Straightforward from Proposition 5.4.8. □

## 5.5 Implementation of the footprint computation

Our targets are small embedded systems. Thus, we need to provide a compact representation of footprints that uses as little memory as possible and that is easy to manipulate. For this purpose, we use sets of pairs of states of the automaton to represent footprints. Using this representation, we only need one set since left factors obviously end at $s_F$ and right factors obviously start at $s_0$.

**Definition 5.5.1 ($\mathcal{G}$-footprint implementation)** *Let $\mathcal{G}$ be a global policy of the system. The* $\mathcal{G}$-footprint implementation *is given by the function:*

$$\begin{array}{rrcl} \Phi \quad : & \mathcal{F}_{\mathcal{G}} & \longrightarrow & \wp(S \times S) \\ & \maltese & \longmapsto & \{(s_i, s_j) \mid \forall s_i, s_j \in S\} \\ & (LF, F, RF) & \longmapsto & \{(s_i, s_j) \mid \exists u \in F, \zeta(s_i, u) = s_j\} \\ & & & \cup \ \{(s_0, s_i) \mid \exists u \in RF, \zeta(s_0, u) = s_i\} \\ & & & \cup \ \{(s_i, s_F) \mid \exists u \in LF, \zeta(s_i, u) = s_F\} \end{array}$$

*We write $F_m$ for the implementation of the $\mathcal{G}$-footprint of a method $m$, namely $F_m = \Phi(foot_{\mathcal{G}}(traces(m)))$.*

We now define an internal operation of composition of footprint implementations.

**Definition 5.5.2 (Composition)** *The composition over $\wp(S \times S)$ is defined as, for $S_1$ and $S_2$:*

$$\begin{array}{rcl} S_1 \oplus S_2 & = & \{(s_i, s_F) \mid (s_i, s_F) \in S_1\} \\ & \cup & \{(s_0, s_i) \mid (s_0, s_i) \in S_2\} \\ & \cup & \{(s_i, s_j) \mid \exists k, (s_i, s_k) \in S_1, (s_k, s_j) \in S_2 \\ & & \quad \textit{or } \exists k, (s_0, s_k) \in S_1, (s_k, s_F) \in S_2 \\ & & \quad \textit{or } (s_0, s_F) \in S_1 \cup S_2\} \end{array}$$

Note that the definition forces the composition to be the full set $S \times S$ as soon as it contains $(s_0, s_F)$. It is easy to see that $\{(s_i, s_j) \mid \forall s_i, s_j \in S\}$ is absorbing for $\oplus$: this corresponds to the element ✠ for normal footprints. So we will also write ✠ for the full set $S \times S$.

**Lemma 5.5.3** *Let $S_1$, $S_2$, $S_3$ and $S_4$ be elements of $\wp(S \times S)$. The composition is monotonic: if $S_1 \subseteq S_2$ and $S_3 \subseteq S_4$ then $S_1 \oplus S_3 \subseteq S_2 \oplus S_4$.*

The proof of this lemma is straightforward.

**Lemma 5.5.4** *$\Phi$ is a morphism* i.e.*, $\Phi(f_1.f_2) = \Phi(f_1) \oplus \Phi(f_2)$.*

**Proof.** If $f_i = ✠$, then $f_1.f_2 = ✠$ and $\Phi(f_i) = \Phi(f_1.f_2) = ✠$ which will absorb $\Phi(f_{3-i})$.
Let us then assume that $f_1 \neq ✠$ and $f_2 \neq ✠$. We write $f_1 = (LF_1, F_1, RF_1)$ and $f_2 = (LF_2, F_2, RF_2)$.
Let us first prove that $\Phi(f_1.f_2) \subseteq \Phi(f_1) \oplus \Phi(f_2)$. If $f_1.f_2 = ✠$, this means that $RF_1.LF_2 \cap \mathcal{L}(\mathcal{G}) \neq \emptyset$. So we have some $u = u_1.u_2$ such that $u_1 \in RF_1$ and $u_2 \in LF_2$ and $u \in \mathcal{L}(\mathcal{G})$, which means that $\zeta(s_0, u) = \zeta(\zeta(s_0, u_1), u_2) = s_F$. From this, we conclude that $(s_0, \zeta(s_0, u_1)) \in \Phi(f_1)$ and $(\zeta(s_0, u_1), s_F) \in \Phi(f_2)$ and so that $\Phi(f_1) \oplus \Phi(f_2) = ✠$ since it contains $(s_0, s_F)$.
Let us now assume that $f_1.f_2 \neq ✠$ and write $f_1.f_2 = (LF, F, RF)$.
We consider each of the three sub-sets composing $\Phi(f_i)$ separately.
We have $F = F_1.F_2 \cap fact(\mathcal{L}(\mathcal{G}))$. Let us consider some $u \in F$. We know that there exist $u_1$ and $u_2$ such that $u = u_1.u_2$, with $u_1 \in F_1$ and $u_2 \in F_2$. If we consider a pair of states $(s_i, s_j)$ such that $\zeta(s_i, u) = s_j$, we know that $\zeta(s_i, u_1)$ is some state $s_k$ and that $(s_i, s_k)$ must be in $\Phi(f_1)$ and that $(s_k, s_j)$ must be in $\Phi(f_2)$ since $\zeta(s_k, u_2) = s_j$, by definition of $\Phi$. So $(s_i, s_j) \in \Phi(f_1) \oplus \Phi(f_2)$.
We have $LF = LF_1 \cup (F_1.LF_2 \cap rf(\mathcal{L}(\mathcal{G})))$. Let us consider some $u \in LF$ and $(s_i, s_F)$ a pair of states such that $\zeta(s_i, u) = s_F$. If $u \in LF_1$, then $(s_i, s_F)$ must be in $\Phi(f_1)$ by definition of $\Phi$; and so it is preserved by $\oplus$. Otherwise, we must have $u \in F_1.LF_2$ so $u = u_1.u_2$ with $u_1 \in F_1$ and $u_2 \in LF_2$. So $\zeta(s_i, u_1)$ must be some state $s_j$ with $\zeta(s_j, u_2) = s_F$. By definition of $\Phi$, we have $(s_i, s_j) \in \Phi(f_1)$ and $(s_j, s_F) \in \Phi(f_2)$ which entails that $(s_i, s_F)$ is in $\Phi(f_1) \oplus \Phi(f_2)$.
By a similar argument we prove that the image of $RF$ is also included in $\Phi(f_1) \oplus \Phi(f_2)$.
Conversely, let us prove that $\Phi(f_1) \oplus \Phi(f_2) \subseteq \Phi(f_1.f_2)$. Let us consider for this $(s_i, s_j)$ in $\Phi(f_1) \oplus \Phi(f_2)$. We reason over the three definition cases of $\oplus$.

1. $s_j = s_F$ and $(s_i, s_F) \in \Phi(f_1)$; then there exists some $u$ such that $\zeta(s_i, u) = s_F$ by definition of $\Phi$ and so $u$ must be in $RF_1 \cup F_1 \cup LF_1$. In all three sub-cases, since it is a right factor of $\mathcal{L}(\mathcal{G})$ (because it can end in $s_F$) it must also be in $LF_1$. This entails that it is in $LF$ by definition of concatenation so $(s_i, s_F)$ is in $\Phi(f_1.f_2)$.

2. $s_i = s_0$ and $(s_0, s_j) \in \Phi(f_2)$. This case is similar to the previous one.

