D6.5: PROTOTYPES COMPANION REPORT

Michela ANGELI (UNITN), Arnaud FONTAINE (INR), Olga GADYATSKAYA (UNITN), Eduardo LOSTAL (UNITN), Fabio MASSACCI (UNITN), Isabelle SIMPLOT-RYL (INR)

Document Information

Document Number	D6.5
Document Title	Prototypes companion report
Version	1.0
Status	Final
Work Package	WP6
Deliverable Type	Prototype
Contractual Date of Delivery	M36
Actual Date of Delivery	M36
Responsible Unit	INR
Contributors	INR, UNITN
Keyword List	Verification, prototype, on-device
Dissemination	PU

Document change record

Version	Date	Status	Author (Unit)	Description
0.1	2011/11/30	Working draft	A. Fontaine (INR)	First version
0.2	2011/12/08	Working draft	A. Fontaine, I. Simplot-Ryl (INR)	Added description of the EVE-TCF prototype
0.3	2011/12/26	Working draft	O. Gadyatskaya, E. Lostal, F. Massacci (UNITN)	Added description of the S×C prototype, structured Section Introduction tentatively
0.4	2011/12/27	Working draft	A. Fontaine (INR)	Added introduction, conclusion and executive summary
0.5	2011/12/29	Working draft	O. Gadyatskaya (UNITN)	Modifications to chapter on S×C prototype and appendix, new appendix on Java Card added, minor changes to the introduction/conclusion
0.6	2011/12/29	Working draft	A. Fontaine, I. Simplot-Ryl (INR)	Typos, layout
0.7	2012/01/24	Working draft	M. Angeli (UNITN), A. Fontaine (INR)	First quality check completed-minor remarks
1.0	2012/01/31	Final version	A. Fontaine (INR), O. Gadyatskaya (UNITN), M. Angeli (UNITN)	Second quality check and finalization

Executive summary

During the two first years of the SecureChange project, the on-device verification part of WP6 has proposed four models for on-device information protection (D6.3 and D6.4), focusing on strongly constrained devices such as smart-cards. During the last year of the project, two of these models have been implemented and applied to the realistic POPS case study scenario. The deliverable D6.5 describe these implementations and discuss their integration in a legacy environment: JavaCard 2.x smart cards with GlobalPlatform.

Index

Document change record 2

Executive summary 3

Introduction 7

1 EVE-TCF : Transitive Control Flow Prototype for Smart Cards 9
- 1.1 Overview of EVE-TCF . . . 9
  - 1.1.1 From development to deployment of JavaCard applications . . . 9
  - 1.1.2 Considerations for on-device integration . . . 11
- 1.2 Off-device embedding of transitive control flow policies . . . 11
  - 1.2.1 The DSL language for transitive control flow policies . . . 11
  - 1.2.2 The `convert` tool . . . 14
  - 1.2.3 The TCF component . . . 14
- 1.3 On-device verification of embedded policies . . . 15
  - 1.3.1 Installation of a new package . . . 17
  - 1.3.2 Removal of an installed package . . . 17
- 1.4 On-device management of policies . . . 18
- 1.5 The POPS case study . . . 20

2 The Security-by-Contract Prototype 23
- 2.1 An Overview of the SxC Prototypes . . . 23
  - 2.1.1 Embedding Contracts . . . 23
  - 2.1.2 The S×C Workflows on Device . . . 24
- 2.2 The S×C Embeddable Prototype Architecture . . . 25
  - 2.2.1 The Claim Checker Algorithm . . . 25
  - 2.2.2 The On-card Policy Store . . . 26
  - 2.2.3 The Policy Checker . . . 27
- 2.3 The Prototype for Testing . . . 27

Conclusion 29

A The Device Architecture 31

B The Security-by-Contract Prototype Details 33
- B.1 Embedding Contracts . . . 33
  - B.1.1 Application Contract . . . 33
  - B.1.2 The Contract Delivered on the Card . . . 34
  - B.1.3 Contract Population . . . 34
- B.2 An Example . . . 35
- B.3 Using the S×C Prototype for Testing . . . 36

Introduction

This document is the companion report of the prototypes deliverable D6.5 for on-device information protection. The purpose of this report is to explain the implementations of the *transitive control flow* model and the *security by contract* model (see deliverables D6.3 and D6.4) developed in WP6 of the Secure Change project. These prototypes have been especially designed to fit the requirements of JavaCard 2.x smart cards, the target devices of the POPS case study of the project, but also the most constrained devices of the whole project.

