D6.4: IMPACT ANALYSIS OF NEW SECURITY REQUIREMENTS

Arnaud FONTAINE (INR-Li), Olga GADYATSKAYA, Fabio MASSACCI (UNITN), Julien BERNET, Quang-Huy NGUYEN (GTO), Fabrice BOUQUET (INR-FC), Frank PIESSENS (KUL), Michela ANGELI (UNITN)

Document Information

Document Number	D6.4
Document Title	Impact analysis of new security requirements
Version	1.0
Status	Final version
Work Package	WP 6
Deliverable Type	Report
Contractual Date of Delivery	M24
Actual Date of Delivery	M24
Responsible Unit	INR
Contributors	INR, UNITN
Keyword List	Verification, On-device
Dissemination	PU

Document change record

Version	Date	Status	Author (Unit)	Description
0.1	15/11/2010	Working draft	O. Gadyatskaya, F. Massacci (UNITN)	First working draft of Direct Control Flow chapter, draft of Introduction, chapters on Criteria and Integration with WP7 added
0.2	17/11/2010	Working draft	A. Fontaine (INR)	First working draft of Transitive Control Flow chapter
0.3	26/11/2010	Working draft	O. Gadyatskaya (UNITN)	Updated Direct Control Flow chapter (UNITN)
0.4	01/12/2010	Working draft	A. Fontaine (INR)	Updated Transitive Control Flow chapter and first draft of Global Policy chapter (INR). first draft of the strategic overview
0.5	04/12/2010	Working draft	A. Fontaine (INR)	Updated Transitive Control Flow, Global policy and Non-interference chapters (INR). Added section in introduction related to validation results plan
0.6	06/12/2010	Working draft	O. Gadyatskaya (UNITN)	Finalized draft of Direct Control Flow chapter
0.7	06/12/2010	Working draft	A. Fontaine (INR)	Finalized draft of Transitive control flow, global policy chapters. Updated draft of the non-interference chapters
0.8	09/12/2010	Working draft	A. Fontaine (INR), O. Gadyatskaya (UNITN)	Updated executive summary and strategic overview of the deliverable, Integration with WP7 chapter added
0.9	16/12/2010	Working draft	A. Fontaine (INR)	Updated executive summary, introduction, conclusions and chapters 3-5

0.10	17/12/2010	Working draft	O. Gadyatskaya (UNITN)	Updated executive summary, introduction, conclusions and chapter 2
0.11	04/01/2011	Working draft	M. Angeli (UNITN)	Quality check completed, minor remarks
1.0	07/01/2011	Final version	A. Fontaine (INR), O. Gadyatskaya (UNITN)	Integration of F. Piessens (KUL) and M. Angeli (UNITN) remarks

Executive summary

In this deliverable we finalize the on-device verification techniques introduced in the deliverable D6.3 [11] and devoted to the enforcement of the "secure extensibility" property of the HOMES use case and the "information protection by flow control" property of POPS use case. Four different techniques are presented: direct control flow, transitive control flow, global policy and non-interference. In the deliverable D6.3 only incremental types of updates were considered (application installation, weakening security policies) in these models, while in this deliverable we introduce algorithms for decremental types of updates (application removal, restricting security policies, *etc.*).

With the direct control flow verification technique we have presented Security-by-Contract framework for smart cards. We provided specification of the PolicyChecker component for incremental types of updates and discussed how the framework can be integrated with the Java Card system. In the Chapter 2 of the current deliverable we finalize the introduction of the Security-by-Contract framework with specifications of the ClaimChecker and the ConflictResolution components, and specification of the PolicyChecker component for the decremental types of changes.

Transitive control flow verification technique aims to capture illicit invocations of application methods especially in case of applications collusion. In the Chapter 3 of this deliverable we finalize this technique by the description of several solutions for dealing with decremental updates, each solution having a different trade-off between computation overhead and additional system memory required.

Global policy verification technique aims to detect forbidden sequences of methods calls at the system level, *i.e.* not necessarily only within one or two applications. This approach is inspired from proof-carrying-code (PCC) paradigm: static bytecode analysis conducted off-device generates proofs annotations embedded in the bytecode for easier on-device verification. In the Chapter 4 of this deliverable we describe how to deal efficiently with decremental changes on-device with this model, mainly application removal because updates of the security policy (sets of forbidden sequences of method calls) have an impact on already loaded code stronger than expected and thus requires additional off-device computations but also on-device verifications.

Non-interference verification technique is also a PCC-like approach but whose goal is to detect illicit flows of data between applications clustered in *domains*. Even if the domain abstraction is strongly inspired from the GlobalPlatform environment, it is generic enough to be applied to any Java-based system. In the Chapter 4 of this deliverable we describe how to efficiently deal with decremental changes on-device with this model.

All the aforementioned techniques support security policy updates. If a system security policy is updated the incremental on-device verification procedures will ensure that all the applications are compliant with new policy. Two approaches are sketched in case some of installed applications are not compliant with new policy. Either the policy update is rejected or the applications conflicting with new policy are made non-selectable.

Depending on the system requirements and stakeholders' needs it is possible to choose the most suitable verification technique. The work remaining for the last year of the project is the implementation of core algorithms for one of the SecureChange project case studies (POPS or HOMES).

WP6 integration with other work packages

The Figure 1 reports the whole integration process between work packages across case studies of the project. The content of each integration link on this graph is the description of the relation between artifacts produced by each work package involved instantiated on the labeled case study. The WP6 is involved in two integration links on the POPS case study with WP4 and WP7.

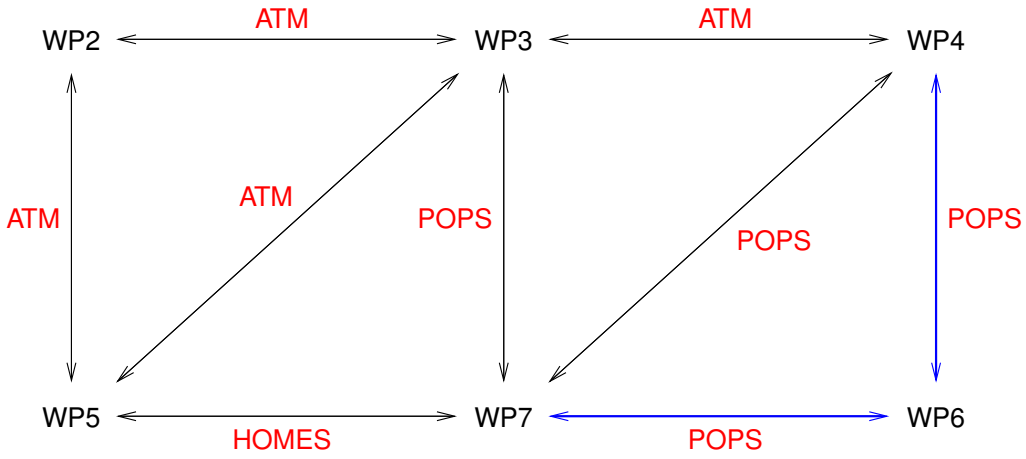


Figure 1: Integration of work packages across case studies.

Integration with WP4

The content of the integration with WP4 on POPS case study is included in the Chapter 4 of the deliverable D4.2. The main idea of the collaboration with WP4 is to verify the same *properties* at the model level using WP4 techniques and at the code level using WP6 techniques to establish a coherency between (high-level) modeling of applications and their (low-level) implementations.

Since WP4 and WP6 have independently worked on different properties from the beginning of the project, we choose to focus on information protection related properties, and more precisely on three of the four models previously developed by the WP6 (see the deliverable D6.3): the two control flow models and the non-interference model. For each of these models integration has been achieved by the establishment of new specific UMLsec stereotypes. For each WP6 model/WP4 stereotype, we rely on the same input, that is the security policy to be enforced. Furthermore, modifications on the model and the code are both dealt in incremental/decremental way to avoid full re-verification of the model and/or the code.

In addition to verify the same properties at different levels, a coherency report is established between UML models and the code analyzed. Actually, upon successful verification at the model level, some information is extracted from UML to permit additional verifications on the code and thus detect potential incoherencies between application(s) design and implementation.

Integration with WP7

The content of integration link between Verification (WP6) and Testing (WP7) work packages is provided in the Chapter 6 of the current deliverable. The main contribution of

collaboration between these work packages is stronger security guarantees for the information protection security property of the POPS case study.

Both WP6 and WP7 work on the information protection property of the POPS case study. This requirement demands that assets (application data and specific services of security domains) of each stakeholder should be protected from unauthorized access. WP6 (on-device verification) provides techniques to ensure absence of illegal access to information data. WP7 is interested in the access to security domains services. It verifies by testing absence of possibility to misuse application installation and re-association processes, which grant direct access to security domains services.

In terms of integration we discussed threats for the information protection property and demonstrated that collaboration of two work packages provides protection against these threats. One of the main advantages of the collaboration is possibility for WP6 to rely on some assumptions about the installation process, because WP7 is testing correctness of these assumptions. Another benefit is possibility to ensure absence of illegal transitive access to the security domains services, which can be verified by the techniques of WP6.

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 1: 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 cards or other small devices with restricted capabilities
OPEN	Card manager on GlobalPlatform

Table 2: Glossary

1. Introduction

Various techniques for loading-time verification on small and memory-constrained devices were presented in the deliverable D6.3 [11]. The main goal of these techniques is to verify absence of illicit information flow paths between applications. As we assume significant constraints on auxiliary memory and computation time, the techniques need to be incremental and highly optimized.

The current deliverable finalizes the formal models and algorithms for verification process and is an intermediate step between them and implementation. Thus we present the formal models in a connection with real devices, for example, Java Card and GlobalPlatform smart cards. As the WP6 (on-device) techniques will be validated on the POPS and HOMES case studies [5], the natural choice is Java as an underlayer in each model. As standard Java is a well-known language we do not provide a lot of details and refer an interested reader, first of all, to the official documentation [16].

POPS case study presents a multi-application smart card which is based on Java Card and GlobalPlatform specifications. As Java Card has not quite the same conceptions as Java, we provide in this chapter more details about Java Card and GlobalPlatform.

HOMES case study is based on the OSGi technology [2]. As on the moment of writing this deliverable the HOMES code and detailed specification were not available, we cannot provide applicability study yet.

1.1 Java Card and GlobalPlatform

In this section we detail some properties of Java Card and GlobalPlatform used later in this deliverable and not described in the Chapter 1 of the deliverable D6.3.

1.1.1 Java Card Application Communications

Java Card firewall protects methods and data of one application to be accessed from other packages [14, sec.6]. Specific means for applications to communicate are provided by shared interfaces. These interfaces, in order to be accessible, necessarily extend interface `Shareable`. In the remainder we will sometimes call such shared interfaces Shareable interfaces.

Application services are methods of such shared interfaces. Applications that provide services are called *servers* and applications invoking services are called *clients*. The main property of services is possibility to invoke them from contexts different from the server context (we refer an interested reader to [14, sec.6] for details of sharing mechanism and contexts).

One of our main interests is to verify absence of (direct and transitive) illegal services invocation. Currently on Java Card access control to services is enforced in the application code. Such a run-time check can be substituted by a loading-time verification.

1.1.2 Java Card Bytecode Details

The Java Card bytecode (JC bytecode) that is sent to the card in the CAP files differs from the standard Java bytecode in the class files [15, sec.4]. In order to optimize the work of the bytecode verifier and the JCVM, JC bytecode is compressed by using tokens instead of Unicode strings. Because of this tokenization in the JC bytecode we see `superInterfaces 1.2` rather than `superInterfaces javacard/framework/Shareable`. Consequently, the verification mechanism working on a bytecode needs to be able to connect tokens with the actual interfaces, methods and classes. This process, which is called token-based linking, is actually performed by the installer during application installation and linking process.

Token-based linking is performed by JCRE using CAP files and export files. Export files provide a specification of public APIs of the CAP file. Thus the export file should include information about all shared interfaces declared by the application. When a package is loaded on the card not only its own CAP files and export file are being sent, but also export files of all the packages it refers to (including `javacard/framework`). It is done in order to ease the linking process. We note that export files contain also actual names of the shared interfaces and their services. Thus service name for us can be represented as actual name of the method (a string). On Java Card it is more practical though to use bytearrays as service names, using encoding similar to AID (details about application identifiers can be found in [11] or [9]).

1.1.3 Security Domains on GlobalPlatform

GlobalPlatform (GP) is a specification for secure card content management [12]. It organizes card contents and provides to application providers secure means for management of their data. Each application provider (and the card issuer) normally have (at least one) security domain, which is a privileged application that manages other applications' data and provides secure connections between applications and their provider authenticated off-card.

Security domains normally are not valuable for the card holder, but they ease life of the application owners, as they provide specific services to the associated applications. For Java Card and GlobalPlatform smart cards these services are described in the GP specification [12, App. A]. Specifically, a security domain on such a card usually implements `SecureChannel` or `SecureChannelX` interface, which declare a well-defined set of communication-assisting methods. We will call such methods *GP-specific services of a security domain*.

1.1.4 GlobalPlatform Privileges

One of the advantages of GP is hierarchical organization of the card. The card issuer is the main stakeholder on GP card and she can manage any data on the card through the Issuer's Security Domain (ISD). Application providers are less privileged, but they also can have some capabilities to manage applications on the card.

The following privileges are required in order to modify the content of the card [12, sec.6.6]:

- Delegated Management – enables a security domain with a capability of delegated card content management.
- Authorized Management – enables a security domain with a capability of card content management.
- Global Delete – enables a security domain with a capability to delete any card content.
- Global Lock – enables a security domain with a capability to lock and unlock any application.

Delegated Management and Authorized Management privileges allow owner of a security domain with these privileges to manage the card contents (including deletion of executable files of other security domains). Delegated Management and Authorized Management are mutually exclusive privileges. Other privileges can be assigned independently and one security domain can have multiple privileges. Normally the privileges are assigned to the security domains (and not simple applications).

There are other privileges that also can be considered:

- Card Lock – enables an application or a security domain with a capability to lock the card
- Card Terminate – enables an application or a security domain with the privilege to terminate the card

After removal of some privileged entity its privileges are assigned to the issuer's security domain. We note that security domains themselves are distinguished from applications by a specific Security Domain privilege.

1.1.5 Security Domains Hierarchy

An application (and security domain) are always associated explicitly with some security domain [12, sec.7.2]. Implicitly each application is associated with all the domains higher in the hierarchy.

Security domains hierarchy on the card is a forest. Roots are security domains associated with themselves. Issuer's security domain is always a root.

Extradition is the means by which an application (or security domain) can be associated with a security domain different from the one it was installed into. Security domains and applications can be extradited during installation or subsequently.

1.2 Expected techniques properties from DoW

Some properties are expected to be featured by WP6 (on-device) techniques, as those already given in the Section 1.4 of the deliverable D6.3 [11]. It is important to clarify that the main originality of the techniques we propose in this work package is not related to the originality of enforced properties (absence of some control flow paths, absence of forbidden sequences of method calls, or absence of undesired information flows), but is

about providing practicable and efficient on-device algorithms for verifying such properties directly on constrained devices, such as smart cards (POPS case study).

The four models we presented in the deliverable D6.3 have been especially designed to fit the requirements of constrained devices in case of installation of new applications. In this deliverable we focus on other kinds of changes in these models: removal of an application and modification of the security policy. Once more, our goal is not only to permit such changes, but to provide practicable and efficient on-device algorithms. Since we investigate incremental/decremental techniques to deal with changes in our models, we have to formally analyze the impact of each kind of change in order to provide sufficient guarantees that our incremental/decremental techniques achieve the same results as a complete (re-)analysis of a system, and more precisely that the system systematically evolves from a secure state to another secure state w.r.t. to the considered model.

The on-device verification processes we developed can be seen as a classifier that assigns a category to each requested change, *positive* or *negative*: *positive* means “the change breaks the security of the system” and *negative* means “the change does not break the security of the system”. The category inferred by a verification process can be wrong because of approximations introduced to make it more suitable for constrained devices. The results of a verification process are thus splitted into four categories:

- *true positives*, set of changes rejected because they would break the security of the system;
- *true negatives*, set of changes accepted because they do not break the security of the system;
- *false positives*, set of changes rejected while they would not break the security of the system;
- *false negatives*, set of changes accepted while they break the security of the system.

Obviously, false negatives must be avoided. Existence of false negatives would clearly denote a lack of caution in the design of the approach, so we have paid a special attention to produce no false negatives. False positives do not break the security of the system, but they denote a lack of sensitivity of the approach. Solutions creating false positives should be avoided as much as possible for lifelong evolving systems because they can bring the system to a state where no new application can be installed.

2. Direct Control Flow

2.1 Introduction

Security-by-Contract (S×C) approach for smart cards was introduced in the deliverable D6.3 [11]. The goal of the framework is to verify at loading time absence of unauthorized direct service (run-time) invocations. We have presented main idea behind loading-time verification with S×C approach, introduced formal models of the smart card platform and application contract and discussed two main components of the S×C framework (the ClaimChecker and the PolicyChecker). The PolicyChecker specification was introduced for incremental types of changes (installation of new application and incremental update of already installed one). We also presented some results on validation of the S×C framework, showing that if the ClaimChecker and the PolicyChecker are implemented correctly, then smart card platform is secure (for introduced definition of secure platform).

In this deliverable we provide specification of the ClaimChecker and extension for specification of the PolicyChecker (for decremental types of changes). We also introduce new component of the S×C framework: the ConflictResolution component. This component finalizes the S×C framework for smart cards. Further we will provide description and specification of this new component and show how security of the platform can be established in the presence of it.

Conflict resolution is required for some cards where hierarchical organization of the stakeholders enables more advanced reasoning about updates than always rejecting potentially dangerous ones. The system should be kept secure under evolution, but in some cases it is possible to accept update of more privileged stakeholder ensuring security by making conflicting applications unselectable. Introduction of the ConflictResolution component is one of the goals of the next deliverable (Deliverable 6.5, which is due on M36). However we provide description of this component in the current deliverable because it finalizes formal specification of the framework. We will also discuss some details of the ConflictResolution implementation and point to problems related to it.

Two approaches may be chosen for implementation of the S×C framework: with and without conflict resolution. The main goal of the framework is to keep the system secure and both approaches ensure security across updates. Card vendors can choose an approach that is more appealing to them and their customers.

Some definitions from the deliverable D6.3 [11] were updated in order to provide better integration of two approaches (with and without the ConflictResolution component). Because of that and also with a purpose to make the deliverable self-contained we will reintroduce the main definitions.

2.1.1 Preliminary Assumptions

In order to provide a formalization of smart card platform and direct control flow verification process we adopted the following assumptions:

- We do not consider packages in the model. Packages are installed at one pass, so we do not consider a possibility for some malicious applet to be installed in one package with an honest one. Consequently we may assume that all applications in the package belong to one provider. Inside a package applications can communicate freely. But since our main goal is to verify security of applications communicating across packages, we may just consider that each package represents one application.

- Security domains are not applications in our model, but entities that provide specific GP services to associated applications. If a security domain itself communicates with other applications through (non-GP-specific) Shareable interfaces mechanism it is necessary to add a corresponding application into the platform model. In fact, security domains provide their GP-specific services (see chapter 1 for details) as Shareable interfaces. But on GP level OPEN ensures that direct access to the services of security domains is (always) granted to (only) associated applications. There is a method `GPSystem.getSecureChannel()` that provides to an application a handle to associated security domain. This is a requirement of GP specification [12]. Besides the `SecureChannel()` interface security domains normally do not participate in the inter-application communication. We can ensure this during the ClaimChecker run on the security domain installation. If an application has a Security Domain privilege (more details are provided in the section 2.7.5), the ClaimChecker needs to check that such application does not have invocations of non-GP-specific services of other applications and all Shareable interfaces it provides extend `SecureChannel`.

- Security policy for inter-application communications is specified only for applications. Security domains are excluded from the security policies due to reasons explained above.