3. We have again three sub-cases.

   - There exists some $s_k$ such that $(s_i, s_k) \in \Phi(f_1)$ and $(s_k, s_j) \in \Phi(f_2)$. So we can find $u_1 \in RF_1 \cup F_1 \cup LF_1$ such that $\zeta(s_i, u_1) = s_k$ and $u_2 \in RF_2 \cup F_2 \cup LF_2$ such that $\zeta(s_k, u_2) = s_j$.
     - If $u_1 \in LF_1$, then $s_j = s_k = s_F$ and $u_2$ must be $\varepsilon$ since there is no transition out of $s_F$. Then $u = u_1$ is in $LF$ and $(s_i, s_F)$ is in $\Phi(f_1.f_2)$.

- If $u_2 \in RF_2$, then $s_i = s_k = s_0$ and $u_1$ must be $\varepsilon$ since there is no transition to $s_0$. Then $u = u_2$ is in $RF$ and $(s_0, s_j)$ is in $\Phi(f_1.f_2)$.
- If $u_1 \in F_1$ and $u_2 \in LF_2$, then $s_j = s_F$ and we just need to show that $u_1.u_2$ is in $rf(\mathcal{L}(\mathcal{G}))$ to prove that $u_1.u_2$ is in $LF$. Since $\mathcal{G}$ is trimmed, there must exist some word $v$ such that $\zeta(s_0, v) = s_i$ so $u_1.u_2$ is indeed a right factor of $\mathcal{L}(\mathcal{G})$, which proves that $(s_i, s_F)$ is in $\Phi(f_1.f_2)$.
- If $u_1 \in F_1$ and $u_2 \in F_2$, since the automaton is trimmed, $u_1.u_2$ is a factor of $\mathcal{L}(\mathcal{G})$ and is in the factors of $f_1.f_2$ so $(s_i, s_j)$ is in $\Phi(f_1.f_2)$.
- If $u_1 \in RF_1$ and $u_2 \in LF_2$ then $u_1.u_2 \in \mathcal{L}(\mathcal{G})$ so $RF_1.LF_2 \cap \mathcal{L}(\mathcal{G}) \neq \emptyset$ so $f_1.f_2 = \maltese$ in which case $(s_i, s_j) \in \Phi(f_1.f_2) = \maltese$.
- If $u_1 \in RF_1$ and $u_2 \in F_2$, then $s_i = s_0$ and there must exist some word $v$ such that $\zeta(\zeta(s_0, u_1.u_2), v) = s_F$ since $\mathcal{G}$ is trimmed. So $u = u_1.u_2$ is a left factor of $\mathcal{L}(\mathcal{G})$, so $u$ must be in the right factors of $f_1.f_2$ which entails that $(s_0, sj) \in \Phi(f_1.f_2)$.

- If $(s_i, s_j)$ is present because there is some $k$ such that $(s_0, s_k) \in \Phi(f_1)$ and $(s_k, s_F) \in \Phi(f_2)$ this means that there is some word $u_1$ such that $\zeta(s_0, u_1) = s_k$ with $u_1$ in the right factors of $f_1$ (because the factors of $f_1$ that are in $lf(\mathcal{L}(\mathcal{G}))$ are also in the right factors; and if $u_1$ is in the left factors, then $s_k = s_F$ so it is also in the right factors) and some $u_2$ such that $\zeta(s_k, u_2) = s_F$ with $u_2$ in the left factors of $f_2$, so $f_1.f_2 = \maltese$ and $(s_i, s_j) \in \Phi(f_1.f_2)$.
- Lastly, if $(s_0, s_F)$ is in $\Phi(f_1)$, then we must have $f_1 = \maltese$ by definition of footprints, since it means that a factor of the underlying language of traces is prohibited. Then $f_1.f_2 = \maltese$ and $\Phi(f_1.f_2) = \maltese$ so $(s_i, s_j) \in \Phi(f_1.f_2)$.

□

$\Phi$ also distributes over $\cup$.

**Lemma 5.5.5** *For any $f_1$ and $f_2$ two footprints, $\Phi(f_1 \cup f_2) = \Phi(f_1) \cup \Phi(f_2)$.*

**Proof.** Let us consider $f_1$ and $f_2$ two footprints. If $f_1 = \maltese$, $\Phi(f_1 \cup f_2) = \Phi(\maltese) = \maltese = \maltese \cup \Phi(f_2)$. The result is identical if $f_2 = \maltese$. If neither of them is $\maltese$, we write $f_1 = (LF_1, F_1, RF_1)$ and $f_2 = (LF_2, F_2, RF_2)$. $f_1 \cup f_2 = (LF_1 \cup LF_2, F_1 \cup F_2, RF_1 \cup RF_2)$ and the definition of $\Phi$ allows us to conclude. □

In the rest of this section, we define a system of equations that allows us to compute the footprint implementation of a method that corresponds to the $\mathcal{G}$-footprint of a method $m$.

For each method $m$ we consider $M$ the set of methods of the strongly connected component of $CG_m$ that contains $m$. We define the system of equations $(S_M)$ where the rules $instr$ are given by Figure 5.1:

$$S_{m''.i} = \bigcup_{(m'.j, m''.i) \in ICFG_M} instr_{P_{m'}[j]}(S_{m'.j})$$

with the initial state

$$S_{m.0} \supseteq \begin{cases} \{(s_i, s_i) \mid s_i \in S\} \cup \{(s_i, s_j) \mid s_i, s_j \in S \text{ if } s_0 = s_F\} & \text{when } m \notin \Sigma \\ \{(s_i, s_j) \mid s_i, s_j \in S, \zeta(s_i, m) = s_j \text{ or } \zeta(s_0, m) = s_F\}\} & \text{when } m \in \Sigma \end{cases}$$

Note that the last rule of Figure 5.1 takes the union of the footprint implementations of all the actual methods that could be invoked by the instruction.

**Proposition 5.5.6** *The system of equations $(S_M)$ admits a least solution.*

$$\frac{b \neq \texttt{invoke}}{instr_b(S) = S}$$

$$\frac{b = \texttt{invoke}\ \ m' \text{ and } m' \notin \Sigma \text{ and } m' \in M}{instr_b(S) = S}$$

$$\frac{b = \texttt{invoke}\ \ m' \text{ and } m' \in \Sigma \text{ and } m' \in M}{instr_b(S) = S \oplus \Phi(foot_{\mathcal{G}}(\{m'\}))}$$

$$\frac{b = \texttt{invoke}\ \ m' \text{ and } m' \notin M}{instr_b(S) = S \oplus \bigcup_{m'' \leq m'} F_{m''}}$$

**Figure 5.1:** Transfer function $instr_b(S)$.

**Proof.** The set $(\wp(S \times S), \subseteq, \cup, \cap)$ is a finite lattice. Lemma 5.5.3 implies that the transfer functions $instr_b$ are monotonic with respect to $\subseteq$. Thus we can apply Knaster-Tarski theorem. □

And finally we prove that this least fixpoint construction indeed implements footprints.

**Proposition 5.5.7** *Let $m$ be a method, $M$ the set of methods of the strongly connected component of $CG_m$ that contains $m$. Then*

$$F_m = \bigcup_{i|P_m[i]=\texttt{return}} S_i$$

*with $S_i$ the least solutions of the equations of $(S_M)$.*

Note that this property justifies the last rule of Figure 5.1: the computation of the solution of $(S_M)$ can be based on the footprint implementations of all the methods that are not in the same strongly connected component of the call graph since that will result in the same value. And this proves that the system of equations indeed provides a way to compute the footprint implementations.