In collaboration with our industrial partner Gemalto/Trusted Labs, we early agreed on some architecture for concrete integration on card. Gemalto/Trusted Labs provided a set of API, called apiobc for mandatory interactions with the operating system, and we suggested a non-invasive way to store meta-data on-device thanks to a JavaCard applet. The architecture of a real platform is overviewed in Appendix A and it serves as a reference for discussions of the prototype integration.

Integration of each prototype on target devices as well as application to the POPS case study are detailed in Chapter 1 for EVE-TCF , the *transitive control flow* prototype, and in Chapter 2 for S×C, the *Security-by-Contract* prototype developed for the direct control flow control. Both prototypes have been released to the industrial partner Gemalto/Trusted Labs for evaluation on an actual JCRE (PC simulator or/and real integrated circuit). The prototypes are expected to be evaluated in terms of memory footprint and compliance with the JCRE. Extensive functional testing of the prototypes was conducted by INR-Lille and UNITN.

1. EVE-TCF : Transitive Control Flow Prototype for Smart Cards

The purpose of this chapter is to describe EVE-TCF , the implementation for JavaCard smart card with GlobalPlatform of the transitive control flow model (see Chapter 4 of the deliverable D6.3 and Chapter 3 of the D6.4). EVE-TCF is a prototype that demonstrates the practicability of the transitive control flow model on highly constrained devices that are smart cards. EVE-TCF has been released to industrial partners (Gemalto/Trusted Labs) for concrete evaluation purposes such as CPU consumption and volatile/persistent memory usage in a real environment.

1.1 Overview of EVE-TCF

Basically, EVE-TCF is a set of three executables:

- `convert` to convert and to embed transitive control flow policies written in a simple DSL language as a *Custom Component*, called the TCF component, into CAP files;
- `extract` to extract the content of the TCF component of a CAP file and to output it as a transitive control flow policy written in the DSL language;
- `simu` to simulate (off-device) the on-device (un)loading process of CAP files and the management of GlobalPlatform security domains (creation, deletion).

The `simu` executable integrates the verification procedure at loading-time of control flow policies, as it is depicted in the deliverables D6.3 and D6.4. This executable is actually a rough environment to test the code responsible for the verification of control flow policies devoted to be integrated in the smart card operating system by Trusted Labs/Gemalto. It permits easier off-device experiments, as well as pre-deployment experiments by application vendors.

EVE-TCF is released with a JavaCard package containing the applet `IFCInstallerApplet` responsible for storing repositories of control flow policies on-device. Interactions between the on-device verifier written in C and the `IFCInstallerApplet` are described in Section 1.3.

EVE-TCF is also released with an Eclipse plugin to permit an easy use of its core executables. This plugin permits easy detection of exceptions to control flow policies directly from Eclipse GUI environment, and incremental syntax-coloring of both the DSL language for control flow policies and the DSL language for simulation scripts.

1.1.1 From development to deployment of JavaCard applications

The Figure 1.1 displays the process to generate, from Java source code, a JavaCard *application(s) package* (*i.e.* CAP file) with embedded transitive control flow policies ready to be deployed. The gray steps are the “normal” steps of the JavaCard development process, while the green step corresponds to the new additional mandatory step that embeds a control flow policy (`.tcf` file(s)) in the application(s) package. The “normal” steps are not changed, thus standard tools from the SDK can be used, so as

home-made tools as long as they produce a CAP file compliant with the JCVM 2.x specifications. The generated CAP file must then be submitted with its control flow policy to the `convert` of EVE-TCF that produces a CAP file with the TCF component. Some deployment strategies also involve an additional signing step of CAP files to ensure their integrity and their origin. In this case, the signing operation must be completed on the CAP file generated by `convert` of EVE-TCF .