- Obtaining a reference to an object is not an information leakage in our model, but only forbidden service invocation is an information leakage. Though a granularity of information sharing on Java Card is a Shareable interface object (which may comprise several methods) we provide more fine-grained verification mechanism that is based on service invocations.

- On GlobalPlatform an application can register as a Global service provider. The Global services mechanism relies on Java Card level on Shareable interface mechanism, but there is a possibility to obtain a reference to a Global service object by invoking a method `GPSystem.getService()`. In this case OPEN will obtain a desired reference to the object and return it to the client. The assumption for Global services is their availability to any application on the platform. Thus, if an application registers a Global service, it authorizes every other applet on the card to call this service. Since our goal is more fine-grained access control, we do not consider Global services in the model.

- Applications from different stakeholders do not inherit from each other. Inheritance assumes availability of the code during compilation and/or possibility of dynamic

class loading. Though standard Java supports these requirements, on Java Card situation is different. Java Card (current version 2.2 [14, 15, 13]) does not support dynamic class loading. Java Card applications are highly secret and code sharing is not expected. Though (partial) code sharing is necessary for the Shareable services mechanism [9], but this is the only restriction.

- AID and service names are unique.
- Only security domains (and not simple applications) can provide on the platform services of the `SecureChannel` and `SecureChannelX` interfaces from the GlobalPlatform specification [12, App. A: GP for Java Cards].

These assumptions restrict applicability of the model, as they tightly relate it with the Java Card and GlobalPlatform smart cards. This trade-off is necessary due to limitations of smart cards memory.

2.1.2 Stakeholders on the Card

We consider the following parties that together provide the card and its contents: card vendor, card issuer and application providers. We do not consider a card holder as a stakeholder.

Card vendor provides a chip with Java Card and GlobalPlatform libraries installed. Consequently it is a card vendor who provides implementation of the Security-by-Contract framework and incorporates the verification process into the loading mechanism. Card vendor does not provide any specific security policy for the card.

Card issuer is the most privileged stakeholder on the card. Together with the card vendor she decides which approach for the S×C framework to implement (with or without the ConflictResolution component). Her security domain is able to manage any data on the card. Card issuer can provide also her own applications (or other security domains) and security policies regulating access to her services and specifying functionally necessary services.

Application provider installs on the card her applications (and security domains) and specifies related security policies. One platform can host several different application providers. Application providers do not decide which verification process should be implemented on the platform, but they may have business agreements with the card issuer describing a level of security that should be guaranteed on the card.

Platform updates are triggered from outside the card by an entity authenticated at the terminal [12]. We suppose that this entity is one of the application providers or the card issuer. This assumption relies on cryptographic schemes (correctly) implemented by the card and the terminal. GlobalPlatform specification [12] describes important requirements on authentication protocols and cryptography. Thus on a (correctly implemented) smart card updates related to specific application could be requested only by the owners of this application or by the card issuer. It is possible to restrict this requirement and ensure only the application owner can update her application.

2.1.3 Application Update

Specifications of Java Card and GlobalPlatform do not define a convenient mechanism for application updates except deleting an old version and installation of a new one in a transactional fashion. We can propose flexible S×C-based approach for verification

that is based on incremental certification of changes in the code. This approach can be described as follows:

1. Application provider specifies incrementally what is changing in the code (in comparison with the old version);
2. The ClaimChecker verifies that this is specification is correct;
3. The PolicyChecker verifies incrementally that update is secure with respect to the platform security policy $\mathcal{P}$

2.2 Security-by-Contract Framework

Security-by-Contract framework consists of the following components: the ClaimChecker, the PolicyChecker and the ConflictResolution.The ConflictResolution component is optional and it is up to the card vendor and the card issuer to decide whether to implement it of not. Verification without resolution of conflicts can be realized on Java Card without GlobalPlatform.

2.2.1 Duties of the Components

Introduced components are parts of verification process. We have already discussed in D6.3 [11] why the ClaimChecker and the PolicyChecker need to be integrated with the card manager (JCRE on Java Card or OPEN on GlobalPlatform). These components (and also the ConflictResolution component, as will be demonstrated further) require access to installation process and some card manager capabilities. Thus we assume that OPEN (or JCRE) orchestrates verification process.

Upon receiving an update request the card manager, among all, performs the next actions:

- The card manager analyzes the update request and identifies which verification steps should be taken.
- For installation of new application or an update of an existing application the card manager requests the ClaimChecker to verify compliance of the application code and its Contract.
- For installation, update or removal of an application the card manager requests the PolicyChecker to verify if the platform will be secure after an update.
- If the PolicyChecker has rejected an update and the ConflictResolution component is implemented, the card manager requests the ConflictResolution component to evaluate the possibility of update.
- If the PolicyChecker approved the update or the ConflictResolution approved the update the card manager performs the update.
- The card manager maintains platform data: security policy $\mathcal{P}$, list of unselectable applications [1].

[1] List of unselectable applications is introduced further.

Duties of the ClaimChecker component

The ClaimChecker component is responsible for verification that the code of new (or updated) application is compliant with its Contract, more specifically: with its Claim on provided and called services. The ClaimChecker can also evaluate if an application with declared security domain privilege does not provide any services besides `SecureChannel`. This component is specified in the section 2.6.

Duties of the PolicyChecker component

The PolicyChecker component verifies that after an update (installation, update or removal of an applet) the evolved platform will be secure. In order to do this for new applet the PolicyChecker checks if the application Contract is compliant with the platform security policy $\mathcal{P}$. For deletion of an applet the PolicyChecker verifies if after removal of an applet the platform will be secure. This component was partially introduced in the deliverable D6.3 [11, Chap.3]. Full specification of this component is provided in the section 2.5.

Duties of the ConflictResolution component

The ConflictResolution component evaluates a possibility of update reasoning on a hierarchy of applications. The main idea behind it is the hierarchical organization of the security domains and capabilities that some security domains have on the card. On Java Card all applications are treated as equal and no application can request deletion or locking other applets. But on GlobalPlatform some applications (for example Issuer's Security Domain or applications with some specific privileges) have possibilities to manage files and applications of other stakeholders. Thus, for example, a "powerful" security domain can use GP commands and delete some applications that hinder its own update (in case the PolicyChecker rejects an update).

Discussion on the security domains hierarchy and the ConflictResolution component specification are provided further in the section 2.7.

2.3 Notations and Main Definitions

In the remainder of the chapter we will use the following notations:

- $\mathcal{D}$ – set of security domains on the card;
- $\Delta_{\mathcal{D}}$ – a domain of security domains names;
- $\mathcal{A}$ – set of applications installed on the card;
- $\Delta_{\mathcal{A}}$ – a domain of application names;
- $\mathcal{S}$ – set of services provided on the card;
- $\Delta_{\mathcal{S}}$ – a domain of service names;

We will usually 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 (identified by its name) of application A.

2.3.1 Platform Specification

First we specify platform model for a simple workflow that does not consider conflicts. This model was initially introduced in the deliverable D6.3 [11] so we do not discuss it a lot in this deliverable.

Definition 2.3.1 (Platform) *Platform* Θ *is a tuple* $\langle \Delta_{\mathcal{A}}, \Delta_{\mathcal{S}}, \mathcal{A}, \mathsf{shareable}(), \mathsf{invoke}(), \mathsf{sec.rules}(), \mathsf{func.rules}() \rangle$*, where* $\mathsf{shareable}(), \mathsf{invoke}() : \Delta_{\mathcal{A}} \rightarrow \wp(\Delta_{\mathcal{S}})$ *are association functions for the services on the platform;* $\mathsf{sec.rules}() : \Delta_{\mathcal{A}} \times \Delta_{\mathcal{S}} \rightarrow \wp(\Delta_{\mathcal{A}})$ *and* $\mathsf{func.rules}() : \Delta_{\mathcal{A}} \rightarrow \wp(\Delta_{\mathcal{S}})$ *define security policies of the applications.*

The set of all services provided on the platform is denoted as $\mathcal{S} = \bigcup_{A \in \mathcal{A}} \mathsf{shareable_A}$.

For every application $A \in \mathcal{A}$ a function $\mathsf{sec.rules(A, s)} = \mathsf{sec.rules_A(s)}$ is a mapping, that defines for each service s of application A which other applications on the platform are authorized to call it. We will sometimes use notations $\mathsf{sec.rules_A}(\emptyset)=\Delta_{\mathcal{A}}$ and $\mathsf{sec.rules_A}(S')=\{A_{k_1}, \ldots A_{k_m}\}$, where $S' \subseteq \mathcal{S}$ and $A_{k_1}, \ldots A_{k_m} \in \mathcal{A}$.

Function $\mathsf{func.rules(A)} = \mathsf{func.rules_A}$ specifies the set of services on the platform that A needs in order to be functional.

2.3.2 Specification of Contract

The contract in the S×C approach is a specification of an application that describes formally all its (security-related) behaviors.

Definition 2.3.2 (Contract) *For application* A *its* $\mathsf{Contract_A}$ *is a pair* $\langle \mathsf{Claim_A}, \mathsf{AppPolicy_A} \rangle$*, where*

- $\mathsf{Claim_A} = \langle \mathsf{Provides_A}, \mathsf{Calls_A} \rangle$*, and* $\mathsf{Provides_A}, \mathsf{Calls_A} \in \wp(\mathcal{S})$*;*
- $\mathsf{AppPolicy_A} = \langle \mathsf{sec.rules_A}, \mathsf{func.rules_A} \rangle$*, and* $\mathsf{sec.rules_A} : \mathcal{S} \rightarrow \wp(\mathsf{A})$*,* $\mathsf{func.rules_A} \in \wp(\mathcal{S})$*.*

We assume dom($\mathsf{sec.rules_A}$*)* $\subseteq \mathsf{Provides_A}$ *and* $\mathsf{func.rules_A} \subseteq \mathsf{Calls_A}$*.*

$\mathsf{Provides_A}$ is a declared set of services that application A has and $\mathsf{Calls_A}$ is a declared set of services of other applets that A calls. The security and functional rules are instantiated in the AppPolicy of each application.

Definition 2.3.3 (Platform Security Policy) *Security policy of the platform* $\mathcal{P}$ *consists of the contracts of all the applications* $\mathcal{A}$ *on the platform:*
$\mathcal{P} = \{\mathsf{Contract_{A_1}}, \ldots \mathsf{Contract_{A_n}}\}_{\mathsf{for} \bigcup_{i=1..n} \mathsf{A_i} = \mathcal{A}}$

This definition of security policy does not include security domains. This is justified by our assumption that security domains do not provide services (except the GP-specific ones). Also such policy definition is justified by the business case, because the application providers sometimes want to forbid any communications even for applications associated with the same security domain.

2.4 Specification of Possible Updates

Java Card and GlobalPlatform support only installation and removal of an application. But an update of already installed application usually is performed in a transactional fashion (because otherwise some other applications on the card can be broken or left in the inconsistent state).

Below is the list of possible atomic updates (related to applications and specified as changes in the Contract). Installation, removal and update of applications can be presented as a sequence of such atomic updates. We also discuss for each type of atomic update if it can cause problems for other applications (lead to a conflict of interests).

Definition 2.4.1 (Atomic Application Updates)

1. *Addition of a service s to* Provides_B. *It might happen that $s \in \text{Calls}_\text{A}$ for some application $A \in \mathcal{A}$. In this case it is necessary that corresponding authorization (s, A) is present in* $\text{sec.rules}_\text{B}$. *Otherwise it is not secure to perform such update and the* PolicyChecker *will reject it. In this case we may say that application A is preventing addition of new service of B.*

2. *Removal of a service s from* Provides_B. *Some application $A \in \mathcal{A}$ may have this service s in* $\text{func.rules}_\text{A}$. *Consequently, to keep some functionality available s cannot be removed. We may say that application A prevents removal of a service of B.*

3. *Addition of a service $A.s$ to* Calls_B. *In this case the biggest concern is whether B is authorized by A to call its service:* $(s, B) \in \text{sec.rules}_\text{A}$. *If it is not the case, update is breaking policy of application A and cannot be performed.*

4. *Removal of a service s from* Calls_B. *This update is always safe.*

5. *Addition of an authorization (s, A) to* $\text{sec.rules}_\text{B}$. *Such update also does not bring any threats on the platform*

6. *Removal of an authorization (s, A) from* $\text{sec.rules}_\text{B}(\text{s})$. *If $s \in \text{Calls}_\text{A} \cap \text{Provides}_\text{B}$, then update is dangerous, as it will break a policy of application B. So application A objects for such a security-threatening update of B (even if it is policy of B that will be broken).*

7. *Addition of a service $A.s$ to* $\text{func.rules}_\text{B}$. *In this case application B may not be functional if application A does not provide this service s: $s \notin \text{Provides}_\text{A}$. But we cannot say that applications A and B are conflicting, because nothing can force A to install a new service on the platform and B's life will not be improved even if A will be completely removed. Thus the* PolicyChecker *will reject this update, but the* ConflictResolution *component will not be invoked.*

8. *Removal of a service s from* $\text{func.rules}_\text{B}$. *Such update does not bring any problems on the platform.*

Note that in discussion on updates above we have specified possible problems with security or functionality of the platform and also have shown how problematic updates create conflicts between applications. Further we provide a motivating example.

Example 1 *Applications EMV, $ePurse$ and $jTicket$ from the POPS case study example [11, P. 31] are installed on the card. Application $ePurse$ would like to withdraw its service*

$payment$ from the platform. As $jTicket$ relies on $payment$ service in its functionality, it would like $payment$ to remain available. Such setting forbids removal of the $payment$ service. In this situation $ePurse$ ($Bank$ stakeholder) and $jTicket$ ($Transport$ stakeholder) have conflict of interests.

2.5 The PolicyChecker Specification

We start with full specification of the (optimized) PolicyChecker component. First, a definition 3.2.2 of the evolved platform from the deliverable D6.3 [11, P.32] needs to be updated in order to consider new (decremental) types of changes. Below we provide full definition of the evolved platform (including evolutions already specified in D6.3):

Definition 2.5.1 (Evolved Platform) *Let B be an application, an evolved platform Θ' for B 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. *Removal of an installed application $B \in \mathcal{A}$: $\mathcal{A}' = \mathcal{A} \setminus \{B\}$;*
3. *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* shareable_B*:* $\text{shareable}'_A = \text{shareable}_A$ *and* $\text{shareable}'_B = \text{shareable}_B \cup \{s\}$;
   (b) *Removal of a service s from* shareable_B*:* $\text{shareable}'_A = \text{shareable}_A$ *and* $\text{shareable}'_B = \text{shareable}_B \setminus \{s\}$;
   (c) *Addition of a service s to* invoke_B*:* $\text{invoke}'_A = \text{invoke}_A$ *and* $\text{invoke}'_B = \text{invoke}_B \cup \{s\}$;
   (d) *Removal of a service s from* invoke_B*:* $\text{invoke}'_A = \text{invoke}_A$ *and* $\text{invoke}'_B = \text{invoke}_B \setminus \{s\}$;
   (e) *Addition of an access authorization to service s for application C by B:* $\text{sec.rules}'_A() = \text{sec.rules}_A()$ *and* $\text{sec.rules}'_B() = \text{sec.rules}_B() \cup \{(s, C)\}$;
   (f) *Removal of an access authorization (s, C) by B:* $\text{sec.rules}'_A() = \text{sec.rules}_A()$ *and* $\text{sec.rules}'_B() = \text{sec.rules}_B() \setminus \{(s, C)\}$;
   (g) *Addition of a necessary service s to* func.rules_B*:* $\text{func.rules}'_A = \text{func.rules}_A$ *and* $\text{func.rules}'_B = \text{func.rules}_B \cup \{s\}$;
   (h) *Removal of a necessary service s from* func.rules_B*:* $\text{func.rules}'_A = \text{func.rules}_A$ *and* $\text{func.rules}'_B = \text{func.rules}_B \setminus \{s\}$;

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

We stress that in our model locking and unlocking applications is the same as removing and reinstalling them again (correspondingly).

In the deliverable D6.3 [11] the PolicyChecker was introduced to deal with incremental types of changes. Now we provide a full definition of the PolicyChecker that runs also on removal of an application or parts of it.

Definition 2.5.2 (Optimized Contract-Policy Compliance Checker (Policy Checker)) *An Optimized* PolicyChecker *(or just* PolicyChecker*) is an algorithm for certification of changes 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. *Removal of already installed application $B \in \mathcal{A}$.*
   - $\text{Provides}_\text{B} \cap \{ \bigcup_{\text{A}\in\mathcal{A}} \text{func.rules}_\text{A}\} = \emptyset$;.
3. *An update of some application $B \in \mathcal{A}$.*
   (a) *Addition of a service s to* Provides_B*:* $A \in \text{sec.rules}_\text{B}(\text{s} \cap \text{Calls}_\text{A})$*;*
   (b) *Removal of a service s from* Provides_B*:* $s \notin \bigcup_{A\in\mathcal{A}} \text{func.rules}_\text{A}$*;*
   (c) *Addition of a service s to* Calls_B*:* $B \in \text{sec.rules}_\text{A}(\text{Provides}_\text{A} \cap \text{s})$*;*
   (d) *Removal of a service s from* Calls_B*: return true;*
   (e) *Addition of an authorization rule for some application C to access a service s of B to* $\text{sec.rules}_\text{B}$*: return true;*
   (f) *Removal of an authorization rule for some application C to access a service s of B from* $\text{sec.rules}_\text{B}$*:* $s \notin \text{Calls}_\text{C}$ *OR* $s \notin \text{Provides}_\text{B}$*;*
   (g) *Addition of a service s to* $\text{func.rules}_\text{B}$*:* $s \in \bigcup_{A\in\mathcal{A}} \text{Provides}_\text{A}$
   (h) *Removal of a service s from* $\text{func.rules}_\text{B}$*: return true;*

The PolicyChecker rejects all updates if they do not comply with these checks. We do not provide algorithms for the PolicyChecker as their implementation is straight-forward from the definition 2.5.2.

2.6 The ClaimCheckerSpecification

We have presented an idea of the ClaimChecker component in the deliverable D6.3 [11]. Below is the definition 3.3.3 from [11].

Definition 2.6.1 (Claim Checker) *A* ClaimChecker *algorithm for a new (updated) application B is an 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* invoke_B=Calls_B.

In this deliverable we present actual specification of the precise ClaimChecker component.

Intuitively, the ClaimChecker has to check that application code and Claim are compliant: all services that B has are declared in $\mathsf{Provides}_B$ and all services that B invokes are declared in Calls_B. If we want to capture functional dependencies we have to require the ClaimChecker to be precise. If, for instance, there is a service $B.s$ that applet B has declared, but does not actually provide, further another applet A, which may need to use that service, can stuck.

The ClaimChecker defined by the current specification compares services just by their names. On Java Card the client needs to import the export file of the server and to know the details of the server implementation beforehand [14]. Consequently, we assume that equal service names guarantee that these are indeed the same services.

We will discuss separately how the ClaimChecker can verify the declared set of provided services (server role) and declared set of called services (client role). When an applet is being installed (or updated) the ClaimChecker will check it for being a honest server and a honest client. Other checks that the ClaimChecker can provide are verification of a security domain code and verification that indeed $\mathsf{func.rules}_B \subseteq \mathsf{Calls}_B$.

The ClaimChecker can run on or off the card. On card implementation is more secure, but also more tricky, as it requires an access to the installer. Because of these reasons now we provide details of the ClaimChecker which can run on the JCA files that are something intermediate between standard class files and CAP files. JCA file contains bytecode opcodes and has a structure similar to the standard class files, but linking (substitution of the methods and classes names by some encoding, as specified in [15]) is implemented similarly to the CAP files and export files. We will also use information from the export files that are available both on card and off card.

We note that currently we provide sketches for the ClaimChecker algorithms. This is due to the fact that on the moment of writing this deliverable the choice between on-card and off-card implementation has not yet been made (Task 6.5 starts on M24). Thus we have chosen to just outline possibilities of bytecode analysis. In fact, undoubtedly, the ClaimChecker specification needs to be finalized with soundness guarantees, which can be provided with a help of abstract interpretation techniques. We are going to provide these guarantees during the 3rd year of the project.