**Proof.** Let us write

$$U = \bigcup_{i|P_m[i]=\texttt{return}} S_i$$

and let us first prove that $F_m \subseteq U$.

$$\begin{array}{rcl} F_m & = & \Phi(foot_{\mathcal{G}}(traces(m))) \\ & = & \Phi(\bigcup_{t \in traces(m)} foot_{\mathcal{G}}(\{t\})) \\ & = & \Phi(\bigcup_{m_1 \ldots m_n \in traces(m)} foot_{\mathcal{G}}(\{m_1\}) \ldots foot_{\mathcal{G}}(\{m_n\})) \\ & = & \bigcup_{m_1 \ldots m_n \in traces(m)} \Phi(foot_{\mathcal{G}}(\{m_1\})) \oplus \cdots \oplus \Phi(foot_{\mathcal{G}}(\{m_n\})) \end{array}$$

Let us consider any pair in $F_m$ and some trace $t$ of $m$ so that the pair is in $\Phi(foot_{\mathcal{G}}(\{t\}))$. Since traces come from actual execution paths, we consider one path $p$ in $CG_m$ such that the path produces the trace $t$. Let us sum up the constraints on $(S_M)$ we obtain by looking at that path $p$. $p$ can be written as $(m_{j_0}.i_{j_0}) \ldots (m_{j_q}.i_{j_q})$ where at each step $j_k$ the instruction $i_{j_k}$ of method $m_{j_k}$ is executed.

Let us reason on the length of $p$ to show that, if we consider a prefix path $p'$ of $p$ of any length, the set $S_{m'.i'}$ must contain $\Phi(foot_{\mathcal{G}}(t'))$ where $t'$ is the part of $t$ corresponding to $p'$.

If $m \in \Sigma$, every trace of $m$ begins with $m$. In that case we also know that $\Phi(foot_{\mathcal{G}}(\{m\})) \subseteq S_{m.0}$ by definition of the initial state of the system $(S_M)$. Otherwise, $S_{m.0}$ initially contains simply $\Phi(foot_{\mathcal{G}}(\{\varepsilon\}))$.
Let us add one instruction to $p'$. For all instruction $P_{m_{j_k}}[i_{j_k}]$ along $p$ that is not an `invoke`, we simply learn that $S_{m_{j_k}.i_{j_k}} \supseteq S_{m_{j_k-1}.i_{j_k-1}}$ by the rules of Figure 5.1. Correspondingly, $t'$ is not extended by a non-invoke instruction.
If the added instruction is $P_{m_{j_k}}[i_{j_k}] = \texttt{invoke}\ m'$, we have to consider the various possibilities for $m'$:

- if $m' \notin \Sigma$ and $m' \in M$, $t'$ is left unmodified and $S_{m_{j_k}.i_{j_k}} \supseteq S_{m_{j_k-1}.i_{j_k-1}}$ ensures the result,
- if $m' \in \Sigma$ and $m' \in M$, $m'$ is appended to $t'$ and we know that we have $S_{m_{j_k}.i_{j_k}} \supseteq S_{m_{j_k-1}.i_{j_k-1}} \oplus \Phi(foot_{\mathcal{G}}(\{m'\}))$; by Lemma 5.5.3, $\Phi(foot_{\mathcal{G}}(\{t'\})) \subseteq S_{m_{j_k-1}.i_{j_k-1}}$ entails that $\Phi(foot_{\mathcal{G}}(\{t'\})) \oplus \Phi(foot_{\mathcal{G}}(\{m'\})) \subseteq S_{m_{j_k}.i_{j_k}}$,
- if $m' \notin M$, the fragment $t''$ of the trace that corresponds to the call of $m'$ is such that $\Phi(foot_{\mathcal{G}}(\{t''\})) \subseteq \bigcup_{m'' \leq m'} F_{m''}$ since $t''$ is a trace of one such method $m''$; so, by Lemma 5.5.3 we get $\Phi(foot_{\mathcal{G}}(\{t'\})) \oplus \Phi(foot_{\mathcal{G}}(\{t''\})) \subseteq S_{m_{j_k-1}.i_{j_k-1}} \oplus \bigcup_{m'' \leq m'} F_{m''} \subseteq S_{m_{j_k}.i_{j_k}}$.

From this we easily conclude that $\Phi(foot_{\mathcal{G}}(\{t\})) \subseteq U$ and consequently that $F_m \subseteq U$.
Let us prove the converse, $U \subseteq F_m$, by considering the set of definitions $(R_M)$:

$$R_{m'.i} = \bigcup_{p \in paths(ICFG_M, \{m.0\}, \{m'.i\})} \Phi(foot_{\mathcal{G}}(\{m.calls(p)\}))$$

with the fact that $F_m = \bigcup_{i|P_m[i]=\texttt{return}} R_{m.i}$. We can show that, for any edge $(m'.j', m''.j'')$ in $ICFG_M$, $R_{m''.j''} \supseteq instr_{P_{m'}[j']}(R_{m'.j'})$ by cases over the definition of $instr_b$ in a similar way to previously shown. Since $U$ is the least solution of $(S_M)$, we directly get that $S_{m'.i'} \subseteq R_{m'.i'}$ for all $m'.i' \in ICFG_M$. Therefore

$$U = \bigcup_{i|P_m[i]=\texttt{return}} S_{m.i} \subseteq \bigcup_{i|P_m[i]=\texttt{return}} R_{m.i} = F_m$$

which concludes the proof. □

This proof uses some property of interest to embed the verification. As we noticed just above, as soon as we have a set of $R_{m'.i'}$ such that, for any edge $(m'.j', m''.j'')$ in $ICFG_M$, $R_{m''.j''} \supseteq instr_{P_{m'}[j']}(R_{m'.j'})$, then the union of footprints at all `return` instructions must contain $F_m$, since it is the least such set. This means that checking for those inclusions will provide a low-complexity technique to ensure that a declared footprint for a method $m$ is safe *i.e.*, is an over-approximation of $F_m$. This also means that the footprint implementation against which some method bytecode will be verified can be any such over-approximation: to handle method overloading we will only use the union of all the footprints of methods in one class and its subclasses.

## 5.6 On-device verification

### 5.6.1 Lightweight verification

The computation of footprints uses a fix-point computation thus, it is too heavy for the computation capability of a smart card. So we use for this kind of policies the “Lightweight

verification" as mentioned in Section 1.5.2. The original Lightweight Bytecode Verification has to be extended to manage the footprints as it was originally dedicated to type verification. Anyway, as we have encoded the footprints with a structure that is a semi-lattice, the verification itself is still the same. The verification algorithm is then linear in bytecode number and analyzes bytecode in stream which is the loading model of Java Card. Each class will be loaded with the footprints of its methods, and a repository of footprints of already loaded methods will be managed on-device.

### 5.6.2 Encoding of the embedded proof

Each class file is analyzed off-board, either alone or with the methods of its strongly connected component when needed. Then, meta-data have to be shipped with the code. In traditional Java, they are added to the class file in the form of class file attributes. For GlobalPlatform with Java Card, we use the same architecture as the one presented in the previous chapter in section 4.4. We need:

- for each method $m$ of the class:
  - the over-approximated footprint $F_m$ of $m$, so that every method $m'$ such that $m' \leq m$ has a footprint included in $F_m$,
  - "proof annotations" that is the list of intermediate footprints $S_i$ computed externally for all the $i$ such that $P_m[i]$ is the target of a jump,
- for each method $m$ that is invoked by methods $m_0, \ldots, m_n$ of the class :
  - "believed footprint" that is the footprint $F_m$ of $m$ that has been used in composition to compute the footprints of the methods $m_0, \ldots, m_n$.