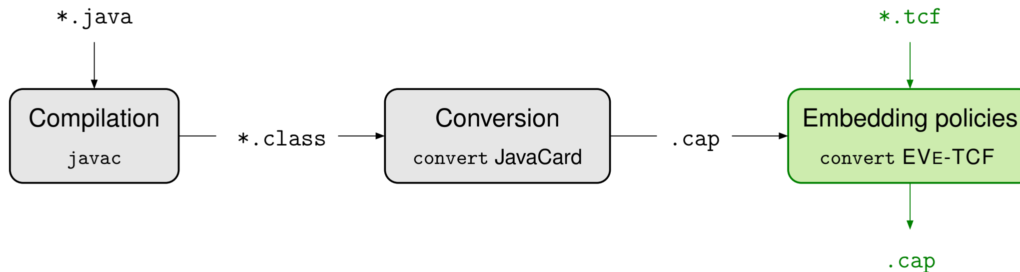


Figure 1.1: Schema of the off-device process for the development of JavaCard applications. The green parts of the process correspond to the steps introduced by the use of EVE-TCF .

The TCF component is fully compliant with the formatting rules of the CAP file format described in the JCVM 2.x specifications. It can thus be deployed on smart cards with EVE-TCF verification code integrated, but also on other smart cards where the TCF component will simply be ignored. The Figure 1.2 displays the deployment steps of a JavaCard application(s) package. The gray steps are the “normal” steps of the JavaCard deployment process, while the green steps correspond to the new additional mandatory steps to verify control flow policies. The “normal” steps are not changed, so that the impact on existing JCVM implementations and smart card operating systems is very limited. In the implementation of EVE-TCF for SecureChange, the *verification* step of control flow policies has been developed to occur after the complete on-device *loading* of the CAP file, so that all its components can be accessed in random way (*i.e.* not linearly at each loading of CAP file chunk), and before its *linking* with already loaded code, in order to avoid rewriting of `invoke` instructions and code optimization that could lead to incomplete and/or erroneous verification.

The Figure 1.2 also depicts the steps for removal of an installed package. The gray steps are the normal steps of package removal, and the green step corresponds to the new additional mandatory step to update the repositories of control flow policies managed by EVE-TCF on package removal. In the implementation for SecureChange, the smart card operating system first decides if a package can be removed, then it notifies EVE-TCF of the upcoming removal and finally commits the removal. When EVE-TCF is notified, it updates its repositories of control flow policies in an irreversible way, as described in the deliverables D6.3 and D6.4.

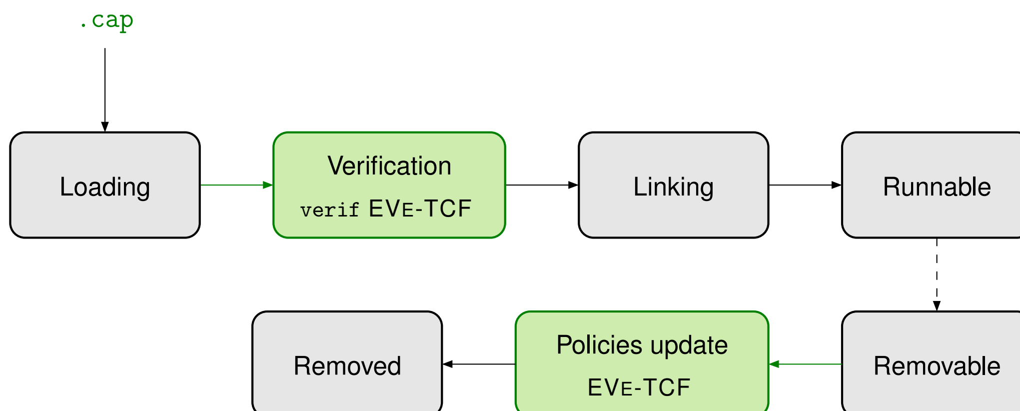


Figure 1.2: Schema of the on-device process for the deployment of JavaCard applications. The green parts of the process correspond to the steps introduced by the use of EVE-TCF .