2.6.1 Verification for Server

Two levels of precision are possible for server code verification. The ClaimChecker can look at the declared public interfaces with `superinterface Shareable`. These are declared interfaces that an applet might implement. However, the applet may never actually return a declared interface object and never declare any class that implements this interface.

As specified in [15, Sec. 5] the export file contains all publicly available interfaces and classes, including interfaces and classes that implement `Shareable`. These classes and interfaces are labeled with special token (a number) in the export file. Looking into export file in order to get all interfaces and their methods names can be a strategy for the ClaimChecker for server code verification.

JCA verification

In the JCA file in order to capture all interfaces that implement `Shareable`, one needs to look for the keyword `.shareable` in the public interface descriptions. Methods of

interfaces with this keyword are the services of application that we are looking for.

The verification discussed above is an over-approximation for the server. To be more precise the ClaimChecker needs to check that the server actually can return an instance of a class implementing shareable interface (for each interface). This can be done by looking at classes which have a label `shareable`. In such classes field `implementedInterfaceInfoTable` provides details about interfaces implemented by these classes. Shareable interface is listed here and also the interfaces defined by the server that implement `shareable`. Their tokens can be matched with the actual interfaces declared in the interfaces field.

Having found for each interface a class implementing it, it is necessary to check that the class has a method `getShareableinterfaceobject()` implemented and it will return references to the corresponding interfaces. For each interface the server wants to share we need to find at least one class that implements this interface and returns a reference to this interface (bytecode sequence: `getfield_a_this X; areturn`, where X is a token of the interface).

2.6.2 Verification for Client

In order to find an over-approximated set of services that may be called by the client, we first look at the export files that client imports and the imported public interfaces it declares. Indeed, the client is required to provide these details for successful linking.

JCA verification

To obtain a list of interfaces which the client can receive we first search for the token of the method `staticMethodRef ⟨...⟩ Ljavacard/framework/Shareable` in the `constantPool`. In order to find all the client calls to the method `getAppletShareableInterfaceObject()` of JCRE, which may return a Shareable interface object the ClaimChecker will look in the bytecode for the instruction `invokestatic X`, where X is the previously found token.

In order to understand which reference the client has received the ClaimChecker will look at the next instruction: `checkcast 0 Y`. 0 in this instruction means that the object is checked against a class or an interface. Y defines a number of the token in the `constantPool`. This token appears in the `constantPool` with a keyword `classref W` where W=V.Z, and V is a number of one of the imported classes, Z is a token for an interface or class in V (it can be found in the export file of V).

We can capture the exact list of services invoked by the client if the ClaimChecker will look for `invokeinterface X Y Z` instructions. `invokeinterface` is the only bytecode instruction that allows context switch and can be used for invocation of the Shareable interface methods [14, Sec. 6.2.8.6]. X is a number of arguments (plus 1), Y is a token of the interface to be invoked and Z is a token of the method of the interface to be invoked. Y has been identified by the ClaimChecker before and Z can be found from the invoked class object table.

2.6.3 The Claim Checker Algorithms

In this section we sketch the ClaimChecker algorithms. For the algorithm that can verify the server code we have chosen an over-approximation algorithm 2.6.1 that just ensures that an application actually has all the services it declared in Provides set.

Require: Provides_B, code of JCA file of an applet B
Ensure: Provides_B is compliant with B's code.
1: $W \leftarrow \emptyset$
2: **for** All interfaces I with keyword `.shareable` **do**
3: **for** all methods m declared in I **do**
4: $W \leftarrow W \cup m$
5: **if** $W = \text{Provides}_\text{B}$ **then**
6: **return** TRUE
7: **else**
8: **return** FALSE

Algorithm 2.6.1: Claim Checker for the Server

For the client code verification the ClaimChecker can follow the algorithm 2.6.2 sketch. The main steps include finding in the export files and the `constantPool` of the client tokens of the shareable interfaces that the client obtains from other applications. Then the ClaimChecker finds a subset of these tokens which appear with the instruction `invokeinterface` and obtains the desired set of methods, which is matched with the set Calls_B. The ClaimChecker for the client on the last step will also check if declared functional needs are honest: $\text{func.rules}_\text{B} \subseteq \text{Calls}_\text{B}$.

The ClaimChecker for the client needs to capture only invocations of services that are not GP-specific methods of security domains. Methods of the SecureChannel and SecureChannelX interfaces [12, App. A] need to be excluded from the Calls set [2]. In the export file there are no details if the applet is a security domain or not. Thus we can only assume that only security domains (and not simple applications) can provide on the platform services of the `SecureChannel` and `SecureChannelX` interfaces.

Require: Calls_B, $\text{func.rules}_\text{B}$, code of JCA file of an applet B, export files received with B.
Ensure: Calls_B is compliant with B's code.
1: $W \leftarrow \emptyset$
2: $V \leftarrow \emptyset$
3: **for** All entries in the `constantPool` with keyword `classref X.Y` **do**
4: Find in the export file of X an object corresponding to Y;
5: **if** Y corresponds to the interface `SecureChannel` or `SecureChannelX` **then**
6: skip
7: **else**
8: $V \leftarrow V \cup (I, X, Y)$, where I is a number of the entry;
9: **for** All instructions `invokeinterface X Y Z` **do**
10: **if** Exists an element $(Y, C, Z) \in V$ **then**
11: Find in an export file for C a service name s for token Z
12: $W \leftarrow W \cup s$
13: **if** $W = \text{Calls}_\text{B}$ AND $\text{func.rules}_\text{B} \subseteq$ W **then**
14: **return** TRUE
15: **else**
16: **return** FALSE

Algorithm 2.6.2: Claim Checker for the Client

[2] In fact, there are other GP-specific services invocations that can appear in the code of the client, but need to be excluded from Calls set: Global Registry service requests, CVM services requests, *etc.*

We note that if an algorithm fails at some step because it cannot find a necessary export file or an entry, then the installation process will fail itself and results of the ClaimChecker verification will not be considered.

Algorithms above specify a precise ClaimChecker.

The ClaimChecker for New Security Domain

As security domains are not supposed to provide services except the ones implemented for GlobalPlatform-defined purposes and to call any services, besides such GP services of higher domains, it is necessary to provide an algorithm specifying the ClaimChecker for a security domain. We will just sketch this algorithm as its details follow from the ClaimChecker algorithms for server and for client.

1. Ensure that loaded application is indeed a security domain by verifying corresponding privileges [3].
2. Using ideas of the ClaimChecker for the server algorithm 2.6.1 find in the JCA file of the security domain set W of interfaces extending `GPSystem.SecureChannel` or `GPSystem.SecureChannelX` interfaces.
3. Check that no other shareable interfaces besides the ones in W are declared. If it is not the case, return FALSE.
4. Check that the only methods of interfaces in W are the ones declared in the GP specification [12, P. 198] and [12, P. 208]. If it is not the case, return FALSE.
5. Using ideas of the ClaimChecker for the client algorithm 2.6.2 find the set V of called services. Ensure that V contains only services declared in the GP specification [12, P. 198] and [12, P. 208] for `SecureChannel` and `SecureChannelX` interfaces. If it is not the case, return FALSE .
6. Return TRUE.

2.6.4 The ClaimChecker for Evolution

The definition 2.5.1 of platform evolution is specified for the actual code of applications (sets shareable and invoke), but the PolicyChecker works only on Contract and security policy $\mathcal{P}$. Thus it is necessary to match the actual code and the declarations in the Contract. This is exactly the ClaimChecker responsibility. For the evolution that represents installation of a new applet the algorithms are presented in the section 2.6.3 (by definition of the precise ClaimChecker). But for the atomic updates the algorithms for the server and client need to be adjusted correspondingly. We provide just general ideas for the updated algorithms. Their implementation is straight-forward and similar to the algorithms of the ClaimChecker listed in the section 2.6.3.

- Server-side verification needed: Addition/removal of a service s to/from shareable_B. In case of addition service s is declared as added to the set Provides_B in the Contract_B and the goal of the ClaimChecker is to verify that indeed this service was added. In order to do that the ClaimChecker looks in the JCA file of B for all interfaces with the keyword `.shareable` and within them tries to find the service s. If the ClaimChecker

[3] See the section 2.7.5 for details.

succeeds, then it returns TRUE, otherwise it returns FALSE. Analogously, inverted results of this check can be used for assurance that the service s has been removed from the set $\mathsf{Provides}_\mathsf{B}$.

- Client-side verification needed: Addition/removal of a service $A.s$ to/from $\mathsf{invoke}_\mathsf{B}$. In case of addition service $A.s$ is declared to be added to the set $\mathsf{Calls}_\mathsf{B}$ and the ClaimChecker is required to guarantee that indeed B can potentially invoke a service s of application A. The idea for the ClaimChecker in this case is that it tries to find an instruction `invokeinterface X Y Z` in the JCA file of B where Y is a token from the `constantPool` that corresponds to an imported interface of A and Z is a token for the service s in the export file of A. If it succeeds to find such command, then the ClaimChecker returns TRUE, otherwise it returns FALSE. It might be the case that updated applet B does not import an export file of A and does not have tokens related to A in the `constantPool`. In this case the ClaimChecker returns FALSE. These checks can be inverted to verify truthful declaration of removal of a service s from $\mathsf{invoke}_\mathsf{B}$.

2.7 Conflict Resolution Component

Business case for smart card domain is usually more demanding than simple rejecting a dangerous update. In the sections 1.1.4 and 1.1.5 we have explained the hierarchical organization of the card content and the set of privileges available on the card. These properties of smart cards with GP allow more refined reasoning about updates.

Security domains are representatives of the stakeholders on the card. A stakeholder can provide one or several security domains. Usually each stakeholder associates her main security domain with itself, thus making it a root in the forest of the security domains hierarchy.

Our main goal is to keep platform secure after every update. If some update is security-threatening, we always have a possibility to reject it. But for satisfaction of industry needs we have to enable the framework with other possible options. We might want to accept certain kinds of updates, always making sure that the platform after an update is again in a secure state. This can be performed by making unselectable other conflicting applications already installed on the card.

Smart card platform can prioritize applets on the card depending on which stakeholder they belong to and which privileges this stakeholder has. For example, card issuer is always the most privileged stakeholder. This fact is also reflected in the ability of the card issuer to assign GP privileges on the card, privilege to delete any card content among them. The issuer's security domain can delete or lock any application A on the card if it prevents issuer's applet B_{ISD} from installation. We have already discussed in the section 2.4 how various conflicts can arise from atomic updates of applications. Now we will demonstrate how such conflicts can be resolved in the presence of the stakeholders hierarchy.

2.7.1 Addition to the formal model

Previously we have presented the formal model of the smart card platform and the ClaimChecker and the PolicyChecker specifications that were built on top of that model. For the conflict resolution component ConflictResolution it is necessary to add the GP

privileges to the model. We also need to specify Issuer's Security Domain (ISD) among other security domains.

We denote as $\mathcal{T}$ the set of chosen privileges which are represented as tokens.

1. t_{AM} is an Authorized Management privilege;
2. t_{DM} is a Delegated Management privilege;
3. t_{GL} is a Global Lock privilege;
4. t_{GD} is a Global Delete privilege;

Thus $\mathcal{T}$=$\{t_{\mathsf{AM}}, t_{\mathsf{DM}}, t_{\mathsf{GL}}, t_{\mathsf{GD}}\}$. An element of this set we will also denote as t. We denote security domain SD with privilege t as $SD\langle t\rangle$. For example we may have on the platform an $SD_A\langle t_{\mathsf{GL}}\rangle$ or $SD_B\langle t_{\mathsf{DM}}, t_{\mathsf{GD}}\rangle$. Security domain without privileges is denoted $SD\langle\ \rangle$. If privileges are not important we will continue to use old notation SD instead of $SD\langle t\rangle$ or $SD\langle\ \rangle$.

Assumptions

We suppose that ISD is a dominant domain on the card, as in the GP specification ISD is explicitly listed as a part of the card manager [12, Sec. 1.5.2.8]. Issuer is a stakeholder with the most capabilities. Other stakeholders (application providers) are considered to be equal. As security domains are representatives of the stakeholders, we consider the corresponding security domains to be equal. But GP privileges empower some of security domains with capabilities to manage data and applications of other security domains. Consequently, we suppose that if a security domain SD_1 has a Global Lock privilege (or any other privilege among those specified in the section 2.7.1) then it has a priority over another security domain SD_2 without any privileges.

Security domains on the card are organized into a forest due to extradition process. GP spec allows a security domain SD to manage (locking and removal) other security domains and applications that are associated (directly or indirectly) with SD.

Following are our assumptions:

1. Security domains from the same provider belong to the same tree.
2. The privileges from set $\mathcal{T}$ are equal.
3. We define privileges as belonging to a specific domain SD and not to all domains that SD is associated with. Another option is to consider that a security domain can use all the privileges of its subdomains. We can easily implement this option in our model – it is enough to propagate the privileges of a security domain SD to all the domains SD is associated with directly or indirectly.

For security and functionality verification process (see the section 2.8 for more details) of the PolicyChecker changes in the security domains hierarchy are not relevant. But these changes influence the ConflictResolution component's process and so we need to consider them now. As the idea of conflict resolution is to make conflicting applications (that were lower in the hierarchy) unselectable, another important update to the platform model Θ is a set of unselectable applications that is maintained by the S×C framework. After each new update applets from this set are checked to be made selectable again.

We use notation $SD_1 \geq SD_2$ if security domain SD_2 is associated with security domain SD_1 directly or indirectly. Association relation $\geq$ is not a partial order, because not all security domains are associated with themselves (so it is not reflexive). But this relation is transitive and antisymmetric. Our another assumption is on association relation $\propto$ that matches each application with corresponding security domain. We consider that each application $A \in \mathcal{A}$ is associated with one security domain $SD \in \mathcal{D}$.

In order for the platform model to be consistent with already defined components we propose a definition of extended platform. This definition can be used by all components of S×C framework.

Definition 2.7.1 (Extended Platform) *Extended platform Θ_{ext} is a tuple $\langle \Delta_{\mathcal{D}}, \Delta_{\mathcal{A}}, \Delta_{\mathcal{S}}, \mathcal{D}, \mathcal{A}, \mathsf{shareable}(), \mathsf{invoke}(), \mathsf{sec.rules}(), \mathsf{func.rules}(), \propto, \geq, \mathsf{priv.domains}, \mathsf{Unselectable} \rangle$, where relation $\propto \subseteq \mathcal{A} \cup \mathsf{Unselectable} \times \mathcal{D}$ is a relation of association between application and security domain; relation $\geq \subseteq \mathcal{D} \times \mathcal{D}$ is transitive and antisymmetric relation for association between security domains; $priv.domains \subseteq \mathcal{D} \times \mathcal{T}$ is a relation for privileges assignment; $\mathsf{Unselectable} \subseteq \Delta_{\mathcal{A}} \backslash \mathcal{A}$ is a set of currently unselectable applications.*

Security policy for the extended platform is defined as in the definition 2.3.3. Again we do not consider security domains and corresponding relations as relevant for the security policy. It is necessary also to stress that Unselectable is a set of unselectable applications that were locked due to the ConflictResolution component reasoning. There could be more unselectable applications on the platform, that were made unselectable because of requests of their providers or due to other reasons, but we do not consider them in the model. In our model such unselectable applications (not members of Unselectable set) are considered being removed.

$ISD \in \mathcal{D}$ is the Issuer's Security Domain. ISD is always installed during card issuance and is normally a native application rather than Java one. It is not possible to associate ISD with any other domain except itself. We assume that ISD has all possible privileges on the platform.

Example 2 *As specified in the deliverable D1.1 [5], in the POPS case study Mobile Operator is a card issuer and an owner of the Issuer's Security Domain (ISD). ISD performs installation of SD_{Bank} security domain and initially SD_{Bank} is associated with ISD: $ISD \geq SD_{Bank}$. Then according to the agreement SD_{Bank} is extradited and associated with itself, becoming a root in its own tree of domains: $SD_{Bank} \geq SD_{Bank}$.*

2.7.2 Security Domains Hierarchy Revisited

We will use notation $SD_1 \triangleright SD_2$ to define the fact that security domain SD_1 is able to manage security domain SD_2 in the domains hierarchy or by specific privileges. If neither of SD_1 and SD_2 is able to manage each other, then these security domains are incomparable. Security domains association relation $\geq$ is a part of the relation $\triangleright \subseteq \mathcal{D} \times \mathcal{D}$.

Definition 2.7.2 *A pair $(SD_X, SD_Y) \in \triangleright$ if and only if one of the following conditions holds:*

- SD_X = ISD;
- $SD_X = SD_Y$;
- $(SD_X, SD_Y) \in \geq$;

- $(SD_X, t) \in priv.domains$ *AND for all* $q \in \mathcal{T}$ $(SD_Y, q) \notin priv.domains$.

Managing relation $\triangleright$ is not transitive. Informally, this relation specifies managing possibilities of specific security domain with respect to another security domain. If $SD_A \triangleright SD_B$, then SD_A is able to perform certain manipulations with applications of SD_B using GP operations. More precisely, SD_A is able to remove/lock SD_B or any of its applications. Consequently, if SD_B is the domain that prevents an update of SD_A then after being rejected by the verifier SD_A may try to remove/lock SD_B or its applications. This will be considered as another update by the verifier. If nobody relies on SD_B this update will be authorized. Then SD_A will freely perform the previously desired update.

We note that in the scenario described above we still assume absence of a possibility to bypass the verification process. GP operations which provide some managing capabilities are in full compliance with this assumption.

2.7.3 Extended Platform Evolution

For extended platform Θ_{ext} we define new possible types of evolution (besides the ones that were defined for Θ and are still applicable). Again we specify just evolved components of the platform.

Definition 2.7.3 (Updates Specified only for Extended Platform) *1. Installation of a security domain SD with a set of privileges $\mathcal{T}_{SD} \subseteq \mathcal{T}$ and association of SD with some domain SD_1: $\mathcal{D}_{new}$ = $\mathcal{D} \cup SD$, $\geq_{new}$= $\geq \cup (SD_1, SD) \cup (SD_k, SD)$ for all SD_k such that $(SD_k, SD_1) \in \geq$, $priv.domains_{new}$=$priv.domains \cup (SD, t)$ for all $t \in \mathcal{T}_{SD}$.*

2. *Removal of a security domain SD associated with SD_1: $\mathcal{D}_{new}$ = $\mathcal{D} \backslash SD$, $\geq_{new}$= $\geq \backslash \{(SD_1, SD), (SD_k, SD)\}$ for all SD_k such that $(SD_k, SD_1) \in \geq$, $priv.domains_{new}$ = $priv.domains \backslash (SD, t)$ for all $(SD, t) \in priv.domains$. We allow such update only when applications associated with the domain SD have already been removed.*

3. *Extradition of a security domain SD from a domain SD_1 and association of it with a domain SD_2: $\geq_{new} = \{\geq \backslash \{(SD_1, SD), (SD_k, SD_q), (SD_k, SD), (SD_1, SD_q)\}\} \cup (SD_2, SD) \cup (SD_m, SD_q) \cup (SD_m, SD) \cup (SD_2, SD_q)$ for all SD_k such that $(SD_k, SD_1) \in \geq$, all SD_q such that $(SD, SD_q) \in \geq$ and all SD_m such that $(SD_m, SD_2) \in \geq$.*

4. *Assignment of a privilege t to SD: $priv.domains_{new}$ = $priv.domains \cup (SD, t)$ where $t \in \mathcal{T}$.*

5. *Removal of a privilege t from SD: $priv.domains_{new}$ = $priv.domains \backslash (SD, t)$ where $t \in \mathcal{T}$.*

6. *Extradition of application $A \in \mathcal{A} \cup$ Unselectable from security domain SD_1 to security domain $SD_3 2$: $\propto_{new} = \{\propto \backslash (A, SD_1)\} \cup (A, SD_2)$.*

It is easy to show that updates specified in the definition 2.7.3 cannot break security or functionality on contract level on the extended platform. Definitions of security and functionality on contract level can be found in [11] or further in this deliverable in the section 2.8.