Let us consider a system with a global policy $\mathcal{G} = (\Sigma, S, s_0, \zeta, s_F)$. Footprints are encoded in a binary form, if $S$ contains $n$ elements, then we need $n \times (n-1)$ bits for a footprint and the bit $0 \leq i < n \times (n-1)$ encodes the presence of $(s_{i/n}, s_{i \bmod n})$ in the footprint of $m$. For readability of the rest of this section, we use the array notation and denotes by $F_m[i]$ the $(i+1)$-th bit of $F_m$.

### 5.6.3 On-device meta-data

The on-device system needs to keep data to manage the policy. We define two repositories, $R$ which maps the verified methods to their footprint and $R_{tmp}$ the temporary repository which maps methods that are not loaded to their believed footprint.

The global policy $\mathcal{G} = (\Sigma, S, s_0, \zeta, s_F)$ is an automaton on a subset of the methods, thus we need to record $\Sigma$ and $\zeta$ providing that we encode the initial state as number $0$ and the final state as number $n-1$ if the automaton has $n$ states. To take $\zeta$ into account and record it on the device, we just include it in the initial state of the on-device verification. The system starts in a state where:

- $R$ is empty: we write $R[m] = \bot$ for all method $m$,
- $R_{tmp}$ contains for each $m$ of $\Sigma$ a believed footprint such that for each $\zeta(s_i, m) = s_j$, the $R_{tmp}[m][i \times n + j]$.

### 5.6.4 Verification algorithm

The verification algorithm is given by the Algorithm 5.6.1. For a policy of $n$ states, a footprint is a vector of $n \times (n-1)$ bits. Then the basic operations are encoded in the following way:

- $F_{init}$ is the vector in which each $F[i * n + i]$ is equal to $0$,
- nonvalid($F_m$) is ($F_m \& mask == mask$) with $mask$ the vector that encodes $\{(s_0, s_n)\}$ *i.e.* all the bits equal to $0$ except the bit $n-1$,
- $F_1 \subseteq F_2$ is $F_1 \mid F_2 == F_2$.

From lines 1 to 7, we check the believed footprints: if another footprint for the same method already exists on the device, the shipped one is just dropped, otherwise it is added to the temporary repository. Then, for all footprints of the methods, we check from line 8 to 11 if the footprint is valid, if it conforms to the believed ones of the applications already loaded on the system, and if it is compliant with the class hierarchy until line 15. Then, the last part of the algorithm verifies the proof of the footprint of the method without any fix-point computation. For each branching bytecode, we just have to verify that the computed footprint is lower than the proof annotation. When the current bytecode is a method invocation, we compose the current footprint with the footprint of the invoked method (from the repository or from the temporary repository) line 25, the composition is detailed in the Algorithm 5.6.2.

The Algorithm 5.6.2 implements the composition in a slightly different way from the formal definition. The composition of $F_1$ and $F_2$ computed here is only the union of:

- $\{(s_i, s_j) \mid \exists k, (s_i, s_k) \in F_1, (s_k, s_j) \in F_2\}$, implemented by lines 2 to 8,
- $\{(s_0, s_i) \in F_2\}$, line 10,
- $\{(s_i, s_F) \in F_1\}$, line 11.

Instead of saturating the composition as soon as $(s_0, s_F)$ is found in it, since the actual implementation must directly reject the code, we simply test after this computation whether $(s_0, s_F)$ was added and fail if this is the case.

## 5.7 Conclusion

In this chapter we have proposed the most powerful control flow policies of our hierarchy of models. However, due to the size of target devices, it is not possible to add such an automaton to each application or even each domain. Thus we propose to use only one "global" policy for each system. This policy can be controlled by the card issuer, or the issuer can delegate the control to another party regarding that the loading mechanism will remain secure.

1: **for all** believed footprint $F_{A'.C'.m'}$ in the class file **do**
2: **if** $R[A'.C'.m'] \neq \perp$ or $R_{tmp}[A'.C'.m'] \neq \perp$ **then**
3: Drop $F_{A'.C'.m'}$
4: **else**
5: $R_{tmp}[A'.C'.m'] = F_{A'.C'.m'}$
6: **end if**
7: **end for**
8: **for all** footprint $F_{A.C.m}$ in the class file **do**
9: **if** nonvalid($F_{A.C.m}$) or $F_{A.C.m} \not\subseteq R_{tmp}[A.C.m]$ **then**
10: **return** FAIL
11: **end if**
12: **if** $A.C.m \leq A'.C'.m'$ and $F_{A.C.m} \not\subseteq F_{A'.C'.m}$ **then**
13: **return** FAIL
14: **end if**
15: **end for**
16: read proof annotations in array $proof$
17: **for all** method $A.C.m$ defined in class $C$ **do**
18: $F_{tmp} = F_{init}$
19: **for all** bytecode $i$ from $0$ to end **do**
20: **if** $P_{A.C.m}[i] \in$ branching bytecodes to address $a$ **then**
21: **if** $F_{tmp} \not\subseteq proof[A.C.m][a]$ **then**
22: **return** FAIL
23: **end if**
24: **else if** $P_{A.C.m}[i] =$ `invoke` $A'.C'.m'$ **then**
25: $F_{tmp} = compose(F_{tmp}, R[A'.C'.m'] | R_{tmp}[A'.C'.m'])$
26: **if** nonvalid($F_{tmp}$) **then**
27: **return** FAIL
28: **end if**
29: **else if** $P_{A.C.m}[i] =$ `return` and $F_{tmp} \not\subseteq F_{A.C.m}$ **then**
30: **return** FAIL
31: **end if**
32: **end for**
33: **end for**
34: drop $proof$
35: add all $F_{A.C.m}$ to $R$
36: **return** SUCCESS

**Algorithm 5.6.1:** Loading of a class $C$ of application $A$.

1: Let $F_{res}$ be initialized to $0$
2: **for all** $0 \leq i < n - 1$ **do**
3:  **for all** $0 \leq j < n$ such that $F_1[i \times n + j] == 1$ **do**
4:  **for all** $0 \leq k < n$ such that $F_2[j \times n + k] == 1$ **do**
5:  $F_{res}[i \times n + k] = 1$
6:  **end for**
7:  **end for**
8: **end for**
9: **for all** $0 \leq i < n$ **do**
10:  $F_{res}[i] = F_{res}[i] | F_2[i]$
11:  $F_{res}[i \times n + n - 1] = F_{res}[i \times n + n - 1] | F_1[i \times n + n - 1]$
12: **end for**

**Algorithm 5.6.2:** Composition of footprint: compose $(F_1, F_2)$.

# 6. Non-interference

Control flow policies described in previous chapters do not catch data flow between applications. For example, a malicious applet running on your mobile phone or smart card can disclose confidential information, financial data, address book, social security number and medical files, etc. If a system runs multiple software units, possibly untrusted, which share code (e.g. API) or data (e.g. collaborative applications), then the underlying information flow in terms of data must be verified in order to ensure data confidentiality. The security threat may originate from the code of the application or from code shared by some malicious software which cannot be trusted. Hence, allowing dynamic loading dictates the need to verify that the incoming applet respects desired non-interference properties. Thus the information flow certification must be done on-device, preferably at loading time, in order to avoid runtime overhead. Moreover, we argue that, in a runtime environment that supports deployment of several software units provided by self-sufficient issuers, the compilation of each software unit must be done without any knowledge about the other potential software. In this case, and because each runtime environment can embed a distinct set of software units, the only place where the whole software units can be checked "together" is the virtual machine where they will be run. Unfortunately, due to the high complexity of the algorithms and the lack of embedded resources, an information flow verifier has not yet been implemented on a small, embedded device.