1.1.2 Considerations for on-device integration

In practice, the removal of an installed package on JavaCard 2.x is possible if and only if no other installed package depends on it. In addition, it is not possible to install some package with unsolved dependencies. These features strongly simplify the implementation of the transitive control flow model presented in the deliverables D6.3 and D6.4. Actually, only one repository is needed on-device for storing control flow policies, and policies update on removal of an installed package simply consists in removing its the control flow policy from the repository without any code or policies reverification.

As requested by the industrial partner, the on-device verifier allows to consider some installed packages as *safe* and will not try to verify control flow policies on the classes, interfaces and methods they define. Concretely, this is implemented in EVE-TCF by setting up a threshold (`NO_POLICY_FOR_PACKAGE_INDEX_LT` variable fixed at compilation-time) on internal package identifiers managed on-device by the JCVM. As result, the invocation, as well as overriding and redefinition of control flow policy, of a method defined in a safe package is not submitted to control flow policy check.

1.2 Off-device embedding of transitive control flow policies

Control flow policies are expressed using a DSL (Section 1.2.1). The `convert` tool (Section 1.2.2) translates policies written in this language into a TCF component (Section 1.2.3) suitable to embedded into the CAP file containing the corresponding code. Basically, the `convert` tool takes as inputs a control flow policy and a CAP file, and produces a new CAP file containing the same content as the input one plus the new TCF component. The Directory component is also updated to reflect this change, as requested by the CAP file format of the JCVM specifications.

1.2.1 The DSL language for transitive control flow policies

The DSL for describing transitive control flow policies is very simple and flexible. The complete BNF grammar of this language is given in Figure 1.3.

Domains aliases

The user can define textual aliases corresponding to sets of security domains (at least one per alias) given their corresponding AIDs using the following syntax,

```
/* an alias to one domain */
domain AliasDomain1 { 0:1:2:3:4 }

/* an alias to two domains */
domain AliasDomain2 { 0x0:0x1:0x2:0x3:0x4, AliasDomain1 }
```

where characters `{` and `}` are optional. The prefix `0x` is also optional: all values in AIDs are assumed to be hexadecimal values. As it is shown in this example, comments in the language are starting with `/*` and are ending with `*/`. Comments can contain any character, including the newline character, as well as nested comments.

Policy of methods

EVE-TCF relies on fully qualified names of classes, interfaces and methods, but also on (classes, interface and method) tokens, as described in the JCVM specifications, to designate classes, interfaces and methods. For methods defined in a package to “annotate”, at least fully qualified names should be used as they are non-ambiguous and less subject to changes across (re-)compilations and/or evolution of the source code; tokens are only mandatory to designate methods, classes or interfaces inherited from other packages, or to provide a complete policy that can be transmitted to a third party that will use the package.

```
           <policy> ::= "package" <java_fqn> <policy>
                      | "domain" <domain_alias> <domain_list> <policy>
                      | "class" <class_policy> <policy>
                      | "interface" <class_policy> <policy>
                      | ""

     <class_policy> ::= <opt_token> <java_fqn> "{" <method_list> "}"

      <method_list> ::= <opt_static> <method_policy> <method_list> | ""

       <opt_static> ::= "static" | ""

    <method_policy> ::=     <token>                              ":" <policy_content> <semicolons>
                      | <opt_token> <java_method> <opt_java_desc> ":" <policy_content> <semicolons>

   <policy_content> ::= <domain_list> | "*" | "top" | "all" | "any" | ""

      <domain_list> ::= <domain_list_aux> | "{" <domain_list_aux> "}"

  <domain_list_aux> ::= <aid_or_alias> <separators> <domain_list_aux>
                      | <aid_or_alias>

       <semicolons> ::= ";" <semicolons> | ";"

       <separators> ::= "," <separators> | ","

     <aid_or_alias> ::= <aid> | <domain_alias>

              <aid> ::= <byte> <aid_colon> | <byte> <aid_string>

        <aid_colon> ::= ":" <byte> <aid_colon> | ""

       <aid_string> ::= " " <byte> <aid_string> | ""

        <opt_token> ::= <token> | ""

            <token> ::= "0x" <one_or_two_hexa>

             <byte> ::= "0x" <one_or_two_hexa> | <one_or_two_hexa>

  <one_or_two_hexa> ::= <hexa> | <hexa> <hexa>

             <hexa> ::= "a" | "b" | "c" | "d" | "e" | "f" | <numeric>

         <java_fqn> ::= <java_name> <java_fqn_aux>

     <java_fqn_aux> ::= "." <java_name> <java_fqn_aux>
                      | "/" <java_name> <java_fqn_aux>
                      | ""

      <java_method> ::= "<init>" | <java_name>

    <opt_java_desc> ::= "(" <opt_java_desc_aux> ")" <java_desc_aux> | ""

<opt_java_desc_aux> ::= <java_desc_aux> | ""

    <java_desc_aux> ::= <java_desc_aux_char> <java_desc_aux>
                      | <java_desc_aux_char>

<java_desc_aux_char> ::= <alpha_> | <numeric> | "/" | "." | ";"

        <java_name> ::= <alpha_> <java_name_aux>

    <java_name_aux> ::= <alpha_> <java_name_aux> | <numeric> <java_name_aux> | ""

     <domain_alias> ::= <java_name>

           <alpha_> ::= "a" | "b" | ... | "z" | "_"

          <numeric> ::= "0" | "1" | ... | "9"
```