We note that application updates specified for platform Θ are applicable to the extended platform Θ_{ext} with minor modifications: for example, application installation (removal) adds (removes) corresponding element to (from) the $\propto$ relation.

Update to the managing relation $\triangleright$ is straightforward. Since security policy $\mathcal{P}$ does not include security domains hierarchy these updates are not verified by the PolicyChecker.

Internal Updates

We want to stress that before we only considered evolutions on the platform that were triggered externally by one of the stakeholders. Instead with applications from the set Unselectable the process of making one of the applets unselectable is internally triggered by the S×C framework.

2.7.4 Conflict Resolution Algorithm

We denote an application that requested an update as B and its security domain as SD_B. We denote as $C_1, \ldots, C_k$ applications that prevent an update to be executed, because the update is conflicting with their interests. Correspondingly we denote as $SD_{C_1}, \ldots, SD_{C_k}$ their security domains.

Conflict resolution component's purpose is to reason about possibility of update taking into account the managing relation $\triangleright$ specified above. Update is allowed if the requester SD_B is able to manage all security domains SD_{C_i}, $i = 1..k$. Conflict resolution component must also take into account other domains, that might be affected if applications $C_1, \ldots C_k$ become unselectable. Consequently, for any application C_k that needs to be made unselectable the ConflictResolution component must run the reasoning again considering request for deletion of C_k as an update (with SD_B as a requester for this update). The process is cascade-like and repeats a part of the PolicyChecker algorithms, since it has to identify the conflicting applications.

Potentially the PolicyChecker can define conflicting applications $C_1, \ldots C_k$ and trigger the ConflictResolution component. But in order to make the PolicyChecker component consistent with the framework without conflict resolution process we do not modify its implementation. So the ConflictResolution will just receive as input a specification of update.

We will denote an update to application B as $\langle A, B, update \rangle$, where A is an AID of application requesting the update (can be B itself), string $update$ is one of the updates specified in the section 2.4.

The algorithm of the ConflictResolution is based on the following steps:

Input $\langle B, B, update \rangle$, Θ_{ext}, $\triangleright$

Output Accepting/Rejecting update

1. Find the conflicting applications $C_1, ..C_k$ and corresponding security domains $SD_{C_1}, ..SD_{C_k}$.
2. Define whether $SD_B \triangleright SD_{C_i}$ for all $i = 1..k$. If it is not the case, reject the update.
3. For each C_i run the ConflictResolution algorithm with $\langle B, C_i, deletion \rangle$ as an input. If some returns with "Reject", reject update.
4. Otherwise, accept update

The algorithm 2.7.2 of ConflictResolution component is provided below. It uses procedure UpdateQuery 2.7.1 with parameters (application A, query of updates $ConflictQuery$). ConflictResolution component algorithm 2.7.2 is invoked on an atomic update of the type specified in the definition 2.4.1. It can be as well invoked on a sequence of such atomic updates (for example, removal of an application or installation of an application is a sequence of atomic updates). It is necessary to emphasize that addition of a service $A.s$ to $\text{func.rules}_\text{B}$ is not checked by the ConflictResolution component as it is not a conflict (see the section 2.4 about atomic updates for more details). In case of this update the final decision will be taken by the PolicyChecker.

Require: $A \in \mathcal{A}$, $ConflictQuery$
Ensure: Updated $ConflictQuery$
1: **for** all $m \in \text{Provides}_\text{A}$ **do**
2: push $\langle B, A, remove\ m\ from\ \text{Provides}_\text{A}\rangle$ to $ConflictQuery$
3: **for** all $m \in \text{Calls}_\text{A}$ **do**
4: push $\langle B, A, remove\ m\ from\ \text{Calls}_\text{A}\rangle$ to $ConflictQuery$
5: **for** all $(m, D) \in \text{sec.rules}_\text{A}$ **do**
6: push $\langle B, A, remove\ (m, D)\ from\ \text{sec.rules}_\text{A}\rangle$ to $ConflictQuery$
7: **for** all $m \in \text{func.rules}_\text{A}$ **do**
8: push $\langle B, A, remove\ m\ from\ \text{func.rules}_\text{A}\rangle$ to $ConflictQuery$
9: **return** $ConflictQuery$

Procedure 2.7.1: UpdateQuery Procedure

Unselectable Applets Support

Note that ConflAppList is a list of applications that are in a conflict with application B which requested the update. If update is accepted all applications from this list must be set as unselectable. So as internal evolution of the Θ_{ext}: $\text{Unselectable}' = \text{Unselectable} \cup \text{ConflAppList}$. Also platform security policy needs to be updated correspondingly: $\mathcal{P}' = \mathcal{P}\backslash\{\text{Contract}_\text{A}\}$ for all $A \in \text{ConflAppList}$. We also save intermediate value of ConflAppList. Further we prove that updated extended platform Θ_{ext} is secure.

As we already mentioned, after each update (of the types of the Def. 2.4.1 that are relevant for the PolicyChecker) unselectable applications from the set Unselectable have to be revisited in order to check if now they are compliant with the security policy $\mathcal{P}$. Though there is no point to check applets that were marked as conflicting during the last invocation of the ConflictResolution component (current value of the ConflAppList set corresponding to the last update). We provide algorithm 2.7.3 for this check of the Unselectable applets. After its run the extended platform should be updated correspondingly. Obviously for this algorithm order of verification matters. We may choose which logic to implement for maintaining some order on the Unselectable set. For example, we can consider order of unselecting applications, $\triangleright$ relation and so on.

On GP a security domain that is directly or indirectly associated with unselectable (locked in GP terminology) application can request OPEN to unlock it. As well, a security domain with Global Lock privilege can request OPEN to unlock any other application. In order to prevent unlocking of unselectable applications (that were locked after the ConflictResolution component run due to security reasons) it is necessary to add a new check to the verification process performed by OPEN: during application unlocking

Require: $\langle B, B, update \rangle$
Ensure: Accept/Reject Update decision
1: $ConflictQuery = \emptyset$
2: add $\langle B, B, update \rangle$ to $ConflictQuery$
3: ConflAppList $\leftarrow \emptyset$
4: **while** $ConflictQuery \neq \emptyset$ **do**
5: pop first element $\langle B, C, update \rangle$ from $ConflictQuery$
6: find SD_B and SD_C such that $(B, SD_B) \in \propto, (C, SD_C) \in \propto$
7: **if** NOT ($SD_B \triangleright SD_C$) **then**
8: **return** Reject
9: **else**
10: **if** $update$ = add service $A.s$ to Provides_C **then**
11: **for** $all\ A \in \mathcal{A}$ **do**
12: **if** $s \in \text{Calls}_A$ AND $(s, A) \notin \text{sec.rules}_C$ **then**
13: **if** $A \notin$ ConflAppList **then**
14: add A to ConflAppList
15: UpdateQuery(A,$ConflictQuery$)
16: **if** $update$ = remove service s from Provides_C **then**
17: **for** $all\ A \in \mathcal{A}$ **do**
18: **if** $s \in \text{func.rules}_A$ **then**
19: **if** $A \notin$ ConflAppList **then**
20: add A to ConflAppList
21: UpdateQuery(A,$ConflictQuery$)
22: **if** $update$ = add service $A.s$ to Calls_C **then**
23: **if** $s \in \text{Provides}_A$ AND NOT $(s, C) \in \text{sec.rules}_A$ **then**
24: **if** $A \notin$ ConflAppList **then**
25: add A to ConflAppList
26: UpdateQuery(A,$ConflictQuery$)
27: **if** $update$ = remove authorization (s, A) from sec.rules_C **then**
28: **if** $s \in \text{Calls}_A$ AND $s \in \text{Provides}_C$ **then**
29: **if** $A \notin$ ConflAppList **then**
30: add A to ConflAppList
31: UpdateQuery(A,$ConflictQuery$)
32: **return** Accept, ConflAppList

Algorithm 2.7.2: Conflict Resolution Algorithm

process OPEN shall check that application is not in the Unselectable set. This check is similar to the ones already defined for application locking and unlocking [12, Sec. 9.6.2].

2.7.5 Conflict Resolution Implementation Details

In this section we provide a guideline for ConflictResolution component implementation on Java Card and GlobalPlatform smart card. Our first goal is to provide details of application code and installation process (on Java Card and GP) that are required for ConflictResolution implementation.

GP specification defines some modifications to the Java Card implementation in order to ensure compliance of two specifications. We discuss further the details of such implementation that highlight connections between our formal model and algorithms with

Require: Unselectable, $\mathcal{P}$, ConflAppList, Θ_{ext}.
Ensure: $X \subseteq$ Unselectable
1: $X \leftarrow \emptyset$
2: $Platform \leftarrow \Theta_{ext}$
3: $Policy \leftarrow \mathcal{P}$
4: **for** All $B \in$ Unselectable$\backslash$ConflAppList **do**
5: Run PolicyChecker algorithm for new application installation on $Platform$ with B, Contract$_B$ as input
6: **if** PolicyChecker accepts such update **then**
7: add B to X
8: update $Platform$ with B
9: update $Policy$ with Contract$_B$
10: **return** X

Algorithm 2.7.3: Algorithm for Unselectable Applets

the actual smart card realization.

Privileges

When updates 2.7.3 specified for the extended platform Θ_{ext} occur OPEN receives details of the privilege (re-)assignment.

Privileges are implemented on GP as tokens and data about them is stored in the GP Registry. GP Registry on Java Card is implemented as an applet. Registry entries might be obtained though calls to specific `public interface GPRegistryEntry` which extends `javacard.framework.Shareable`. Fields and methods of this interface are specified in GP specification [12, P. 109]. Fields contain data on the privileges assigned to this application. Application A that wants to have an information from the Registry about an entry for the applet B makes a call to OPEN with AID of B. If A has corresponding privilege (Global Registry privilege) or has access right for this entry, OPEN returns to A the desired object.

So one possibility is to implement ConflictResolution component as an applet with Global Registry privilege. Below we specify from which details of the application code (on Java Card and GP) it is possible to obtain privilege assignment $priv.domains$ relation.

1. Security domain installation. We note that security domain is an applet itself, so we will look at the application installation process. [12, sec. A1] Method `install` of class Applet defined for the Java Card applications [13] should be invoked with parameters `bArray, bOffset, bLength`. Parameter `byte [] bArray` contains, among all, data on privileges length and the privilege tokens. OPEN verifies the received parameters. Then applet registers on the platform (and in the GP Registry) with the method `register (byte [] bArray, short bOffset, byte bLength)`. Thus we may obtain an information on the requested privileges from the applet code itself. Or this information can be obtained from the GP Registry. In order to become a security domain an application needs to have corresponding Security Domain privilege.
   For the bytecode verification we are interested in the method `register()` which is a part of `javacard.framework`. `register()` is listed in the `constantPool` section of the applet's JCA file. As we obtain the number `X` of this command we can

find invocation of it in the bytecode (`invokespecial X`, `invokestatic X` or `invokevirtual X`) and obtain details of parameters on the stack.

2. Another interesting detail is obtaining details of privileges reassignment and Registry update. Unfortunately these details are available only at run-time. Privileges reassignment and application extradition are requested by the stakeholder authenticated to the card and are not specified in the application code. Thus we cannot obtain these details from the bytecode. We may try to investigate the `process()` method of the application in order to see which commands an application could receive from outside the card but it will be a huge overapproximation. Consequently we have to rely on availability of the GP Registry data.

Additions to the Conflict Resolution Logic

The managing relation $\triangleright$ and conflict resolution logic can be defined differently. For example, we may assume that security domains association provides to the higher-level domains an ability to use the privileges of their subdomains. Another reasonable addition to the ConflictResolution logic could be to ensure an ability of each security domain to manage its own applications decrementally (to make some service unavailable or to forbid its usage). Card issuer can decide which logic to implement taking into account all the business agreements between stakeholders.

2.8 Validation of the Approach

In the deliverable D6.3 [11] we have presented validation results for the tandem of the ClaimChecker and the PolicyChecker. In this deliverable we provide validation results for the extended platform Θ_{ext} and the ConflictResolution component. Specifically, we want to prove that ConflictResolution process provides sufficient guarantees that Θ_{ext} is secure after an update that was approved by the ConflictResolution component.

We recall that the Theorem 3.4.2 of [11] states that the ClaimChecker and the StaticPolicyChecker together ensure that the platform Θ is secure. The Theorem 3.4.3 of [11] states that in fact the PolicyChecker (which performs optimized checks) is equivalent to the StaticPolicyChecker that runs complete compliance checks for all applications. We refer an interested reader to the deliverable D6.3 [11, Sec. 3.4] for more details. Thus for reasoning about guarantees that the ConflictResolution component delivers it is enough to show that if an update is accepted, then security and functionality on contract level are maintained.

Security and functionality on extended platform Θ_{ext} are defined similarly to security and functionality on platform Θ. In the model Θ_{ext} application communications are performed in the same fashion as in the Θ model. Namely, the applications communicate through usage of services of each other. Security domains association relations do not bring communication means (in assumption that security domains themselves do not organize shareable interfaces except the ones required by the GlobalPlatform specification).

We recall the definition of the StaticPolicyChecker:

Definition 2.8.1 (Static Policy Checker) *A* StaticPolicyChecker *algorithm for platform* Θ *and changed application* B *is an algorithm, that returns true iff for all applications* $A, B \in \mathcal{A}$ *and services* $s \in \mathcal{S}$

- ***Security on contract level**: if $B.s \in \text{Calls}_\text{A}$ and $s \in \text{Provides}_\text{B}$ 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.

Theorem 2.8.2 *If extended platform Θ_{ext} is secure and the* ConflictResolution *component is sound and accepted an update, and the applications specified by the* ConflictResolution *component in the* ConflAppList *set were made unselectable, then updated extended platform Θ_{ext}' maintains security and functionality on contract level.*

Proof. It is easy to show that updates specified in the definition 2.7.3 do not threat security or functionality of the extended platform (definition 2.8.1). Only updates specified for applications in the definition 2.4.1 can be security- or functionality-threatening.
Let an update of application B of one of the types defined in 2.4.1 has been performed. Assume security on contract level is broken after this update: for some application $C \in \mathcal{A}$ exists an application $D \in \mathcal{A}$ such that $C.s \in \text{Calls}_\text{D}$ and $s \in \text{Provides}_\text{C}$, but $(s, D) \notin \text{sec.rules}_\text{C}$. This means that C was not identified as a conflicting application in the ConflictResolution algorithm. Before update the platform was secure, consequently either $C.s$ was not in the set Calls_D, or $s \notin \text{Provides}_\text{C}$, or $(s, D) \in \text{sec.rules}_\text{C}$ and the change was a consequence of the conflict resolution process.
We reason by the type of update. The ConflictResolution process only reasons on the initial update and on potential locking of applications that are in conflict with the initial update. Application locking can produce removal of an authorization rule, but cannot produce addition of a service to Calls or Provides sets. Consequently these two changes can only be a result of the initial update and we only need to check if such types of update (and also removal of an authorization rule as initial update) can bring the extended platform Θ_{ext} to be insecure.

- Initial update: addition of a service s to Provides_C. ConflictResolution component will check for all applications $A \in \mathcal{A}$ that if they invoke service s then they are authorized to do it. If ConflictResolution component had found such an application $D \in \mathcal{A}$ that calls s and is not authorized, then D is added to the ConflAppList. If update was accepted, then applications from ConflAppList become unselectable and do not provide their AppPolicy to the platform security policy $\mathcal{P}$. Hence such type of update (if the ConflictResolution component is implemented correctly and accepted an update) cannot lead to the extended platform Θ_{ext} being insecure on contract level.
- Initial update: addition of a service $C.s$ to Calls_D. The ConflictResolution component checked that if $s \in \text{Provides}_\text{C}$, then $(s, D) \in \text{sec.rules}_\text{C}$. If it does not hold, then ConflictResolution had added D to a list of conflicting applications. If update was accepted, then D has become unselectable. Consequently, extended platform is secure on contract level.
- Initial update: removal of an authorization (s, D) from $\text{sec.rules}_\text{C}$. The ConflictResolution component had ensured that if $s \in \text{Provides}_\text{C}$ and $s \in \text{Calls}_\text{D}$, then A was added to the ConflAppList. Consequently, since the update was accepted, D has become unselectable.

- A conflicting application C: an authorization rule (s, D) has been removed from $\mathsf{sec.rules_C}$ after execution of the ConflictResolution component. In this case application C has been found as conflicting with an initial update of application B and request for locking C had been processed. As initial update was accepted, we know that C was added to the ConflAppList, as only applets from this list are processed for possibility of removal. Consequently, C has become unselectable and is excluded from the security policy of the platform.

Let after a run of the ConflictResolution component functionality on contract level is broken. It means, by definition, that for some applications $C, D \in \mathcal{A}$ service $D.s \in \mathsf{func.rules_C}$ but $s \notin \mathsf{Provides_D}$.
As the extended platform Θ_{ext} was secure before the ConflictResolution run, then before an initial update either service $D.s$ was not in $\mathsf{func.rules_C}$, or $D.s$ was in $\mathsf{Provides_D}$, but was removed.
We reason again by the types of update. Addition of a functionally necessary service may happen only as initial update requested by applet B. We remind that this type of update is not processed by the ConflictResolution component and is, instead, verified by the PolicyChecker. Functionality of the platform for this type of update was established in the deliverable D6.3 [11]. Removal of a provided service may happen as initial update, or it can be an update created by the ConflictResolution component if some application was conflicting with applet B.

- Initial update: removal of a service s from $\mathsf{Provides_D}$. In this case for all $C \in \mathcal{A}$ the ConflictResolution component had checked that if $D.s \in \mathsf{func.rules_C}$ then C was added to the set ConflAppList of conflicting applications and, since the update was accepted, C has become unselectable.

- Service s has been removed from $\mathsf{Provides_D}$ during an update created by the ConflictResolution component run on the initial update of an applet B. Thus application D was labeled as a conflicting application and was added to the ConflAppList. Since the initial update was accepted, D has become unselectable. Moreover, any application $C \in \mathcal{A}$ such that $D.s \in \mathsf{func.rules_C}$ was also made unselectable.

□

It is also easy to prove (also reasoning by cases) that if the card manager will make selectable applications found by the algorithm 2.7.3, the extended platform will be secure.

2.9 Brief Notes about Verification for Access to Security Domains Services

As we discuss further in the chapter 6, one of the security properties that WP6 is considering is information protection. But only information protection by flow control is in the scope of the WP6 (on-device verification) in terms of the SecureChange project (see the chapter 6, [4] and [3] for the details). However, we may also consider information protection by access control property focusing on the access to the security domains services.

Access to the associated security domain services is granted to an application by the OPEN (`GPSystem` operations). Such operations are not considered in the Security-by-Contract framework, as they are not specified in the application policies. Correspond-

ingly, the ClaimChecker does not verify authorizations for access to security domains GP-specific services, though malicious applets can use standard Java Card means to reach such services. GP specification requirement states that "The `SecureChannel` interface shall only be exposed through the `GPSystem.getSecureChannel()` method" [12, P. 198]. Consequently, one possibility to protect against that is to forbid to security domains implementation of `getShareableInterfaceObject` method.

Another option, that brings more flexibility, is to verify also access to the security domains services. The idea for this verification could be the following: if an application B invokes a method that is among the `SecureChannel` and `SecureChannelX` services of security domain SD_C, then we need to check that $B \propto SD_C$ or $B \propto SD_B$ AND $SD_C \geq SD_B$.

Thus we will be able to verify the direct access to the security domains services on the Java Card level. Correctness of the access control implementation on the GlobalPlatform level is ensured by the Testing work package (see the chapter 6). More interesting is another threat scenario, that assumes collaboration of a malicious applet B and another applet C, that is associated with a security domain SD_C. In such setting, if C is also malicious or else buggy, B may use services of C to get access to the GP-specific services of SD_C. Such threat can be potentially investigated by, for example, the transitive control flow verification techniques, presented in the chapter 3.