Information flow analysis [39] has been actively investigated for several years, leading to a rich theory and language design, based on type-checking or static analysis. However, the information-flow based enforcement mechanisms have been scarcely applied in practice [47], even for desktop computers. The few practical approaches for embedded systems rely on the complete call graph of the application and deal with off-board static verification, and they do not address the challenges raised by open environments.

In previous works, we have proposed a tool, the STAN tool, that implements a model for checking secure information flow in Java-enabled, open, multiapplication, small systems. We keep the main features of the Java Virtual Machine including support for dynamic class loading and overriding. The tool computes *flow signatures* in two steps, an off-board analyzer, and an embedded verifier as described in Section 1.5.2. To our knowledge, this is the first implementation of an embedded information flow verifier. Experimental results show that our verifier could be efficiently used for our target systems and hence, it can be successfully applied in practice.

In this chapter, we first describe quickly the main principle of this tool in Section 6.1, then we propose in Section 6.2 non-interference policies for systems based on GlobalPlatform, or at least using the same concept of domains, and we propose to implement the verification of these policies in a new tool that takes flow signatures as an input in Section 6.3.

## 6.1 Overview of the information flow model of the STAN tool

In this section, we describe an information flow framework (in the sense of [39]) suitable for small systems, as it has been implemented in the STAN tool [5]. As flow signatures must be embedded with the code, the challenge is to make these signatures as concise as possible. We first identify sources of confidential data, which, in small systems, typically reside in instance fields. To express secrecy, we define a security lattice with two security levels (public and secret), with which class fields are labeled. In order to keep the model as compact as possible, we perform a *field independent*, but *security level sensitive analysis*. Thus, the abstract domain is limited to parameters and other abstractions for information flow sources (static fields, exceptions, return value of the method, etc.).

The support for openness and dynamic class loading is achieved by performing a compositional analysis. For each method we compute a context-insensitive *flow signature* which contains all potential flows between abstract values generated by the execution of the method. The definition of the flow relation between values allows to have a compact representation of flow signatures: one byte is sufficient to encode the possible flows between two abstract values. Non-interference (Section 1.3.3) is ensured if flow signatures do not contain any flows from confidential data to a public output.

### 6.1.1 Security levels

#### Security lattice

As already said, we define a security lattice with two security levels:

$$L = \{s, p\},$$

where $s$ stands for secret (high level) and $p$ stands for public (low security level). The order relation $\sqsubseteq$ between elements in $L$ is defined as follows: $p \sqsubseteq s$. Using this order relation, we define the security lattice $L$ of security levels.

Security levels are associated with information flow sources (objects' fields) and they should not be confused with Java modifiers (`private`, `public`, `protected`). While Java modifiers express accessibility within the Java language, the $s$ defined above expresses secrecy, the fact that the information must not be made accessible through information flow to unauthorized parties. The default security level of object's field is $p$, but restrictions on classes and their fields can be specified in an external file (*e.g.*, an text or XML file). We denote by $\mathcal{L}(C, f)$ the level associated with a field $f$ in a class $C$.

#### Security levels of object fields

Tracing each field is expensive and memory consuming, thus this is not adequate in the domain of mobile code and small systems. To reduce the size of information flow annotations, we perform a *field independent* but *security level sensitive* analysis. In a field independent analysis, all the fields in a structure are modeled as having the same location; thus, writing into one field is considered as writing into all the fields in the structure.

Thus, considering the security levels $L = \{s, p\}$, an object $o$ is modeled as two parts (locations): a secret part (denoted by $o^s$), for fields with security levels $s$, and a public part (denoted by $o^p$), for fields with security level $p$. We denote by $\mathcal{T}(o)$ the type of an element $o$. To deal with security levels at different field depths, we use the following convention:

- The secret part $o^s$ of an object $o$ contains all the field access paths that contain at least one field having the security level $s$:

$$o^s = \{o.f_1 \dots f_n \mid \exists 0 < i \leq n, \mathcal{L}(\mathcal{T}(o.f_1 \dots f_{i-1}), f_i) = s\}, \tag{6.1}$$

- The public part $o^p$ of an object $o$ refers to all fields of $o$ that contain only fields with security level $p$ on their access path:

$$o^p = \{o.f_1 \dots f_n \mid \forall 0 < i \leq n, \mathcal{L}(\mathcal{T}(o.f_1 \dots f_{i-1}), f_i) = p\}. \tag{6.2}$$

The choice of splitting an object in two parts was only dictated by the need to make memory saving for the embedded proof and for the flow signatures. We noticed that our choice of splitting an object in two parts is sufficient for applications from literature such as PACAP [12]. However, safe programs could be rejected because of this simplification. Yet, our experiments on real Java applications encourage us to believe that two parts are sufficient in most cases. Indeed, the object programming paradigm encourages the class proliferation instead of field proliferation within the same class.

### 6.1.2 Flow relation

Following the non-interference definition, we consider that there is a flow from an input $a$ to an output $b$ in a program $P$, denoted by $b \rightarrow a$, if an observer of $b$ (some untrusted piece of code that can access $b$ before and after the execution of $P$) can infer information about $a$.

We consider Java languages thus we can have aliases between objects, as well as primitive assignments and implicit flows. Because an alias may lead to further data propagation while primitive assignments do not, the origin of flows must be distinguished. This choice is imposed by the approximation of the field independent analysis. In a field sensitive analysis, this distinction is not mandatory, as every field of every object is tracked independently, and the flow type is given by the field type. Hence we type the flow relation with types in $\mathcal{F} = L \times L \times \{\mathtt{r}, \mathtt{v}, \mathtt{i}\}$. A flow $(l_1, l_2, t)$ from $a$ to $b$ is denoted by $b^{l_1} \xrightarrow{t} a^{l_2}$, where:

- *reference flows* (`r`), generate *interference through aliasing* and denote aliases that may lead to further data transfers,
- *value flows* (`v`), generate *interference through copy* and represent data transfer through explicit assignment of primitive types,
- *implicit flows* (`i`), generate *interference through inference* and stand for flows arising from the control structure of the program.