2.10 Conclusions

In this chapter we presented full S×C framework for multi-application smart cards. This framework consists of two necessary components: the ClaimChecker and the PolicyChecker, and optional ConflictResolution component. Verification process done by this framework ensures that at run-time there are no illegal service invocations on the card (no illegal direct control flow).

In the deliverable D6.3 [11] we had presented the Security-by-Contract framework and advocated its benefits. In the current deliverable we extended the framework and presented its tight connections with the Java Card and GlobalPlatform-enabled smart cards. Thanks for the ClaimChecker component, which ensures that application contract is compliant with the actual code, we have shown how application implementation is related with the formal model.

Proposed framework provides strict security guarantees that were demonstrated formally. The guarantees also include introduced ConflictResolution component. This component brings on the platform more flexibility, as it allows the card issuer to specify more complex approach to updates (while keeping the platform secure).

The next goal that needs to be achieved by M36 with the deliverable D6.5 is realization of the S×C framework as a part of actual Java Card + GlobalPlatform smart card implementation (or on a card simulator).

3. Transitive Control Flow

The transitive control flow model described in the Chapter 4 of the deliverable D6.3 and in [10] only deals with addition of new applications and domains. The goal of this chapter is to investigate the impact of application removal and modifications of the security policy in this model. It is important to notice that the objective is not to roughly allow such changes but to provide an efficient solution with an overhead cost (memory consumption and computations) as low as possible to suit the requirements of smart cards, the most constrained target from the case studies of SecureChange.

We build on the definitions and algorithms introduced in the Chapter 4 of the deliverable D6.3. Nevertheless, before we detail the impact of new changes in the model, we first slightly modify the existing model in Section 3.1. Several solutions to deal with application removal are then detailed in the Section 3.2, and with domain removal in the Section 3.3. Finally we describe how to deal with modifications of domains hierarchy on-device in the Section 3.4, and with modifications of control flow policies for applications already installed in the Section 3.5.

3.1 Amendments to the existing model

In this model we focus on maintaining the device *secure* across some requested changes, but not on functional requirements and dependencies between applications. We assume such dependencies to be handled by the underlying platform.

3.1.1 Non-selectable applications

In the model described in the Chapter 4 of the deliverable D6.3, we constrain each application with unsolved dependencies to be *non selectable*. From the specific point of view of secure control flows, this constraint is useless when applied to an application with unsolved dependencies. In fact, if an installed application B relies on some services provided by an application A not yet loaded, then B cannot break the security of the system if it becomes selectable since it cannot concretely invoke methods while their bytecode is not loaded, whatever the security policy that will be attached by A to its shared services.

For this reason, we modify the $install$ procedure (Section 4.3.1 of the deliverable D6.3) applied after a successful verification (Algorithm 4.3.2 of the deliverable D6.3) of a newly installed application A in a domain d of a secure system $\mathcal{S} = (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P})$:

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

with $\mathcal{P}'$ defined as previously and

$$\mathcal{U}' = \{A' \mid A' \in \mathcal{A} \cup \{A\}, \exists A'.C'.m' \in \mathcal{M}_{A'}, (A'.C'.m' \leq A''.C''.m'') \implies (A'' \notin \mathcal{A})\}$$

This new definition of $\mathcal{U}'$ makes non-selectable only applications having unsolved inheritance dependencies, which corresponds to the actual behavior according to Java Card specifications [1].

3.1.2 Cleaning verification operations

The Algorithm 4.3.2 of the deliverable D6.3 refers to some "cleaning operations" (line 17) applied at the end of the installation of a new application in the system. These operations have not been detailed in the deliverable D6.3, we now explicit them.

Cleaning the $unsolved$ mapping consists in removing mappings of methods for which verifications have been postponed until their loading on the system and that have been successfully loaded during a run of the algorithm. Let $\mathcal{E}$ be the set of applications still not loaded in the system after a run of the Algorithm 4.3.2 of the deliverable D6.3.

$$\mathcal{E} = \{A \mid \exists A' \in \mathcal{A}, A \in wait(A') \implies A \notin \mathcal{A}\}$$

Let $\mathcal{M}_\mathcal{E}$ be the set of methods invoked by some loaded bytecode and defined in an application of $\mathcal{E}$.

$$\mathcal{M}_\mathcal{E} = \{E.C'.m' \mid E \in \mathcal{E}, \exists A.C.m \in \mathcal{M}, E.C'.m' \in \mathcal{I}^{direct}(A.C.m)\}$$

The methods in $\mathcal{M}_\mathcal{E}$ are the only one for which the control flow policy is still unknown, and thus for which the verification is postponed. Expected control flow policies gathered during the verification of methods calling methods in $\mathcal{M}_\mathcal{E}$ and stored in the $unsolved$ mapping are thus the only one to be kept. As a consequence, cleaning the $unsolved$ mapping simply consists in restricting its domain to $\mathcal{M}_\mathcal{E}$.

$$unsolved = unsolved_{|\mathcal{M}_\mathcal{E}}$$

Cleaning the $ruleswait$ mapping consists in removing mappings of methods loaded, installed and without any unsolved dependency to another application. Let $\mathcal{W}$ be the set of applications waiting for at least one other application.

$$\mathcal{W} = \{A \mid A \in \mathcal{A}, wait(A) \neq \emptyset\}$$

Let $\mathcal{M}_\mathcal{W}$ be the set of methods defined in applications of $\mathcal{W}$.

$$\mathcal{M}_\mathcal{W} = \{W.C.m \mid W.C.m \in \mathcal{M}_W\}$$

The methods in $\mathcal{M}_\mathcal{W}$ are the only one for which the verification process is stalled because the application in which they are defined has unsolved dependencies to other applications. Cleaning the $ruleswait$ mapping simply consists in restricting its domain to $\mathcal{M}_\mathcal{W}$.

$$ruleswait = ruleswait_{|\mathcal{M}_\mathcal{W}}$$

3.2 Removal of an application

There is no unique solution to handle application removal in the transitive control flow model. Before we investigate some solutions, we first provide an impact analysis of application removal in this model in the Section 3.2.1. From this analysis, we propose three solutions with different compromises between additional memory used and on-device computations required to deal with application removal. These solutions are presented by increasing additional memory required in the Sections 3.2.2, 3.2.3 and 3.2.4.

3.2.1 Impact analysis of application removal

The removal of an application installed in a *secure system* (Definition 4.2.4 of the deliverable D6.3) cannot lead to a non-secure system. This property is obvious from the definition of a *secure system* and can easily be proved by contradiction.

However, to be compliant with the incremental verification process applied at installation of new applications, the removal of an application A must put the system in the same state as if A was never installed. Basically, removing an application A from a secure system $\mathcal{S} = (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P})$ consists in rolling back to a state $\mathcal{S}' = (\mathcal{D}', \mathcal{A}', \mathcal{U}', \delta', \mathcal{P}')$ such that:

1. A is removed from the list of applications installed in the system;
$$A' = \mathcal{A} \setminus \{A\}$$
2. A is not mapped in δ;
$$\delta' = \delta_{|\mathcal{A}\setminus\{A\}}$$
3. U' is recomputed as on a successful installation (see Section 3.1.1);
4. any mapping related to A is removed in the mappings $rules$, $ruleswait$ and $wait$
$$ruleswait' = ruleswait_{|\mathcal{M}\setminus\mathcal{M}_A}$$
$$rules' = rules_{|\mathcal{M}\setminus\mathcal{M}_A}$$
$$wait' = wait_{|\mathcal{A}\setminus A}$$
5. the content of $rules'$, $wait'$, $ruleswait'$ and $unsolved'$ is restored as if only the applications in $\mathcal{A}'$ were installed.

This last point is actually the most complex to achieve as it requires some information not available in the system without re-analysis of some bytecode and/or additional information stored on-device. Several strategies can be applied to retrieve this information. Each strategy requires a varying amount of additional memory and computations. The strongest difficulty is to find a solution that provides a good compromise between memory and computations while ensuring that, at least, all methods impacted by the removal have been reconsidered.

Before we investigate different solutions to deal with application removal, it is important to determine formally what parts of the system are impacted by the removal of an application A. The smallest *entities* considered in this model are methods (Section 1.2). Given a method $A.C.m$, three sets of methods can be impacted by its removal: the methods that $A.C.m$ invokes directly or transitively, the methods that invoke $A.C.m$ directly or transitively, and the methods that override $A.C.m$. The methods invoked by $A.C.m$ are not actually impacted by its removal within the transitive control flow model simply because this is part of the incremental verification process to ensure at installation that $A.C.m$ invokes only authorized methods according to the domain in which it is going to be installed. However, the methods that invoke $A.C.m$ have been verified against a security policy that does not remain "valid" after the removal of $A.C.m$. If a new version of $A.C.m$ is installed after the removal of a version of $A.C.m$ with a different security policy, then it is mandatory to re-verify that all methods already installed that invoke $A.C.m$ are still authorized by the new security policy attached to $A.C.m$.

Among the methods that invoke $A.C.m$, only the methods that directly invoke $A.C.m$ are in fact impacted because of a property of the transitive control flow model. If a method $B_2.C_2.m_2$ transitively invokes $A.C.m$, then there exists a method $B_1.C_1.m_1$ that invokes $A.C.m$ directly and that is (transitively or directly) invoked by $B_2.C_2.m_2$. Since applications B_1 and B_2 have been successfully installed in the system, $rules(B_2.C_2.m_2) \subseteq rules(B_1.C_1.m_1)$ because of Lemma 4.3.2 of the deliverable D6.3. The re-verification of $rules(B_1.C_1.m_1) \subseteq rules(A.C.m)$ is thus sufficient to ensure that $rules(B_2.C_2.m_2) \subseteq rules(A.C.m)$, and to exclude all methods that transitively invoke the method $A.C.m$ to be removed from the set of methods impacted this removal.

We introduce a function Ω that gives for a method the set of methods impacted by its removal:

$$\begin{aligned}&\Omega : \mathcal{M} \longrightarrow \wp(\mathcal{M})\\ &A.C.m \longmapsto \{A'.C'.m' \mid A'.C'.m' \in \mathcal{M} \setminus \{A.C.m\}\\ &\qquad\qquad \wedge (A.C.m \in \mathcal{I}^{direct}(A'.C'.m') \vee A'.C'.m' \leq A.C.m)\}\end{aligned}$$

We extend this definition to applications, and state that an application A' is impacted by the removal of an application A if and only if there exists a method of A' that is impacted by the removal of a method of A:

$$\begin{aligned}&\Omega : \mathcal{A} \longrightarrow \wp(\mathcal{A})\\ &\quad A \longmapsto \{A' \mid A'.C'.m' \in \Omega(A.C.m), A' \in \mathcal{A} \setminus \{A\}, A'.C'.m' \in \mathcal{M}_{A'}, A.C.m \in \mathcal{M}_A\}\end{aligned}$$

3.2.2 No additional memory requirement

Without any additional information stored on-device, it is necessary to re-analyze the bytecode of some methods remaining installed after the removal of an application A. In order to reduce as much as possible the amount of bytecode to re-analyze after A's removal, we assume that the system is secure before A is removed.

Relying only on control flow policies, we are able to build a set $\Delta_{\mathcal{A}}(A)$ of applications that contains at least all applications having some methods to be re-analyzed when application A is removed. A method must be re-analyzed in two cases: it invokes a shared service of A, or it is implemented in a class that inherits a class defined by A.

Only the domains mentioned in control flow policies attached to the shared services of A can obviously harbor applications that invoke directly or indirectly (transitively) some shared services of A. For a secure system $\mathcal{S} = (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P})$ before $A \in \mathcal{A}$ is removed, the set $\Delta^{\mathcal{I}}_{\mathcal{A}}(A)$ of applications installed in domains from which A permits to invoke at least one of its shared services is:

$$\Delta^{\mathcal{I}}_{\mathcal{A}}(A) = \{B \mid B \in \mathcal{A} \setminus \{A\}, \exists A.C.m \in shareable(A), \delta(B) \in rules(A.C.m)\}.$$

This set of applications can contain applications that should not be re-analyzed because they do not invoke a shared service of A, but this cannot be determined *a priori*.

By definition, if an application B contains a class $B.C'$ that inherits a class $A.C$ defined in A, then all methods declared in $B.C'$ are necessarily attached a policy that permits the domain $\delta(A)$ where A is installed to call it (see Lemma 4.3.2 of the deliverable D6.3).

Using this property, we can build a set of applications that contains at least all applications having a class that inherits a class defined in A. The set $\Delta^{\leq}_{\mathcal{A}}(A)$ of applications having at least a class that implements a method attached a control flow policy which contains $\delta(A)$ is:

$$\Delta^{\leq}_{\mathcal{A}}(A) = \{B \mid B \in \mathcal{A} \setminus \{A\}, \exists B.C.m \in \mathcal{M}_B, \delta(A) \in rules(B.C.m)\}.$$

Remark 3.2.1 *A better definition of $\Delta^{\leq}_{\mathcal{A}}(A)$ could be achieved using the inheritance relation $\leq$, but this relation is not necessarily easy to test on-device out of the verification/installation process on the contrary to control flow policies.*

The set $\Delta_{\mathcal{A}}(A)$ of applications that contains at least all applications having some methods to be re-analyzed is simply the union of these two set of applications.

$$\Delta_{\mathcal{A}}(A) = \Delta^{\mathcal{I}}_{\mathcal{A}}(A) \cup \Delta^{\leq}_{\mathcal{A}}(A)$$

All applications impacted by A's removal will in this case be re-analyzed. From the definition of $\Omega(A)$ given in the Section 3.2.1, if $B \in \Omega(A)$ then $B \neq A$ and $\exists B.C'.m' \in \mathcal{M}_B, \exists A.C.m \in \mathcal{M}_A$ such that $A.C.m \in \mathcal{I}^{direct}(B.C'.m')$ or $B.C'.m' \leq A.C.m$. Since the system is *secure* before A's removal, if $A.C.m \in \mathcal{I}^{direct}(B.C'.m')$ then $\delta(B) \in rules(A.C.m)$, and if $B.C'.m' \leq A.C.m$ then $\delta(A) \in rules(B.C'.m')$. So in both cases $B \in \Delta_{\mathcal{A}}(A)$ by definition, and thus $\Omega(A) \subseteq \Delta_{\mathcal{A}}(A)$.

Since we cannot determine *a priori* which methods of applications in $\Delta_{\mathcal{A}}(A)$ need to be re-analyzed because of a lack of available information, the set $\Delta_{\mathcal{M}}(A)$ of methods to be re-analyzed contains all methods of all applications in $\Delta_{\mathcal{A}}(A)$.

$$\Delta_{\mathcal{M}}(A) = \bigcup_{B \in \Delta_{\mathcal{A}}(A)} \mathcal{M}_B$$

All methods remaining installed impacted by A's removal will be re-analyzed. From the definition of $\Omega(A)$, the set of methods impacted is $\bigcup_{A.C.m \in \mathcal{M}_A} \Omega(A.C.m)$, and if there exists a method $B.C'.m'$ in $\Omega(A.C.m)$ then $B \in \Omega(A)$. Since $\Omega(A) \subseteq \Delta_{\mathcal{A}}(A)$ and $\Delta_{\mathcal{M}}(A) = \bigcup_{B \in \Delta_{\mathcal{A}}(A)} \mathcal{M}_B$ then $\bigcup_{A.C.m \in \mathcal{M}_A} \Omega(A.C.m) \subseteq \Delta_{\mathcal{M}}(A)$.

This solution has the advantage to consume no additional memory, but lacks of sensitivity and can lead to useless re-analysis of a lot of bytecode.

3.2.3 Low additional memory

In order to refine the solution detailed in the previous section, and especially to reduce the number of applications unnecessarily re-analyzed, we can store some information about inter-applications dependencies.

At loading-time, a mapping $revdep : \mathcal{A} \longrightarrow \wp(\mathcal{A})$ can be used to store for each application A the set of applications that directly invoke one of A's shared services. Algorithm 3.2.1 shows the verification algorithm modified to include the building of $revdep$ (lines 3 and 13). When an application A is removed from a secure system, $revdep(A)$ contains exactly the set of applications having some bytecode to be re-analyzed. So, to reuse the same notations as is in the previous section, we have now for the removal of an application $A \in \mathcal{A}$:

$$\Delta_{\mathcal{A}}(A) = revdep(A)$$

$$revdep : A \longmapsto \{A' \mid \exists A'.C', \exists A.C, A'.C' \leq A.C\} \cup \{A' \mid \exists A.C.m \in \mathcal{I}^{direct}(A'.C'.m'), A'.C'.m' \in \mathcal{M}\}$$

All applications impacted by A's removal are obviously included in $revdep(A)$ because its definitions is exactly the definition of $\Omega(A)$.

```
1: if A.C inherits A'.C' then
2:
3:    revdep(A') ← revdep(A') ∪ {A}
4:
5:    if ∃A'.C'.m and (rules ∪ ruleswait)(A'.C'.m) ⊈ rules^i_A(A.C.m) then
6:       return FAIL
7:    else if wait(A') ≠ ∅ then
8:       wait ← wait[A ↦ wait(A) ∪ {A'}]
9: if unsolved(A.C.m) ⊈ rules^i_A(A.C.m) then
10:   return FAIL
11: for all invoke A'.C'.m' bytecode do
12:
13:   revdep(A') ← revdep(A') ∪ {A}
14:
15:   if A'.C' is already loaded then
16:      if {d} ∪ rules^i_A(A.C.m) ⊈ (rules ∪ ruleswait)(A'.C'.m') then
17:         return FAIL
18:   else
19:      unsolved ← unsolved[A'.C'.m' ↦ unsolved(A'.C'.m') ∪ {d} ∪ rules^i_A(A.C.m)]
20:      wait ← wait[A ↦ wait(A) ∪ {A'}]
21: return SUCCESS
```

Algorithm 3.2.1: Verification of the method $A.C.m$ loaded in the domain d with inter-applications dependencies analysis.

It is not possible to determine which methods of applications in $\Delta_{\mathcal{A}}(A)$ need to be re-analyzed. The set $\Delta_{\mathcal{M}}(A)$ of methods to be re-analyzed when application A is removed is thus:

$$\Delta_{\mathcal{M}}(A) = \bigcup_{B \in \Delta_{\mathcal{A}}(A)} \mathcal{M}_B$$

All methods remaining installed impacted by A's removal will be re-analyzed, *i.e.* $\bigcup_{A.C.m \in \mathcal{M}_A} \Omega(A.C.m) \subseteq \Delta_{\mathcal{M}}(A)$, for the same reason as the one given in the previous section.

When an application A is removed from a secure system $\mathcal{S}$ and after bytecode re-analysis, there is no need to keep the content of $revdep(A)$ on-device that can be cleaned with the following statement.

$$revdep' = revdep_{|\mathcal{A} \setminus \{A\}}$$

This solution has the advantage to consume very few additional memory, exactly $\lceil |\mathcal{A}|/8 \rceil$ byte(s) for each application A to store the content of $revdep(A)$ with a bit-encoding where each bit corresponds to an application present (bit set to one) or absent (bit set to 0) of $revdep(A)$. Even if this solution permits to reduce the amount of bytecode to be re-analyzed, it is not sensitive enough to select only methods that need to be re-analyzed within applications.

3.2.4 Large additional memory

To conclude the impact analysis of applications removal, we describe a last solution that has the advantage to require few computations and no bytecode re-analysis, but more memory on-device.

The idea is to apply the same kind of strategy described in the previous section but on methods. Across installations, we can maintain inter-methods dependencies in a mapping $revdep : \mathcal{M} \longrightarrow \wp(\mathcal{M})$ such that:

$$A.C.m \longmapsto \{A'.C'.m' \mid \exists A'.C', (A.C.m \in \mathcal{I}^{direct}(A'.C'.m')) \vee (A'.C'.m \leq A.C.m)\}.$$

The control flow policies of all methods remaining installed and impacted by A's removal will be re-verified because the definition of $revdep$ is exactly the definition of Ω given in the Section 3.2.1.

The Algorithm 3.2.2 corresponds to the incremental verification algorithm modified (line 4 and line 14) to introduce the building of $revdep$.

```
 1: if A.C inherits A'.C' then
 2:   if ∃A'.C'.m then
 3:
 4:     revdep(A'.C'.m) ← revdep(A'.C'.m) ∪ {A.C.m}
 5:
 6:     if (rules ∪ ruleswait)(A'.C'.m) ⊈ rules^i_A(A.C.m) then
 7:       return FAIL
 8:   if wait(A') ≠ ∅ then
 9:     wait ← wait[A ↦ wait(A) ∪ {A'}]
10: if unsolved(A.C.m) ⊈ rules^i_A(A.C.m) then
11:   return FAIL
12: for all invoke A'.C'.m' bytecode do
13:
14:   revdep(A'.C'.m') ← revdep(A'.C'.m') ∪ {A.C.m}
15:
16:   if A'.C' is already loaded then
17:     if {d} ∪ rules^i_A(A.C.m) ⊈ (rules ∪ ruleswait)(A'.C'.m') then
18:       return FAIL
19:   else
20:     unsolved ← unsolved[A'.C'.m' ↦ unsolved(A'.C'.m') ∪ {d} ∪ rules^i_A(A.C.m)]
21:     wait ← wait[A ↦ wait(A) ∪ {A'}]
22: return SUCCESS
```

Algorithm 3.2.2: Verification of the method $A.C.m$ loaded in the domain d with inter-methods dependencies analysis.

In case we are not interested in partial application's installation/removal (removing one loaded class or method), we can build $revdep$ only between methods defined in different applications, *i.e.* if $A \neq A'$ in Algorithm 3.2.2 (lines 4 and 14). Intra-applications dependencies are indeed not needed when only complete application's installation/removal is considered since in this case nothing has to be checked in the removed application.

The information stored in $revdep$, in addition to the control flow policies stored on-device, is sufficient to avoid any bytecode re-analysis. So, to reuse the notation of previous sections, the set $\Delta_{\mathcal{M}}(A)$ of methods to be re-analyzed when an application A is

removed is:

$$\Delta_{\mathcal{M}}(A) = \emptyset$$

However, the content of $unsolved$ and $wait$ has to be updated in consequence. The Algorithm 3.2.3 permits to reset these mappings according to the values stores in $revdep$.

1: **for all** $A.C.m \in \mathcal{M}_A$ **do**
2: **for all** $B.C'.m' \in revdep(A.C.m)$ **do**
3: $unsolved(B.C'.m') \leftarrow unsolved(B.C'.m') \cup \{A.C.m\}$
4: $wait(B) = wait(B) \cup \{A\}$
5: **if** $rules(B) \neq \emptyset$ **then**
6: $ruleswait(B) = rules(B)$
7: $rules(B) = \emptyset$

Algorithm 3.2.3: Refilling the mappings $unsolved$ and $ruleswait$ upon removal of an application A.

This solution to deal with application removal has the strong advantage to require no bytecode re-analysis thanks to the additional data kept on-device. The amount of data kept is however larger than with the solutions described previously and it can thus only be applied if its memory overhead is suitable for the targeted system.

3.3 Removal of a domain

Removing a domain is possible only if it harbors no application. This assumption is reasonable whatever the underlying system is. If the underlying system permits removal of a domain while it contains some application(s), we assume that these applications are removed using one of the techniques described in the previous section.

To permit the removal of a domain $D \in \mathcal{D}$ in a system $\mathcal{S} = (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P})$, the following condition must hold:

$$\nexists A \in \mathcal{A}, \delta(A) = D$$

Under this assumption before removal of a domain D, no additional care is needed for the transitive control flow model.

$$removeDomain(\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P}) = (\mathcal{D} \setminus \{D\}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P})$$

3.4 Modifications of the domains hierarchy

In the transitive control flow model, domains are seen as individual and independent containers of applications, without any specific behavior between domains having hierarchical links (father domain, sub-domains or brother domain). Changes to domains hierarchy, such as extraditing a domain (and of course the applications it harbors) elsewhere in the domains hierarchy, have thus no impact on this model and do not require any re-verification.

Nevertheless, as already depicted in the deliverable D6.3, it is crucial that entry (*i.e.* installation) of an application in a domain is submitted to access control rules of the underlying system that cannot be bypassed (see Chapter 6 for details on GlobalPlatform). In case an application is extradited to another domain, it is mandatory to re-verify completely its bytecode, exactly as if it was newly installed, to ensure that the methods it invokes can be invoked from its new domain.

3.5 Modifications of control flow policies

The weakening of control flow policies has already been described in the Section 4.3.2 of the deliverable D6.3. We now present how to deal with restrictions of control flow policies of already installed applications. We describe for each solution provided for the application removal problem described in the previous section a corresponding solution for the problem of restricting control flow policies.

Restricting the control flow policy of a method has exactly the same impact on the system as removing a method described in the Section 3.2.1 in terms of re-verification required. Indeed, restricting the control flow policy of a method is equivalent to replace (remove and then install) the method by itself but with a different security policy. Proofs that all impacted applications (or methods) are re-analyzed are not provided because they can be simply derived from the ones provided for application removal.

3.5.1 No additional memory requirement

Given a set M of methods whose control flow policies have been restricted, $\Delta^{\mathcal{I}}_{\mathcal{A}}(M)$ is the set of applications that can contain some methods that need to be re-analyzed because of a possible call to a method of M.

$$\Delta^{\mathcal{I}}_{\mathcal{A}}(M) = \{B \mid B \in \mathcal{A}, \delta(B) \in \bigcup_{A.C.m \in M} rules(A.C.m)\}$$

For the same set M of methods whose control flow policies have been restricted, $\Delta^{<}_{\mathcal{A}}(M)$ is the set of applications that can contain some methods that need to be re-analyzed because they can override a method of M.

$$\Delta^{<}_{\mathcal{A}}(M) = \{B \mid B \in \mathcal{A}, \exists A.C.m \in M, \delta(A) \in rules(B.C.m)\}$$

Remark 3.5.1 *A better definition of $\Delta^{<}_{\mathcal{A}}(M)$ could be achieved using the inheritance relation $<$, but this relation is not necessarily easy to test on-device out of the verification/installation process on the contrary to control flow policies.*

The set $\Delta_{\mathcal{A}}(M)$ of applications that contains at least all applications having some methods to be re-analyzed is simply the union of these two set of applications.

$$\Delta_{\mathcal{A}}(M) = \Delta^{\mathcal{I}}_{\mathcal{A}}(A) \cup \Delta^{<}_{\mathcal{A}}(A)$$

Since with this solution it is not possible to determine which methods of these applications need to be re-analyzed, the set $\Delta_{\mathcal{M}}(M)$ of methods to be re-analyzed when the control flow policies of the methods of M have been restricted is simply the union of all methods of applications in $\Delta_{\mathcal{A}}(M)$.

$$\Delta_{\mathcal{M}}(M) = \bigcup_{B \in \Delta_{\mathcal{A}}(M)} \mathcal{M}_B$$

3.5.2 Low additional memory

The $revdep$ mapping introduced in the Section 3.2.3 can be reused to determine the set $\Delta_{\mathcal{A}}(M)$ of applications to be re-analyzed given a set M of methods whose control flow policies have been restricted.

$$\Delta_{\mathcal{A}}(M) = \bigcup_{A.C.m \in M} revdep(A)$$

Once more, it is not possible with this solution to determine which methods of these applications need to be re-analyzed, so all their methods need to be re-analyzed.

$$\Delta_{\mathcal{M}}(M) = \bigcup_{B \in \Delta_{\mathcal{A}}(M)} \mathcal{M}_B$$

It is important to remark that in the case of restricted control flow policies the mapping $revdep$ does not need to be updated, on the contrary to application removal.

3.5.3 Large additional memory

Once more we reuse a mapping introduced in the analogous solution for application removal to determine the set of methods to be-reanalyzed when control flow policies of some methods are restricted. The mapping $revdep$ introduced in the Section 3.2.4 actually contains necessary information to avoid bytecode re-analysis. The Algorithm 3.5.1 simply describes the re-verification of control flow policies for methods impacted by the restriction of control flow policies attached to methods of M. To be correct this algorithm must be applied in a transactional way, *i.e.* if it fails the previous control flow policies attached to methods of M must be restored.

```
1: for all A.C.m of M do
2:   for all A'.C'.m' ∈ revdep(A.C.m) do
3:     if rules(A'.C'.m') ⊈ rules(A.C.m) then
4:       return FAIL
5: return SUCCESS
```

Algorithm 3.5.1: Verification of the consistency when restricting control flow policies of methods of M.

3.6 Conclusions

In this chapter we completed the transitive control flow model introduced in the Chapter 4 of the deliverable D6.3 [11] to permit application removal and update to the security policy of an installed application. Since dealing with these new changes requires additional resources on the system, we proposed several solutions with variable compromise between overhead memory used and computations. For each proposed solution, we showed that no false negative can be produced by later installations of applications, and tried to minimize as much as possible the number of false positives.

4. Global Policy

The global policy model, introduced in the Chapter 5 of the deliverable D6.3, aims at detecting sequences of method calls that are forbidden by the system. This model only deals for now with addition of new applications. Before we introduce application removal and security policy updates in this model, we first provide in the Section 4.1 a correction of the verification Algorithm 5.6.1 from the deliverable D6.3 because it is unfortunately not a correct implementation of the theoretical model. Then, we build on the definitions and algorithms introduced in the Chapter 5 of the deliverable D6.3 to enrich the model with application removal in the Section 4.2, and modification of the global security policy in the Section 4.3. On the contrary to the transitive control flow model described in the previous chapter, the global policy model does not use on the notion of domains so it is not impacted at all by the relocation of an application into another domain nor to changes of the domains hierarchy.

4.1 Amendments to the existing implementation

The Algorithm 5.6.1 of the deliverable D6.3 for on-device verification of proof annotations has some drawbacks that can cause a method (a class, an application) to be accepted while it must not be accepted according to the theoretical model. In addition, we also clarify the way the security policy is recorded on-device as well as initial values in footprint repositories, and make some modifications to avoid the possible rejection of a class (an application) with dead code. The Algorithm 4.1.1 replaces the Algorithm 5.6.1 of the deliverable D6.3 and includes all the following modifications:

- line 2: the believed footprint of $A'.C'.m'$ is dropped only if there exists a verified footprint in R for $A'.C'.m'$;
- line 5: if there is no verified footprint in R for $A'.C'.m'$, then the believed footprint in R_{tmp} for $A'.C'.m'$ ($\top$ by default) is restricted to its common parts with the footprint $F_{A'.C'.m'}$ coming with the application currently verified; since this restriction can only remove factors contributing to invalid traces, it cannot cause methods already loaded and successfully verified with the previous footprint to be rejected if they were re-verified with the restricted footprint;
- line 13: the initial footprint F_{init} of a method $A.C.m \in \Sigma$ now includes its expected contribution according to the automaton of forbidden sequences of the system:

$$F_{init} = \{(s_i, s_j) \mid \zeta(s_i, A.C.m) = s_j \vee (\zeta(s_i, A'.C'.m') = s_j \wedge A.C.m \leq A'.C'.m')\}$$

- line 15: if the address of the current instruction is the point of a branching instruction, then the current footprint F_{tmp} must be included in the footprint annotation $proof[A.C.m][i]$ at this point;
- line 18: the current footprint is replaced by the footprint annotation $proof[A.C.m][i]$ that can be more complete than the one currently computed, especially if the address of the current instruction is the point of more than one branching instructions; indeed this case should require to compute the union of footprints coming from all possible paths, which is not computable on-device;
- line 20: in order to deal with dead code, the current footprint is reset if the current instruction is a non-conditional branching instruction to an address other than $i+1$, the next bytecode instruction;
- line 22: if $P_{A.C.m}[i]$ is a bytecode instruction branching at address a, then it must exists a proof annotation for the footprint at position a otherwise the verification must fail;
- line 34: $R[A.C.m] = F_{A.C.m}$ and $R_{tmp}[A.C.m] = \bot$.

The system starts with no application installed and repositories initialized such that:

- the repository R of verified methods footprints is empty: we write $R[m] = \bot$ for all methods m;
- the temporary repository R_{tmp} of methods footprints is empty: we write $R_{tmp}[m] = \top$ for all methods m.

In this initial state, no method has a verified footprint so each method is assigned the encoded empty footprint $\bot$ (all bits set to zero). On the contrary, each method is assigned in the temporary repository the fully saturated footprint denoted ✠ in the formal model and encoded by the value $\top$ (all bits are set to one). This default value denotes the "worst" possible footprint of method; this assumption is mandatory to be able to reject a method during its verification if it invokes another method for which no footprint is known in R or in R_{tmp}.

4.2 Removal of an application

The global policy model implies off-device computations that cannot be done on-device because of the low resources available. Currently, proof annotations computed off-device are dropped after the verification process at loading-time because bytecode in never re-verified. Without these proof-annotations no re-verification is possible on-device, so the amount of practicable solutions for dealing with application removal is quite limited.

The Section 4.2.1 is an impact analysis of application removal in the global policy model. After this analysis we describe two solutions to deal with application removal in the Section 4.2.2 and in the Section 4.2.3.

4.2.1 Impact analysis of application removal

As it has already be described in the Section 3.2.1, the removal of an application installed in a *secure system* cannot lead to a non-secure system. To be compliant with the incremental verification process, it is however needed to reset the system in a coherent state for later installations.

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

Algorithm 4.1.1: Loading of a class C of application A.

Basically, removing an application A from a secure system $\mathcal{S} = (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P})$ with a global policy $\mathcal{G}$, a repository of verified footprints R and a repository of believed footprints R_{tmp} consists in rolling back to a state $\mathcal{S}' = (\mathcal{D}', \mathcal{A}', \mathcal{U}', \delta', \mathcal{P}')$ with the same global policy $\mathcal{G}$, a repository of verified footprints R' and a repository of believed footprints R'_{tmp} such that:

1. A is removed from the list of applications installed in the system;

$$A' = \mathcal{A} \setminus \{A\}$$

2. A is not mapped in δ;

$$\delta' = \delta_{|\mathcal{A}\setminus\{A\}}$$

3. U' is recomputed as on a successful installation (see Section 3.1.1);

4. any footprint related to A is removed from the repository R of verified footprints;

$$R' = R_{|\mathcal{M}\setminus\mathcal{M}_A}$$

5. the content of the repositories R and R'_{tmp} is restored as if only the applications in $\mathcal{A}'$ were installed.

Resetting the content of the two repositories is not trivial to achieve as it requires some information not available in the system without additional information stored on-device, and/or re-verification of some bytecode (if proof annotations are permanently kept on-device).

As depicted in the Algorithm 4.1.1, the footprint of a method is compositionally verified at installation on-device using the verified footprints of invoked methods if they are available, or the intersection of believed footprints brought by applications themselves otherwise. If we consider the removal of a method $A.C.m$, only methods that override or invoke $A.C.m$ are obviously impacted by the removal of $A.C.m$. However, not only the methods that directly invoke $A.C.m$ can be impacted by its removal. If a method $B_1.C_1.m_1$ directly invokes $A.C.m$, then the footprint of $B_1.C_1.m_1$ can be impacted if a new version of $A.C.m$ is installed. But it is not mandatory that the footprint of $B_1.C_1.m_1$ will change after this update of the footprint of $A.C.m$, and so recursively for the footprints of methods that invoke $B_1.C_1.m1$.

In this context, it is only possible to give an over-approximation for the set of methods impacted by the removal of a method $A.C.m$ that is the set of all methods that directly or transitively invoke $A.C.m$ or override $A.C.m$. To reuse the notations introduced in the Section 3.2.1, we introduce a function Ω that gives for a method the set of methods impacted by its removal:

$$\begin{aligned}
&\Omega : \mathcal{M} \longrightarrow \wp(\mathcal{M}) \\
&A.C.m \longmapsto \{A'.C'.m' \mid A'.C'.m' \in \mathcal{M} \setminus \{A.C.m\} \\
&\qquad\qquad \wedge (A.C.m \in \mathcal{I}(A'.C'.m') \vee A'.C'.m' \leq A.C.m)\}
\end{aligned}$$

We extend this definition to applications, and state that an application A' is impacted by the removal of an application A if and only if there exists a method of A' that is impacted

by the removal of a method of A:

$$\Omega : \mathcal{A} \longrightarrow \wp(\mathcal{A})$$
$$A \longmapsto \{A' \mid A'.C'.m' \in \Omega(A.C.m), A' \in \mathcal{A} \setminus \{A\}, A'.C'.m' \in \mathcal{M}_{A'}, A.C.m \in \mathcal{M}_A\}$$

4.2.2 Footprints ghosts

Let us consider the removal of an application A having a method $A.C.m$, and an application B with a method $B.C'.m'$ that invokes $A.C.m$ remaining installed in the system after A's removal.

If A was installed before B, then the verified footprint of $A.C.m$ has been used to verify the footprint of $B.C'.m'$. When A is removed, the verified footprints of A's methods can obviously not be kept in the repository R of verified footprints. To guarantee that the verified footprint of $B.C'.m'$ remains valid after A's removal, it is just necessary to store the verified footprint of $A.C.m$ before A's removal as a new believed footprint of this method in R_{tmp}. By construction, this will ensure upon a later installation of $A.C.m$ that its attached footprint will be included in the footprint $R_{tmp}[A.C.m]$ used to verify $B.C'.m'$, so that no re-verification of $B.C'.m'$ is needed.

If A was installed after B, then $B.C'.m'$ has been verified with a believed footprint of $A.C.m$. This believed footprint of $A.C.m$ was in this case the intersection of all the believed footprints of $A.C.m$ brought by applications known when B was verified. Since $A.C.m$ has been successfully installed in the system, it implies that the verified footprint of $A.C.m$ was included in its believed footprint. Upon A's removal, the verified footprint of $A.C.m$ can thus be transferred from R to R_{tmp} in the exact the same way as described previously, and with the same guarantee that the verified footprint of $B.C.m$ remains valid without re-verification on installation of a new version of $A.C.m$.

Actually, in both cases only the footprints of A's shared methods need to be transferred from R to R_{tmp}. Since by hypothesis applications can interact only through shared methods, there cannot exist invocation of a non-shared method of A from B. Stored in R_{tmp}, the footprints of removed methods act as *ghosts* in the exact same way as the believed footprints of methods not yet verified in the original algorithm.

The impact analysis of application removal in this model given in the Section 4.2.1 states that any method that (directly or transitively) invokes some method to be removed is potentially impacted by this removal. However, footprints ghosts permit to limit the impact of removal thanks to a constraint applied to new footprints of a previously installed method. If a method $A.C.m$ is removed, we ensure that its removed footprint will remain a correct over-approximation of any new possible footprint for this method and so that any previously verified footprint of a method that (directly or transitively) invokes $A.C.m$ remains a valid over-approximation of its true footprint.

Concretely, the Algorithm 4.2.1 achieves the necessary modifications on the repositories upon the removal on an application A. This solution could be slightly improved by keeping only footprints ghosts of shared methods invoked by other applications, which implies to attach an invocation counter to each shared method and to maintain these counters across installations (to increment the counters) and removals (to decrement the counters).

Using footprints ghosts to deal with application removal has the strong advantage to require no additional memory or bytecode re-verification. However, it also suffers from a strong drawback: once a shared method is loaded in the system with its verified footprint,

1: **for all** $A.C.m$ **do**
2: **if** $A.C.m \in shareable(A)$ **then**
3: $R_{tmp}[A.C.m] = R[A.C.m]$
4: $R[A.C.m] = \bot$

Algorithm 4.2.1: Creation of footprints ghosts upon the removal of an application A.

this footprint can be only restricted during the rest of the system's life-cycle, *i.e.* no new version of the method can have a behavior that involves more factors (left, regular or right factors) than the one previously installed.

4.2.3 Relying only on believed footprints

To cope with the main limitation of footprints ghosts, another solution is to rely only on believed footprints during verification rather than on verified footprints (if they are known).