From an *interference through copy* we can obtain at least the same amount of information as it can be obtained from an *interference through inference*. In the same way, the *interference through aliasing* provides at least the same information and privileges as the *interference through copy*. As a conclusion, we consider that the amount of information leaked by an *interference through aliasing* is bigger than the one leaked by an *interference through copy*; similarly, the amount of information leaked by an *interference through copy* is bigger than the one leaked by an *interference through inference*. Therefore, we define an order relation on the types of flows:

$$\mathtt{i} \leq \mathtt{v} \leq \mathtt{r}.$$

This relation allows us to define a relation between flows, for two elements $(l_1, l_2, t)$ and $(l'_1, l'_2, t')$ of $\mathcal{F}$,we have:

$$((l_1, l_2, t) \leq (l'_1, l'_2, t')) \Leftrightarrow (l_1 = l'_1 \wedge l_2 = l'_2 \wedge t \leq t').$$

### 6.1.3 Flow signature of a method

Sensitive data are stored in object fields, while objects are made accessible to a method through parameters, objects allocated inside the method or objects returned by invoked methods. We compute the signature of a method by abstract interpretation and we use the *object allocation site model*: all the objects created/returned at the same program statement have the same abstraction. We need to consider abstract values representing all the elements of the program to perform the analysis but certain abstract values are locally defined inside a method and are not relevant outside. Thus, the global result must be restricted to the values that survive at the end of the method:

$$\Sigma_m = P \cup \{R, Static, Ex, IO\}$$

with

- $P$ the abstract domain for parameters of a method $m$ (they are denoted by $p_0$, $p_1$, ...),
- the return value of the current method, denoted by the abstract value $R$,
- input/output channels; all the channels are abstracted by a single value, $IO$,
- static fields; all static fields are modeled as the fields of a single object, denoted by the abstract value $Static$,
- exceptions; all thrown values flow to the abstract value $Ex$.

Note that in order to unify the model, we associate security levels with all abstract values, including those not abstracting objects. Hence, the input/output channels $IO$, the static world $Static$, the exceptions $Ex$ have the default security level $p$, as all sensitive data flowing to them potentially leak to unauthorized parties. Parameters of primitive type also have security level $p$. If a primitive instance field with a security level $s$ is given as parameter to $m$ in a particular context, its security level is taken into consideration whenever $m$ is invoked in the given context.

The flow signature of a method is an overapproximation of the set of flows potentially generated by the execution of that method. The flow signature carries relevant information for a later use of the method. The flow signature is the result of the analysis performed by the STAN tool:

**Definition 6.1.1 (Flow signatures)** *Let $m$ be a method. The flow signature of $m$, denoted by $S_m$, is the result of the analysis of the* STAN *tool. It is a subset of $\Sigma_m \times \Sigma_m \times \mathcal{F}$.*

with the following property:

**Property 6.1.2 (Flow signature soundness)** *The information flow signature of a method $m$ is sound with respect to non-interference: if there is no $(l_1, l_2, t)$ such that $(a, b, (l_1, l_2, t))$ belongs to $S_m$, then two concrete executions of the method $m$ starting in states that only differs by the value of $b^{l_2}$ will lead to final states having the same value of $a^{l_1}$.*

### 6.1.4 Enforcing non-interference

The flow signature of a method allows us to verify non-interference [24]: a program is secure w.r.t. non-interference if there is no flow of information from an abstract value with security level $s$ to an abstract value labeled with $p$. In our framework, a method is secure if for any flow $a^{l_1} \xrightarrow{\phi} b^{l_2} \in S_m$, we have $l_2 \sqsubseteq l_1$, according to the ordering relation on the lattice of security levels. Considering $L = \{s, p\}$, a method is secure if there is no flow from a $s$ value to a $p$ value. Hence, we can define non-interference w.r.t. to the flow signature of a method:

**Definition 6.1.3 (Non-interference for flow signatures)** *The flow signature of a method, $S_m$, is secure w.r.t. to non-interference if it does not contain flows of type $a^p \xrightarrow{\phi} b^s$, except the case when $a$ represents the return value of the method, $R$. The verification of flows of confidential data to return values is postponed until the method is invoked.*

### 6.1.5 Implementation

The STAN tool proposes an off-board analysis that computes the flow signatures and that can be embedded and verified on-device using lightweight bytecode verification as exposed in Section 1.5.2 (see [21] for more details).

## 6.2 Non-interference policies for GlobalPlatform

As already said in the previous chapters, GlobalPlatform specifications do not clearly define permitted/forbidden interactions between applications according to their installation domains in order to let implementations fit specific needs and objectives. If we want to describe how secret data can circulate between applications, we have to define specific policies.

The simplest way to consider authorized data transfers is to consider that secret information can only be shared inside a security domain and cannot leak outside. We call this situation *isolation*. Other architectures may be considered to allow more flexible information transfers. Method calls can be allowed through shareable interfaces and we have seen in the previous chapters that calls to methods can be controlled by a policy mechanism. However, this control mechanism does not prevent from illegal data flow between domain. For example, let us consider an application $A_0$ that is authorized to call a method $A_1.C_1.m_1$ and is not authorized to call any method of application $A_2$. If $A_1$ is authorized to call some methods of $A_2$ then data flow may go from $A_0$ to $A_2$ and/or from $A_1$ to $A_0$ through parameters and return values. Thus, we also want to constrain data flows and, for this purpose, we use the non-interference signatures to track potential illegal data flows between domains.

### 6.2.1 GlobalPlatform policies

In order to express security policies describing collaborations and data flows between domains, we add a policy to each domain. The security policies can be expressed with a very simple language, but have enough power to model collaborations schemes in a smart card, with extension capabilities. In a smart card, the entities exchanging or sharing data are the applications, but applications have to comply with the policies of the domains in which they are loaded. Thus the language expresses policies of domains but the concrete rules will be verified on the code of applications. As we consider that

confidential data resides in class fields, a policy expresses flows authorized for data that resides in *secret* fields.

In GlobalPlatform the set of domains is a forest. A forest $h$ is completely defined by $h = (V, pred)$ where $pred$ is a function that associates $pred(v)$ to $v$ if there is an edge from $pred(v)$ to $v$ in $h$.

**Definition 6.2.1** *Let $\mathcal{S} = (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P})$ be a system. We denote by $h_{\mathcal{S}} = (\mathcal{D}, pred_{\mathcal{S}})$ the hierarchy of domains of $\mathcal{S}$.*

The card issuer will provide some policy rules and functions to verify these rules. Then, a non-interference rule is a function that allows to define for each domain $d$ a set of domains from which the applications can acquire secret data of $d$, depending on the system it belongs to. $Systems$ denotes a set of systems.

**Definition 6.2.2 (Non-intererence policy rule)** *A non-interference policy rule is a function $nip : Domains \times Systems \longrightarrow \wp(Domains)$ where $nip(d, (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P})) = \emptyset$ when $d \notin \mathcal{D}$.*

Let us consider some examples of such policy rules that we can define. The simplest one is the isolation policy rule that restricts access of the data of a domain to the domain itself in any hierarchy. It can be defined by:

$$\begin{array}{rccc} isolation: & Domains \times Systems & \longrightarrow & \wp(Domains) \\ & d, \mathcal{S} & \longmapsto & \{d\} \end{array}$$

We could also consider that a domain wants to give access to its data to each of its direct sub-domains, then we have:

$$\begin{array}{rccc} subdomains: & Domains \times Systems & \longrightarrow & \wp(Domains) \\ & d, \mathcal{S} & \longmapsto & \{d' \mid d = pred_{\mathcal{S}}(d')\} \end{array}$$

These non-interference policy rules do not have to be defined by the programmers, each device supports a non-interference language that has been implemented by the issuer.

**Definition 6.2.3 (Non-interference language)** *A non-interference language is a set of non-interference policy rules that contains at least the function* isolation*.*

And the non-interference policy of the system is a mapping that attaches a rule of the language supported by the system to each domain of the system.

**Definition 6.2.4 (Non-interference policy of a system)** *Let $\mathcal{S} = (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P})$ be a system that supports the non-interference language $\mathcal{N}$. The non-interference policy $\mathcal{P}$ is a mapping $\mathcal{P} : \mathcal{D} \longrightarrow \mathcal{N}$ such that secret data are authorized to flow from a domain $d$ to the domains $\mathcal{P}(d)(d, \mathcal{S})$.*

Then, we can define what is a secure method for a system equipped with a non-interference policy.