If an application A is not installed, it may exist believed footprints for A's methods in R_{tmp} brought by other installed applications that invoke A's methods. Once A is successfully verified and installed, all the believed footprints of A's methods are removed from the temporary repository R_{tmp}, and the verified footprints of A's methods are stored in R. At this stage it is thus not possible to restore the believed signature of A's methods if A is removed. Moreover, if A invokes some shared methods of other applications not yet installed, then the believed footprints of these methods have been updated to intersect with the believed footprints brought by A. If A is removed, the believed footprints of these methods can also not be restored. To achieve the reset of the temporary repository when A is remove, it is mandatory to keep all the individual believed footprints of shared methods brought by applications until there are uninstalled and not only an aggregation of these footprints.

The Algorithm 4.2.2 corresponds to the verification algorithm modified to achieve this solution. Concretely, the modifications are the followings. A new repository $F_{restore}$ is defined to gather the collection of believed footprints coming with each application. We choose to still maintain R_{tmp} in a incremental way (line 4) to avoid multiple comparisons (line 6) with all believed footprints now stored in $F_{restore}$. However, R_{tmp} can be removed to reduce memory requirements if necessary, which will increase the number of required comparisons. Indeed, in case R_{tmp} is removed, each expression of the form $F_{A.C.m} \not\subseteq R_{tmp}[A.C.m]$ has to be replaced by its equivalent: $\exists A \in \mathcal{A}, F_{A.C.m} \not\subseteq F_{restore}[A.C.m][A]$.

Composition is not done with verified footprints anymore but with the believed ones brought on by the application itself. Actually, because the verified footprint of a method $A.C.m$ can vary across installations and removals of applications, it is mandatory to ensure that the footprints of $A.C.m$ used during the verification of other methods will remain some valid over-approximations of the verified footprint of $A.C.m$.

Upon removal of an application A, the believed footprints of the external methods invoked by A must be restored. This last operation requires to recompute the believed footprints according to the collection of believed footprints of all external methods brought by applications remaining installed, as depicted in the Algorithm 4.2.3.

Finally, the removal of an application A must reset the verified footprint of each method of A to its default value $\bot$ in R:

$$\forall A.C.m, R[A.C.m] = \bot.$$

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

Algorithm 4.2.2: Loading of a class C of application A.

1: **for all** $F_{restore}[A.C.m]$ **do**
2: $R_{tmp}[A.C.m] = \top$
3: **for all** $A' \in \mathcal{A}$ **do**
4: $R_{tmp}[A.C.m] = R_{tmp}[A.C.m] \& F_{restore}[A.C.m][A']$

Algorithm 4.2.3: Rollback after the removal of an application.

4.3 Modification of the security policy

A security policy in this model is monolithic in the sense that it consists in one security graph of forbidden method calls for the whole system. The security policy is currently setup before any application is installed in the system. This choice has several consequences on the practicable solutions to deal with modifications of the security policy.

From theoretical point of view, any modification to the global security policy $\mathcal{G} = (\Sigma, S, s_0, \zeta, s_F)$ (Definition 5.3.1 of the deliverable D6.3) invalidates some footprints computed with the previous policy as soon as the language $\mathcal{L}(\mathcal{G})$ is modified. For instance, if a label m is replaced by a label m' in a transition from a state s_1 to a state s_2 in ζ, then the footprint of any method that involves this transition, before or after the replacement of m by m', becomes invalid. However, some footprints also become invalid even if $\mathcal{L}(\mathcal{G})$ is not modified. For instance, if a method m is added to Σ or removed from Σ then the footprint of any method that invokes m is invalid because the projection onto Σ of its possible execution traces is not valid anymore (Definition 5.4.2 of the deliverable D6.3).

Methods footprints are compositionally computed (Proposition 5.4.9 of the deliverable D6.3) so footprints computed using footprints now marked as invalid can also become invalid. Even if this domino effect is not systematic as it depends on the content of methods themselves, any footprint that recursively depends on an invalid footprint can also potentially become invalid. The impact of a modification to the security policy of a system is thus equivalent to the one provided in the Section 4.2.1 in case of application removal in the global policy model.

Methods footprints are computed off-device and are then bitwise encoded (Section 5.6.2 of the deliverable D6.3) in order to minimize their size for on-device verification and storage. The bitwise encoding of footprints is directly derived from the security policy graph $\mathcal{G}$ because each bit of the encoded footprint corresponds to a possible transition in $\mathcal{G}$. So all footprints stored on-device have at least to be re-encoded as soon as the topology of $\mathcal{G}$ changes.

All the issues previously highlighted strongly reduce the scope of practicable solutions for dealing with modifications of the security policy in the global policy model. In fact, in case the security policy of a system is modified, the only solution is to replay the verification of all its code according to new footprints and proof annotations from scratch, *i.e.* with emptied repositories of footprints and all applications marked as non selectable. This process must be achieved in a transactional way to ensure that the system remains in a secure state, so if the verification process fails at some point previous footprints and security policy must be restored.

4.4 Conclusions

In this chapter we first gave some corrections to the original model introduced in the Chapter 5 of the deliverable D6.3 [11], and then to provide necessary methods to permit application removal and update to the security policy of an installed application. The main difficulty we had to face was the monolithic security policy stored on-device that strongly restricts the number of practicable solutions for application removal but especially for security policy updates. We did not propose any solution where proof annotations are kept on-device because of the large memory overhead it *de facto* represents, even though it does not solve the problem we encountered with policy updates.

For each solution proposed for application removal, we showed that no false negative can be produced by later installations of applications. Using footprints ghosts has the strong advantage to require very few additional memory, but it also has the strong disadvantage to create a potentially large number of false positives which is not a desired feature for lifelong evolving systems. Relying only on believed footprints actually copes with this drawback using some additional memory for storing believed footprints collections. The global policy model relying only on believed footprints constitutes a very promising approach to enhance some security aspects in small open devices.

5. Non-interference

The non-interference model for open systems based on GlobalPlatform, or at least using the same concept of domains, was introduced in the Chapter 6 of the deliverable D6.3 [11]. The goal of this model is to verify information flow in terms of data in order to ensure data confidentiality. The non-interference model is two-fold: on one hand the flow signatures of methods computed off-device by the STAN tool and verified on-device at loading-time, and on the other hand the inter-domain sharing policy of intra-domain secret data. In the deliverable D6.3, only installation of new applications was described, not application removal nor modifications of the security policy. In this chapter we complete the non-interference model to permit such changes. We first describe in the Section 5.1 how to deal with removal of an application, and in the Section 5.2 with the removal of a domain. Then we describe in the Section 5.3 how to deal with modifications of on-device domains hierarchy, and finally in the Section 5.4 how to deal with modifications of the two-fold security policy of this model.

5.1 Removal of an application

The on-device verification algorithm of flow signatures is highly similar to the on-device verification of methods footprints in the global policy model described in the Chapter 4 of this deliverable. Both algorithms rely on two repositories of flow signatures/methods footprints, a first one for verified signatures/footprints and a second one for believed signatures/footprints. These repositories are maintained in the exact same way in the two models, so the solutions described in the Section 4.2 for dealing with application removal in the global policy model can also be applied to the non-interference model.

The inter-domain sharing policy is not impacted at all by the application removal (or installation), so it remains unchanged when an application is removed (or installed).

5.2 Removal of a domain

We assume that a domain can be removed from the system if and only if it harbors no application, and that this assumption is checked by the underlying subsystem. If the underlying system permits removal of a domain while it contains some application(s), we assume that these applications are removed using one of the techniques described in the previous section.

Concretely, removing a domain has no impact in this model and can be done without any specific treatment. However, in order to save some memory, one may want to also remove the non-interference policy rule attached to the removed domain. This operation is

straightforward as it simply consists in removing from the $dataflow$ array of encoded non-interference policy rules (Section 6.3.1 of the deliverable D6.3) the line and the column corresponding to its encoded non-interference policy rule.

5.3 Modifications of a domains hierarchy

The non-interference policy of a system strongly depends on its domains hierarchy (Definition 6.2.4 of the deliverable D6.3). To simplify the management of non-interference policy rules on-device, inter-domain flows of secret data authorized by the non-interference security policy of the system are precomputed for a given hierarchy of domains and encoded into a two dimensional array of bits $dataflow$ of size n if the system can support up to n domains.

In the non-interference model, extraditing a domain elsewhere in the hierarchy of domains has the same impact as modifying the non-interference policy of the system. For this reason, the process described in the Section 5.4.2 for dealing with modifications of the non-interference policy of a system can also be applied for dealing with extradition of a domain.

In case an application is extraded to another domain, we proceed in the exact same way as in the global policy model. It is actually mandatory to check all flow signatures of methods installed in the system, but not the adequation between flow signatures and the bytecode since neither the bytecode nor the flow signatures have changed.

5.4 Modifications of the security policy

As the non-interference model itself, the security policy is also two-fold. Secret attributes of each class of each application are defined off-device to permit computation of methods signatures by the STAN tool before it embeds them in the corresponding bytecode. When an application is installed on-device in some domain, the system first verifies that signatures of the application's methods are valid, and then verifies that the new application does not create any illicit flow of data between domains according to its inter-domain security policy. Evolution of the "security policy" can thus occur at two levels: modifications of the secret attributes of a class, and inter-domain sharing policy.

5.4.1 Modifications of per-class secret attributes

The computation of the flow signature of a method by the STAN tool requires to define which attributes of classes are *secret*, the others being *public* by default. This label assigned to attributes of classes may be updated as an application evolves, so it is mandatory to provide a way to update the flow signatures of methods already loaded on-device.

On the contrary to the global policy model where it is not easy to update methods footprints without a complete re-verification of all methods because the policy is monolithic and system-centric, it is easy to provide an incremental solution in this model because the definition of secret attributes is independent from one class to another.

Changing the flow signatures of a bunch of methods has of course some impact on other parts of the system. The set of methods impacted by the modification of the flow signature of a method is in fact equivalent to the set of methods impacted by its removal, so as for the footprints of methods in the global policy model. As these models rely on the same techniques (compositional off-device computation and on-device verification

according to proof annotations), the impact of modifying the flow signature of a method is equivalent from the theoretical point of view to the one provided for the modification of a footprint in the global policy model described in the Section 4.3. Nevertheless, since the definition of secret attributes is made class per class independently, an efficient manner of updating flow signatures can be reached in practice on the contrary to the complete re-verification of all methods mandatory in the global policy model.

Concretely, to update the new flow signatures computed off-device for a class whose definition of secret attributes is updated, we first remove all its methods from the system (not the code in fact but only flow signatures of its methods) using one of the solutions described in the previous section. Then we just have to replay the verification of its code according to newly updated proof annotations and flow signatures of its methods. In order not to break the system upon verification failure of the new flow signatures, this process must be conducted in a transactional way: if the verification of the new signatures fails then previous signatures must are restored.

5.4.2 Modifications of non-interference policy of a system

Changing only the non-interference policy rules of a system (Definition 6.2.4 of the deliverable D6.3) does not impact flow signatures computation, so it does not require any code re-verification. Upon a change of non-interference policy rules, only re-verification of some flow signatures of methods already installed is needed.

A non-interference policy rule (Definition 6.2.2 of the deliverable D6.3) is a function assigned to a domain that gives a set of domains authorized to acquire its secret data according to the current domains hierarchy of the system (Definition 6.2.1 of the deliverable D6.3). A system is said secure if all installed methods are secure (Definition 6.2.5 of the deliverable D6.3) with respect to the non-interference policy rules of the system.

Because the non-interference policy of a system is precomputed and encoded in the array $dataflow$, changing some non-interference policy rules consists in re-encoding the new non-interference policy rules according to the hierarchy of domains in the system, and then to verify that already installed methods are secure according to this new policy. Flow signatures of methods are clearly not to be re-verified against the corresponding bytecode because neither the bytecode nor the flow signatures have changed.

Actually, all flow signatures must be checked upon a change of non-interference policy rules only if there exists a previously authorized flow between two domains that is now forbidden. Indeed, if all inter-domain flows previously authorized are still authorized after the modification of the non-interference policy rules, then the new security policy is more permissive so the security of the system is obviously preserved. If a re-verification of flow signatures against the new security policy is required, it must be done on all methods installed in the system according to the rule depicted in the Definition 6.2.5 of the deliverable D6.3. The complete re-verification process must of course be conducted in a transactional way not to put the system in an incoherent and insecure state, *i.e.* if the re-verification of at least one flow signature fails with the new non-interference policy rule then the old one must be restored.

5.5 Conclusions

In this chapter we completed the non-interference model introduced in the Chapter 6 of the deliverable D6.3 to permit application removal, modifications of the security pol-

icy and modification of the domains hierarchy of a system. Since the non-interference model is strongly ressembling to the global policy model described in the previous chapter, we build on the results achieved for the global policy model to introduce application removal in the non-interference model. This is fortunately not the case for modifications of the security policy in the non-interference model. The problems encountered with the global policy model to deal with such kind of modification are caused by its monolithic and system-centric security policy, which is not the case for the non-interference model where methods flow signatures computed off-device are independent of the inter-domain sharing policy of the system. The solutions we propose to deal with changes are free of false negative, and minimize as much as possible the number of false positives.

6. Integration with Testing Work Package

This chapter describes a connection between Verification (WP6) and Testing (WP7) work packages of SecureChange project. WP6 and WP7 both work on information protection security property of POPS case study and their collaboration is therefore instantiated on this case study. In this chapter we discuss contribution of each work package for information protection property. Also we provide descriptions of attack scenarios that could break information protection. We then demonstrate how WP6 and WP7 together provide protection against these attack scenarios and the advantages of connection.

6.1 POPS case study reminder

We do not provide a full description of POPS case study, which can be found in the deliverable D1.1 [5], the deliverable D1.2 [3] and the Case Study Properties document [4]. We give a reminder of information protection security property identified for POPS case study. Following is a citation from [3, Sec. 5.2]. This property can be decomposed in two parts:

- **Information protection by access control**: Any command received by the card must respect the card and applet life-cycle. Its means that any command received in a state s leads to a state s' and the resulting transition from s to s' is correct w.r.t. the specifications.
- **Information protection by flow control**: The applications on the card must be "isolated" (segregation), that means no illegal access to the data from one application to another. In order to enforce isolation, 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 GP (key isolation relying on the underlying Java Card firewall and the GP OPEN). Therefore, some properties must be verified, 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.

There are two change requirements that are identified for POPS case study.

Specification evolution: An UICC card embeds a component called the card manager, implemented accordingly to GlobalPlatform specifications v2.1. This card component has been extensively verified and tested. The GlobalPlatform specification [12] has been enhanced and extended and v2.2 has been issued. The card manager software component has been updated and extended against this new version. For simplicity reason, we restrict the 2.2 scope to the UICC configuration. **Goal**: prove/demonstrate/test that the security properties are still preserved.

Software update: The certified UICC card is deployed in the field. The mobile operator, owner of the card, has a new partner, a bank. He loads a new security domain (a Java Card application) on the UICC (card) using an OTA mechanism. This bank will have the delegated management privilege from the Mobile Network Operator to manage its applications in a confidential way. In particular, the bank will use its security domain to load an e-purse on the card. **Goal**: prove/demonstrate/test that the new application preserves (does not break) the consistency of the existing and implemented security policies.

Specification evolution and Software update are change scenarios and information protection property has to be preserved under these change scenarios.

6.2 Verification Work Package (WP6)

WP6 provides two types of verification techniques: development-time and run-time (or on-device). Development-time verification techniques for POPS case study are dedicated to Denial of Service security property and are out of scope of the integration.

On-device verification techniques are focusing on inter-application communications. For POPS case study WP6 has provided the following techniques: direct control flow verification, transitive control flow verification, global policy verification and non-interference verification. Below we specify briefly each of these techniques.

Direct control flow verification: This technique is based on Security-by-Contract approach and uses the fact that on Java Card applications can interact only through specially defined Shareable interfaces. Each application obtains a contract that specifies methods of shared interfaces that this application provides to and calls from other applications, desired security policy on usage of its services and functionally necessary services. This contract is verified on device to be compliant with the security policy provided by all the stakeholders together.

Transitive control flow: This approach provides an on-device verification of applet collusions absence. That is, when one applet can invoke methods of other applets not only directly, but also through some malicious or buggy applets' methods.

Global policy: This approach allows enforcing an on-device policy that is global *i.e.* all applets on the card should obey it. This policy expresses forbidden sequences of method invocations.

Non-interference: This technique improves information flow (non-interference) on-device verification by addition of specific information flow policies describing allowed and forbidden data exchange between security domains on GlobalPlatform.

The verification process is executed during the installation (or update) of an application. The verifier analyzes the bytecode of the applet in order to capture the details of information exchange (calls to other applets' methods, methods of the applet implementing shareable interfaces, possibly details on information exchange between variables) and checks if these details are compliant with the security policy of the smart card system.

Specific details of these techniques can be found in the deliverable D6.3 [11] and the current deliverable. Thus, main research objective of WP6 (on-device) is information protection (by flow control) property verification in a software update change scenario (for POPS case study).

6.3 Testing Work Package (WP7)

For POPS case study, one of research objectives of WP7 is to validate information protection by access control and by flow control verification during specification evolution change scenario. WP7 focuses on access control verification (by testing), ensuring that protected information cannot be accessed without acquiring specific rights. Correctness of access control implementation (with respect to GlobalPlatform specification) is achieved by two following kinds of tests:

- tests focusing on application with correct access rights;
- tests focusing on application without access rights.

In the second category, WP7 has also two types of tests: tests based on applications that never had access rights and tests based on applications that have lost access rights.

WP7 provides tests to validate the access control to the specific security services provided by a security domain. In particular, on GlobalPlatform an application can only communicate with the external world through its associated security domain.

WP7 also provides tests to validate the access control for association and extradition processes. WP7 ensures that an application (or a security domain) cannot be associated with a security domain without specific rights that are expressed as security domain owner's cryptographic keys or GP-specific privileges. This is a part of verification of the security domains hierarchy consistency that is required for information protection property.

Another direction of WP7 work is testing correctness of application installation process. The test model takes into account all behaviors of the install command:

- Nominal case if status word = 9000;
- Error case: more than 100 error cases are modeled in the test model by 11 status words extracted from the GP specification decomposed into 4 dedicated cases on `install` command [12, Sec. 11.5.3.2] and 7 general error cases [12, Sec. 11.1.3].

We note that WP7 does not provide full testing of correctness of smart card platform implementation as it is not in the scope of the project. WP7 validates the implementation with respect to some part of GP specification.

Details of the testing techniques can be found in the deliverables D7.1 [7], D7.2 [6] and D7.3 [8].

6.4 Threat Scenarios for Information Protection Property

Before discussing collaborations between WP6 and WP7 first we identify possible threat scenarios.