**Definition 6.2.5 (Secure method)** *Let $\mathcal{S} = (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P})$ be a system that supports the non-interference language $\mathcal{N}$. A method $m$ installed in the domain $d \in \mathcal{D}$ is secure if and only if:*

$$\forall (a, b, (s, s, t)) \in S_m, \delta(\mathcal{T}(a)) \in \mathcal{P}(\delta(\mathcal{T}(b)))(\delta(\mathcal{T}(b)), \mathcal{S})$$

As we use for each system a language that is a subset of the rules defined by the card issuer, the policy definition is flexible and several systems issued by the same card issuer can be equipped with different policy languages. Moreover, the policy is adapted to the system configuration: the set of domains that have access to a given data depends on the domains hierarchy of the system.

**Example 3** *Remember example 1 of Chapter 3. Application $jTicket$ provides to the card holder certain number of tickets for public transportation. Tickets can be bought using $ePurse$, and the act of ticket purchasing requires the invocation of service $ePurse.payment$ by $jTicket$. The $Bank$ security domain owner allows data exchanges between $EMV@Bank$ and $ePurse@Bank$, and between $ePurse@Bank$ and $jTicket@Transport$, but not between $EMV@Bank$ and $jTicket@Transport$. On the other hand, $Transport$ owner not only allows data exchanges between $ePurse@Bank$ and $jTicket@Transport$, but she actually needs this exchange, otherwise her application is useless. Then, we can define that $\mathcal{P}(Bank)$ is $isolation$, and $\mathcal{P}(Transport)$ depends on the card configuration: if $Transport$ is a subdomain of $Bank$, we can use for example:*

$$\begin{array}{rlll} father: & Domains \times Systems & \longrightarrow & \wp(Domains) \\ & d, \mathcal{S} & \longmapsto & \{d' \mid d' = pred_{\mathcal{S}}(d)\} \end{array}$$

*or if $Transport$ and $Bank$ are two subdomains of the same domain:*

$$\begin{array}{rlll} brothers: & Domains \times Systems & \longrightarrow & \wp(Domains) \\ & d, \mathcal{S} & \longmapsto & \{d' \mid pred_{\mathcal{S}}(d) = pred_{\mathcal{S}}(d')\} \end{array}$$

## 6.3 On-device verification

Besides the signatures of methods that are verified and recorded by STAN tool, we have to verify that incoming applications respect the non-interference policies of the domains of the card.

### 6.3.1 On-device meta-data

We have to keep on-device a function for each element of the language supported by the device. This code is part of the trusted base of the system and has to be provided by the card issuer.

To simplify the policy management on the device, we precompute the authorized flows between domains and keep a two dimensional array of bits $dataflow$ of size $n$ if the smartcard can support $n$ domains. Then $dataflow[i][j] == 1$ means that information can flow from domain $i$ to domain $j$. As secrets can circulate at least inside a domain, $dataflow[i][i] == 1$ for each $i$.

Let us remark that for a system, that can support $n$ domains and on which $k$ domains have been installed, $dataflow[i][j] == 0$ for each $k \leq i < n$, $k \leq j < n$, $i \neq j$.

### 6.3.2 Verification of a new application

When an new application is loaded with the function $install(A, d_i, \mathcal{S})$, the signatures the methods of $A$ are verified using the STAN tool one by one. When the STAN verification of a method $m$ succeeds, then we have an assurance that the code received respects the signature of flows. We have to add a next step consisting in the verification of the compliance of the signature with the policy of the device in terms of non-interference.

**Definition 6.3.1 (On-device verification)** *An application $A$ is compliant with the non-interference policy of a system if for each method $m$ of each class $C$ of $A$, for each $a^{l_1} \xrightarrow{\phi} b^{l_2} \in S_m$:*

- *either: $l_2 = p$,*
- *or: $l_1 = s$ and $\delta(\mathcal{T}(b)) = d_j$ and $dataflow[i][j] == 1$,*
- *or the verification fails.*

### 6.3.3 Addition of a domain

When a new domain is added on the system with the $add(d, \mathcal{S})$, we assign to it the most secure policy rule, that is "isolation". The encoding of domains assigns to $d$ the next available index, let say $i$. The $dataflow$ array does not need to be modified as it already has $dataflow[i][i] == 1$ and $dataflow[i][j] == 0$ for $i \neq j$.

We may also want to add a new domain with a specific policy rule. For this purpose, we propose to overload the function $add$:

$$add(d, p, \mathcal{S}) = (\mathcal{D} \cup \{d\}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P}[d \mapsto p]).$$

To perform this addition, we have to:

- check that the policy rule $p$ belongs to the policy language of $\mathcal{S}$,
- check that the new policy rule is compliant with the already loaded ones,
- update the $dataflow$ array.

The verification algorithm is given in the Algorithm 6.3.1. First, we have to apply the policy rule of the new domain to the current hierarchy of domains of the system, Line 4. Then, we have to check the compliance with the existing policy rules, that is to check that the implicit transitivity of the policies does not break the requirements of the new domain (done Lines 5 to 11). Then, if the verification succeeds, we update the array of authorized flows (Lines 12 to 14).

### 6.3.4 Addition of a non-interference policy

We can also consider the case when the non-interference language of the system has to be updated. This change is a major change in terms of security since the goal is to modify the verification mechanism of the system. A new policy element is a pair, containing the name of the policy rule (that will be later used by the domains) and the code of the function itself used to compute it on a given hierarchy of domains. We cannot handle the verification of such an element since we do not provide verification of behavioral specifications. This extension mechanism can only be available for the card issuer and the code of the function that implements the policy rule has to be signed. However, as soon as we consider that the new code is trusted, the addition can be done without any verification on-device: already existing domains are not impacted. If a domain is later loaded with a new policy rule, the verification will be performed as described in the previous subsection.

```
1: if p does not belong to the language of S then
2:    return Fail
3: end if
4: Compute the set A = p(d, S)
5: for all j in A do
6:    for all 0 ≤ k < n do
7:       if dataflow[j][k] == 1 and k ⊈ A then
8:          return Fail
9:       end if
10:   end for
11: end for
12: for all j in A do
13:   dataflow[i][j] == 1
14: end for
15: return Success
```

**Algorithm 6.3.1:** Addition of domain $i$ with policy $p$ on system $\mathcal{S}$.

## 6.4 Conclusion

Privacy and confidentiality represent a real concern in modern computing systems especially in small open systems. Hence techniques to ensure their security are required. The new model described in this chapter permits to detect undesirable data flows in terms of non-interference in GlobalPlatform.

The model is easy to use. The main limitation to support evolution of systems comes from the choices made in the STAN tool: once the flow signatures are computed, it is not possible to change the security levels of data.

# 7. Conclusion

This document defines a set of compositional techniques for loading time on-device verification of several security properties. These techniques contribute to information protection security property verification and can be used on such restricted devices as smart cards. In the presence of evolution, when applications can be updated or removed, and new applications can arrive, the system should be able to verify autonomously the received updates. To do that trusted card manager can be extended implementing the proposed techniques.

One type of considered security property is a control flow of the application services invocations. The document contains techniques for autonomous loading time verification of a direct control flow, a transitive control flow and a global control flow, which also takes into account an order of service invocations and provides a basis for integration with the notation of WP4.