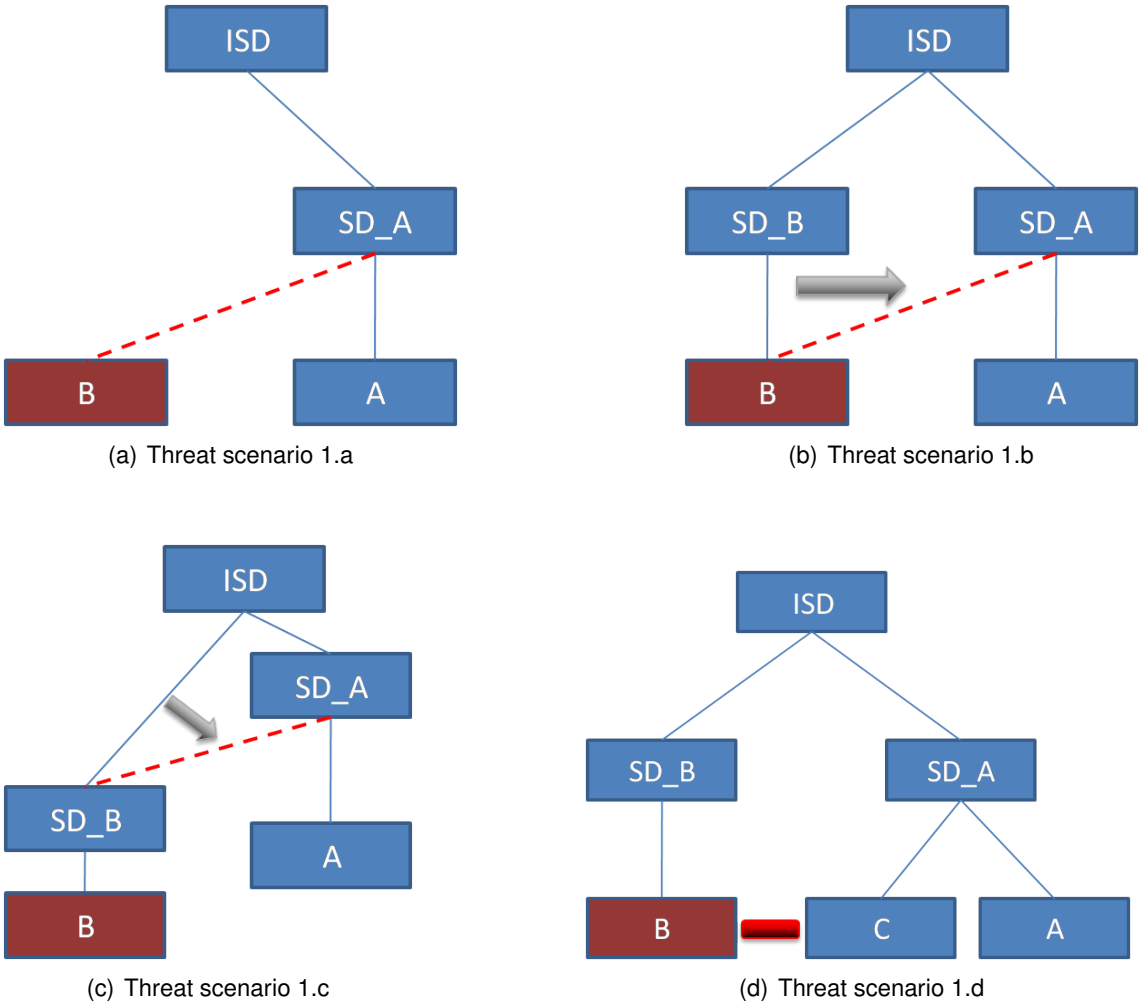
As assets (data that should be protected) we consider data and methods of applications and security domains GP-specific methods (like encryption and decryption). Further we will discuss how these assets can be accessed illegally.

Our attacker model is a malicious application that tries to be installed on a smart card (or updated) in order to get access to assets of honest stakeholders. Access to these assets is protected by security policy and correct (with respect to Java Card and GlobalPlatform specifications) smart card implementation. For specification of threat scenarios we assume that a honest application A is installed on the card and is associated

with a honest security domain SD_A. Assets in our threat scenarios are data of application A and GP-specific services of SD_A. A malicious applet B (provided by a stakeholder different from the owner of application A) tries to get an access to these assets. We do not consider in threat scenarios applications with certain GP privileges that allow card content management.

6.4.1 Threat Scenario 1: Illegal Access to Security Domain Services

A malicious application B tries to get access to the GP-specific services that are provided by security domain SD_A by associating B with SD_A directly or indirectly. It can be achieved by the following attacks:



(a) Threat scenario 1.a

(b) Threat scenario 1.b

(c) Threat scenario 1.c

(d) Threat scenario 1.d

Figure 2: Threat scenario 1

Scenario 1.a

Description of scenario: Application B tries to be installed into security domain SD_A. Figure 2(a) depicts a possible files hierarchy associated to this scenario. The red slash line represents the addition of the application to the hierarchy after the attempt.

Scenario 1.b

Description of scenario: Application B is installed on the platform and is associated with the security domain SD_B, which belongs to an owner of application B. Then application B tries to be extradited from SD_B to SD_A. Figure 2(b) depicts a possible files hierarchy associated to the scenario. The black arrow between blue line and red slash line represents the extradition attempt.

Scenario 1.c

Description of scenario: Application B is installed on the platform and is associated with the security domain SD_B. Then security domain SD_B tries to be associated with the security domain SD_A or one of its sub-domains. Figure 2(c) depicts a possible files hierarchy associated to the scenario. The black arrow between blue line and red slash line represents the extradition attempt.

Scenario 1.d

Description of scenario: Application B is installed on the platform and is associated with the security domain SD_B. B cooperates with a malicious or buggy application C that is associated with SD_A, for example, due to an agreement between the providers of SD_A and C. Application C can provide an access to its services to application B and be a mediator in the interaction between SD_A and B. For instance C can implement a service that will receive as a parameter some data of application B and invoke a GP-specific service of SD_A with this data. Figure 2(d) depicts a possible files hierarchy associated to the scenario. The red tube represents the communication attempt.

If one of the attacks of Scenario 1 succeeds, B obtains access to the GP services of security domain SD_A.

6.4.2 Threat scenario 2: Illicit Information Flow

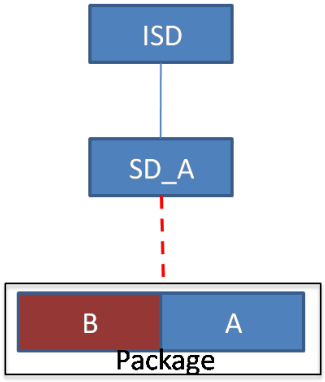
A malicious application B tries to get access to application A. On Java Card, applications from the same package can have access to any data of each other. Applications from different packages can only have access to methods that are defined in specific interfaces that extend `Shareable`. Further we discuss more precisely the threat scenarios of this kind.

Scenario 2.a

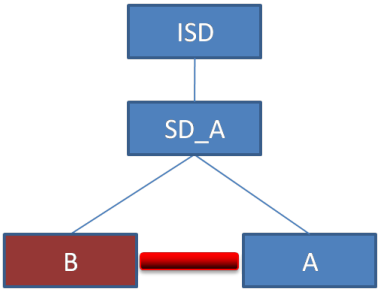
Description of scenario: Application B tries to be installed in the package of application A. On Java Card this can be done only if applications A and B are installed simultaneously, meaning the installation process is performed by the same off-card entity. If B succeeds, it obtains full access to all data of application A. Figure 3(a) depicts a possible files hierarchy associated to the scenario. The red slash line represents the installation attempt.

Scenario 2.b

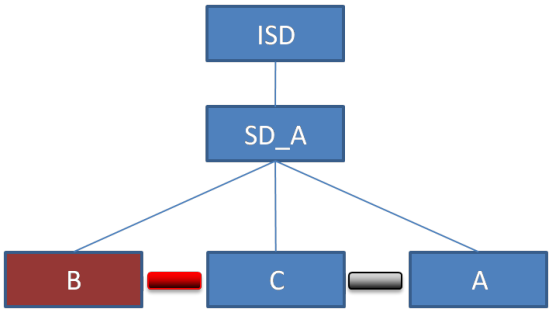
Description of scenario: Application B is installed into some package, which is different from A's package. It receives a reference to an object of application A that implements



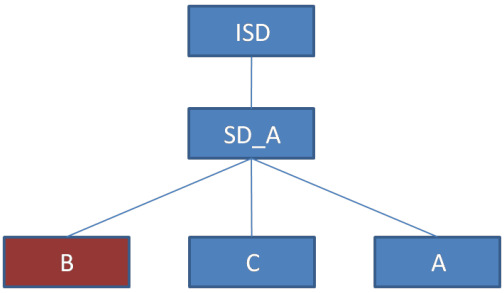
(a) Threat scenario 2.a



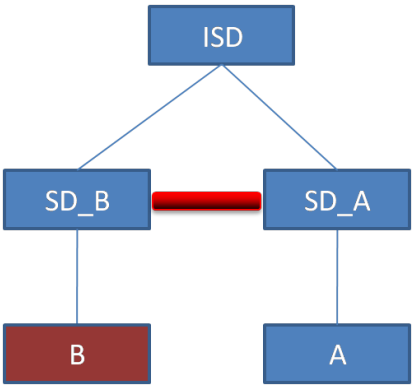
(b) Threat scenario 2.b



(c) Threat scenario 2.c



(d) Threat scenario 2.d



(e) Threat scenario 2.e

Figure 3: Threat scenario 2

`Shareable` interface. It tries to directly invoke one of the services of application A defined in this interface, and this invocation is forbidden to him by security policy of application A. If B succeeds, it obtains access to the service. Figure 3(b) depicts a possible files hierarchy associated to the scenario. The red tube represents the communication attempt.

Scenario 2.c

Description of scenario: Application B is installed into some package, which is different from A's package. It tries to invoke a shared method of an application C, which in turn will invoke a service of application A. If C does not have an authorization to call a service of A, considered scenario coincides with scenario 2.b. Thus we consider the case when C does have a permission to invoke services of A. This attack scenario also includes the case when B starts invoking a cascade (transitively or/and in sequence) of service calls with at the end a call to a service of application A. In this case we assume that all direct calls in this sequence are authorized by the security policy, but the transitive call of B to invoke service of A is forbidden. If B succeeds, it obtains access to the service of A. Figure 3(c) depicts a possible files hierarchy associated to the scenario. The red tube represents the communication attempt.

Scenario 2.d

Description of scenario: Application B is installed into some package, which is different from A's package. In cooperation with another application(s) B tries to produce some sequences of method calls that are forbidden by the smart card platform global security policy, which is provided by the card vendor or the card issuer. Some (sequences of) method calls could in fact lead the system to an undesired state such as `CARD_LOCKED`. Forbidden (sequences of) method calls can also involve methods of applications themselves. For instance, we may want to allow application A to access its security domain keys (encrypt and decrypt methods of the GP API) but avoid further (in sequence) invocations of methods in other security domains through methods of shared objects and thus avoid the leak of security domain keys. Figure 3(d) depicts a possible files hierarchy associated to the scenario. Communications are done by usual Java Card mechanisms.

Scenario 2.e

Description of scenario: Application B is installed into some package, which is different from A's package and is associated to security domain SD_B. B tries by a series of method invocations to fetch some secret data from application A. Non-interference security policy declares that all secrets sharing from SD_A to SD_B are forbidden by the provider of application A. If B succeeds it obtains an illegal access to secret data of A. Note that in this threat scenario we do not consider an attack when B tries to be associated to SD_A in order to obey the non-interference policy, because this attack is covered in the Scenario 1. Figure 3(e) depicts a possible files hierarchy associated to the scenario. The red tube represents the communication attempt.

If one of the attacks of Scenario 2 succeeds, application B obtains an illegal access to data of application A. Though, for example direct control flow attack 2.b is a subset of transitive control flow attack 2.c or a non-interference attack is a transitive control flow attack with a higher level of precision, we consider them separately. For details on

benefits of each verification techniques (that protects against certain attack scenario) an interested reader may refer to the deliverable D6.3 [11].

6.4.3 Fulfillment of Identified Threat Scenarios

On smart cards access to such assets as application data and services or security domain services should be implemented in accordance with specifications and security policies of the stakeholders. We have specified in the threat scenarios possible ways to obtain access to these assets (under certain assumptions). We claim that the Scenarios 1 and 2 fulfill the information protection security property required by the POPS case study (with respect to the current state of the art) assuming that:

- The Java Card implementation is correct (with respect to the specification) and robust (against physical attacks);
- The applications are correctly identified (by its AID), successfully bytecode-verified and do not contain native code (*i.e.* A, B, C are well-formed Java Card applets).

6.5 Protection against Threats

In this section we will show how WP6 and WP7 together provide protection against identified threats.

6.5.1 Protection against threat scenario 1

Scenarios 1.a, 1.b and 1.c are investigated by WP7, which provides test suites for smart card platform implementation. These attacks can be successful in two cases: implementation of smart card platform is inconsistent with the specifications or owner of application B has got a cryptographic key to the security domain SD_A. WP7 provides tests for correctness of installation process and ensures that the application B can be associated with the security domain SD_A if and only if it has a specific cryptographic key (as an access right token) for this security domain (correctness of association and extradition). We assume that this may happen if and only if B belongs to the owner of SD_A or the owners of B and SD_A had made a business agreement. Full description of testing methodology is provided in D7.1, D7.2 and D7.3 [7, 6, 8].

Scenario 1.d cannot be fully covered by WP7, though it is related to the access control property. WP7 ensures that installation process is correct and application A is correctly associated with SD_A. But protection against illegal access of B to services of SD_A in this threat scenario can be provided by verification methods of WP6. This can be performed with different levels of precision. The GP-specific services of security domains on Java Card level are implemented as methods of `SecureChannel` interface provided by the security domain. Application A can obtain a reference to an object of SD_A that implements `SecureChannel` interface through a call to a GP-specific method `GPSystem.getSecureChannel()`. Transitive information flow verification techniques can capture forbidden transitive service invocations from application B to SD_A. Another possibility can be declaration of `GPSystem` methods calls as specific set of methods and verification with transitive control flow techniques that there is no illegal sequence of service invocations from B to all `GPSystem` methods. More precise verification can be achieved by adapting a non-interference technique of WP6. In this case one can check

that there is no information flow between application B and SD_A. Consequently, protection from threat scenario 1.d can be implemented only by testing and verification work packages working together.

6.5.2 Protection against Threat Scenario 2

In fact we do not consider threat scenario 2.a as important because in this case (under assumption that Java Card implementation is correct) application B is necessarily an applet of A's owner or the owners of applications A and B had made a business agreement. Scenarios 2.b, 2.c, 2.d and 2.e are investigated by WP6, which provides assurance that if some application is installed on the card (or was updated) then it will respect security policies of all the stakeholders. Full specification of verification techniques is provided in the deliverable D6.3 [11] and the current deliverable.

WP6 uses as important underlying assumption correctness of smart card platform implementation, which is partially validated by testing in WP7. This is a particularly important requirement for WP6, because verification work package makes certain assumptions on the data exchange processes between applications. If implementation of Java Card or GlobalPlatform is not correct with respect to the specification, verification process cannot be considered trusted. Particularly, correctness of application installation process is a very important requirement for WP6.

We make a remark here that especially scenario 2.e is tightly related to the scenarios 1.a – 1.c. Namely, WP6 relies on WP7 to ensure consistency of security domains hierarchy. If attack scenarios 1.a – 1.c are possible on some smart card platform due to faulty implementation (related to application and security domain association and extradition processes), then attack scenario 2.e is also possible no matter what verification results are. So we may conclude that, again, only collaboration of verification and testing work packages can provide full protection against attack scenario 2.

Full attacker model for Scenarios 1 and 2 can be provided formally. Allowed or forbidden access to assets can be formalized, for example, as a predicate, which can be derived from security policies of the card and specifications. We do not construct a full formal model and we do not prove any theorems related to this model as it is out of the scope of the project.

6.6 Conclusion

WP6 and WP7 can explore separate verification techniques for protection against threats of the Scenarios 1 and 2, except for the 1.d. In fact, the interest of a connection between these two work packages is that each hypothesis proposed by one work package is tackled by the other. So the completeness of validation can only be achieved if the work packages work together as proposed in the SecureChange process.

7. Conclusion

In this deliverable we have finalized the four different models originally introduced in the deliverable D6.3 [11] for loading-time verification of security properties related to information protection on constrained Java-based systems.

The direct control flow model and the transitive control flow model, respectively presented in the Chapter 2 and in the Chapter 3, have the common objective to offer an application a way to control access to its shared services. These two models follow different approaches and provide different features. On one hand, the direct control flow model follows the Security-by-Contract approach and deals with inter-application functionality requirements. On the other hand, the transitive control flow model is a static verfication analysis completely done on-device, so it is well appropriate for autonomous open systems able to load code from untrusted entities, and it is sensitive to application collusion. All types of changes (addition/removal of application, modification of control flow policy, modification of domains hierarchy) are now supported by both models in an incremental way, and we have shown that this differential approach is more efficient than complete reverification of the system while providing the same security guarantees.

The last two models are following the proof-carrying-code paradigm. The global policy model presented in the Chapter 4 aims to detect forbidden sequences of methods calls defined at the system level. This model now supports installation of application, removal of application and modification of domains hierarchy in an efficient incremental way with the same security guarantees as complete reanalysis of the system. The non-interference model in the Chapter 5 aims to detect illicit flows of data between domains of the system. All types of changes are now supported by this model in an efficient incremental way with the same security guarantees as a complete re-verification of the system.

All these techniques require integration of algorithms in the Java-based core system itself in order to be able to reject at installation a new application that would break the security of the system. These algorithms must also be interleaved in the bytecode loading process of the underlying system to ensure that the verification cannot be bypassed. Consistency of the domains hierarchy is also a crucial issue we assume to be verified in the underlying system. This last assumption is fortunately tested on GlobalPlatform by WP7, as described in the Chapter 6.

All on-device WP6 techniques have different level of complexity, and thus different resources requirements. The last year effort of WP6 will consist in the development of a proof-of-concept implementation of the presented techniques by increasing complexity order, and the evaluation of their adequacy and applicability to SecureChange use cases by industrial partners.

References

[1] Java Card specifications. http://java.sun.com/javacard/.

[2] O. Alliance. Osgi service platform core specification. Specification Release 4, Version 4.2, 2009.

[3] A. Armenteros, B. Chetali, E. Chiarani, M. Felici, V. Meduri, A. Tedeschi, F. Paci. D1.2: Applicability of SecureChange technologies to the scenarios. SecureChange EU project public deliverable, www.securechange.eu, 2010.

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

[5] 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.

[6] F. Bouquet, F. Dadeau, S. Debricon, E. Fourneret, J. Julliand, P. Masson, P. Paquellier, M. Felderer, B. Legeard, J. Botella, Z. Micksei, B. Agreiter, B. Chetali, Q. Nguyen, A. Pacheco, E. Chiarani, F. Massacci. D7.2: How to integrate evolution into a model-based testing approach. SecureChange EU project public deliverable, www.securechange.eu, 2010.

[7] F. Bouquet, F. Dadeau, S. Debricon, P. Masson, Z. Micksei, D. Varro, B. Agreiter, M. Felderer, B. Legeard, J. Botella, O. Bussenot, E. Jaffuel, C. Grandpierre, J. Hamot, A. Masson, E. Chiarani, F. Paci, F. Massacci, J. Jurjens. D7.1 evaluation of existing methods and principles. SecureChange EU project public deliverable, www.securechange.eu, 2009.

[8] F. Bouquet, F. Dadeau, S. Debricon, P. Masson, Z. Micksei, D. Varro, B. Agreiter, M. Felderer, B. Legeard, J. Botella, O. Bussenot, E. Jaffuel, C. Grandpierre, J. Hamot, A. Masson, E. Chiarani, F. Paci, F. Massacci, J. Jurjens. D7.3: A model-based testing approach for evolution. SecureChange EU project public deliverable, www.securechange.eu, 2010.

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

[10] A. Fontaine, S. Hym, I. Simplot-Ryl. On-Device Control Flow Verification for Java Programs. Proceedings of the 3rd International Symposium on Engineering Secure Software and Systems (ESSoS11), Lecture Notes in Computer Science, pages to appear, Madrid, Spain. Springer-Verlag, 2011.

[11] A. Fontaine, S. Hym, I. Simplot-Ryl, O. Gadyatskaya, F. Massacci, F. Paci, J. Jurgens, M. Ochoa. D6.3 compositional technique to verify adaptive security at loading time on device. SecureChange EU project public deliverable, www.securechange.eu, 2010.

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

[13] S. Microsystems. Application programming interface specification. Java Card^{TM} platform, version 2.2.2. Specification 2.2.2, Sun Microsystems, 2006.

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

[15] S. Microsystems. Virtual machine specification. Java Card^{TM} platform, version 2.2.2. Specification 2.2.2, Sun Microsystems, 2006.

[16] Oracle. http://www.oracle.com/technetwork/java/javase/documentation/index.html. Available on the web.