Another considered security property is a classical non-interference, that is implemented for smart cards using lightweight verification approach. The document extends this verification process with a technique for compositional verification of a security domains hierarchy. The hierarchy is equipped with a non-interference policy and the system is able to verify at load-time that the updated application complies with this policy.

The techniques that are described in the document can be extended to support the updates of the device security policies. Task 6.3 only assumes conservative updates that are incremental and do not require the re-verification of the applications already installed on the device. Task 6.4 solutions will provide more complex techniques as extensions for methodologies proposed in this deliverable, that will support an autonomous on-device load time verification of all possible types of application and device security policy updates, which are related to information flow.

# Bibliography


[1] Common criteria. http://www.commoncriteriaportal.org.

[2] GlobalPlatform specifications. http://www.globalplatform.org/.

[3] Java Cardspecifications. http://java.sun.com/javacard/.

[4] Open mobile terminal platform. http://www.omtp.org.

[5] Stan, an information flow analysis for small embedded systems. http://stan-project.gforge.inria.fr.

[6] A. Armenteros, B. Chetali, M. Felici, F. Massacci, V. Meduri, A. Tedeschi. Selected change requirements and security properties. Report D1.1.1, 2010. Submitted to the EU Consortium.

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

[8] M. Avvenuti, C. Bernardeschi, N. D. Francesco, P. Masci. A tool for checking secure interaction in Java Cards. Proc. of EWDC2009, 2009.

[9] V. banking services. http://www.venyon.com/banking. Available on the web.

[10] G. Barthe, D. Gurov, M. Huisman. Compositional verification of secure applet interactions. Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering, wolumen 2306 serii Lecture Notes in Computer Science, strony 15–32. Springer, 2002.

[11] F. Besson, T. Blanc, C. Fournet, A. D. Gordon. From stack inspection to access control: a security analysis for libraries. Proceedings of the 17th workshop on Computer Security Foundations (CSFW04), strona 61. IEEE Computer Society, 2004.

[12] P. Bieber, J. Cazin, P. Girard, J.-L. Lanet, V. Wiels, G. Zanon. Checking Secure Interactions of Smart Card Applets: extended version. Journal of Computer Security, 10:369—-398, 2002.

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

[14] M. credit cards. http://www.miles-and-more.com. Available on the web.

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

[16] D. Deville, G. Grimaud. Building an “impossible” verifier on a Java Card. Proc. of WIESS’02, strony 2–2. USENIX Association, 2002.

[17] N. Dragoni, F. Massacci, K. Naliuka, I. Siahaan. Security-by-contract: toward a semantics for digital signatures on mobile code. Proceedings of the 4th European PKI Workshop: Theory and Practice (EuroPKI07). Springer, 2007.

[18] N. Dragoni, F. Massacci, C. Schaefer, T. Walter, E. Vetillard. A Security-by-Contract Architecture for Pervasive Services. SECPerU, strony 49–54, 2007.

[19] W. Enck, M. Ongtang, P. McDaniel. On lightweight mobile phone application certification. CCS09, strony 235–245, Chicago, IL, USA, November 2009. ACM.

[20] E. Fromentin, C. Jard, G.-V. Jourdan, M. Raynal. On-the-fly analysis of distributed computations. Information Processing Letters, 54(5):267–274, 1995.

[21] D. Ghindici, G. Grimaud, I. Simplot-Ryl. An information flow verifier for small embedded systems. Proc. Workshop in Information Security Theory and Practices 2007 Smart Cards, Mobile and Ubiquitous Computing Systems, number 4462 serii Lecture Notes in Computer Science, strony 189–201, Heraklion, Crete, Greece, 2007. Springer-Verlag.

[22] D. Ghindici, I. Simplot-Ryl. On practical information flow policies for java-enabled multiapplication smart cards. Proc. 8th Smart Card research and Advanced Application IFIP Conference (CARDIS 2008), wolumen 5189 serii Lecture Notes in Computer Science, strony 32–47, Egham, Surrey, UK, 2008. Springer.

[23] P. Girard. Which security policy for multiapplication smart cards? USENIX Workshop on Smartcard Technology. USENIX Association, 1999.

[24] J. A. Goguen, J. Meseguer. Security policies and security models. Proc. IEEE Symposium on Security and Privacy, strony 11–20, 1982.

[25] D. Gurov, M. Huisman, C. Sprenger. Compositional verification of sequential programs with procedures. Information and Computation, 206(7):840–868, 2008.

[26] M. Huisman, D. Gurov, C. Sprenger, G. Chugunov. Checking absence of illicit applet interactions: a case study. FASE’04, wolumen 2984 serii Lecture Notes in Computer Science, strony 84–98, 2004.

[27] G. Inc. GlobalPlatform Card Specification, Version 2.2. Specification 2.2, 2006.

[28] I. Ion, B. Dragovic, B. Crispo. Extending the Java Virtual Machine to Enforce Fine-Grained Security Policies in Mobile Devices. ACSAC, strony 233–242, 2007.

[29] J. Jürjens. Secure Systems Development with UML. Springer Verlag, 2004.

[30] X. Leroy. On-card bytecode verification for Java Card. Smart Card Programming and Security, wolumen 2140 serii LNCS, strony 150–164. SV, 2001.

[31] X. Leroy. Bytecode verification on Java smart cards. Software – Practice & Experience, 32(4):319–340, April 2002.

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

[33] I. Narasamdya, M. Périn. Certification of smart-card applications in Common Criteria. SAC '09, strony 601–608. ACM, 2009.

[34] N. Narasimhan, R. Vemuri. Specification of control flow properties for verification of synthesized vhdl designs. Proc. 1st International Conference on Formal Methods in Computer-Aided Design (FMCAD'96), wolumen 1166 serii Lecture Notes in Computer Science, strony 327–345. Springer, 1996.

[35] G. C. Necula. Proof-carrying code. Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL97), strony 106–119, Paris, France, 1997.

[36] M. Ongtang, S. McLaughlin, W. Enck, P. McDaniel. Semantically rich application-centric security in Android. Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC09), strony 340–349. IEEE Computer Society, 2009.

[37] P. project website. http://www.cert.fr/francais/deri/wiels/pacap/.

[38] E. Rose. Lightweight bytecode verification. Journal of Automated Reasoning, 31(3-4):303–334, 2003.

[39] A. Sabelfeld, A. C. Myers. Language-based information-flow security. IEEE Journal on Selected Areas in Communications, 21(1):5–19, 2003.

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

[41] G. Schellhorn, W. Reif, A. Schairer, P. Karger, V. Austel, D. Toll. Verification of a formal security model for multiapplicative smart cards. ESORICS'00, wolumen 1895 serii Lecture Notes in Computer Science, 2000.

[42] SecureChange. Deliverable 4.1, 2010. http://securechange.eu/sites/default/files/deliverables/D4.1_Security_Modelling_Notation_for_Evolving_Systems.pdf.

[43] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, D. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. strony 15–28, 2003.

[44] G. J. C. solutions. http://www.gemalto.com/techno/javacard/. Available on the web.

[45] M. Tanaka. A smart card evaluation experience under a Japanese scheme. http://www.commoncriteriaportal.org/iccc/9iccc/pdf/b2502.pdf. Presentation available on the web., 2008.

[46] N. Yoshioka, H. Washizaki, K. Maruyama. A survey on security patterns. Progress in Informatics, (5):35–47, 2008.

[47] S. Zdancewic. Challenges for information-flow security. Proc. 1st International Workshop on Programming Language Interference and Dependence (PLID04), Verona, Italy, 2004.