Figure 1.3: BNF grammar of the DSL language for transitive control flow policies.

CAP files submitted to EVE-TCF tools **must** embed the Debug component in order to perform classes/interfaces/methods names resolution. Output CAP files do not embed the Debug component unless it is specified to the `convert` tool by the option `--keep-debug`.

The following syntax permits to define control flow policies of the method `debit` of the interface `IEPurseService` defined in the package `com.gemalto.securechange.epurse`:

```
package com.gemalto.securechange.epurse
domain TransportDomain 0:1:2:3:4
interface IEPurseService {
    debit : TransportDomain;
}
```

Note that the security domain where the code will be installed is implicitly added to all policies, so it is not mandatory to mention it. Of course, a control flow policy can be empty (no AID or alias).

The `package` instruction permits to simplify `interface` or `class` definitions, but is not mandatory. Fully qualified names can be used directly, which in this example gives:

```
interface com.gemalto.securechange.epurse.IEPurseService {
    debit(S)V : TransportDomain;
}
```

The method description can be omitted if and only if it is not overloaded (*i.e.* other methods with same name but different descriptions). As it is the case in the previous example, we can simply write:

```
interface IEPurseService {
    debit : TransportDomain;
}
```

Class, interface and method tokens should be given in control flow policies if and only if both the class and the method tokens are not `0xff`, but this is not mandatory. These tokens are however mandatory if they refer to methods defined outside the analyzed package as these class, interface and method names are not embedded in the Debug component. Below is the previous example where tokens are given:

```
interface 0x0 IEPurseService {
    0x0 debit : TransportDomain;
}
```

As static methods of classes have their own tokens namespace, it is mandatory to use the `static` keyword to designate a static method, even if you don't specify its token value:

```
class 0x1 EPurseService {
    static <init>(Lcom/gemalto/securechange/epurse/EPurseApplet;)V : ;
}
```

By default, when the policy of a method is not given, the `convert` tool tries to inherit the policy of the super method (if it exists), or from an interface (if the method is the implementation of a method's interface). If no policy is found for it, the tool will infer the smallest control flow policy that satisfies policies of callers and callees, according to the call graph of the package.

There exists a special definition for the control flow policy of a method that permits it to be called from *any domain*. Instead of a set of AIDs (or aliases), one of the following equivalent symbols can be used to specify the *any domain* policy:

```
*    top    all    any
```

1.2.2 The `convert` tool

As described in the Section 1.1.1, the `convert` tool of EVE-TCF requires two input files:

- the CAP file (ex: `input.cap`) to analyze, including the Debug component to permit resolution of classes, interfaces and methods names;
- a policy file (ex: `policy.tcf`) written in the DSL language described in the Section 1.2.1 that contains the control flow policy to be attached to the methods of the classes and interfaces defined in the CAP file.

Given these input files, the `convert` tool produces a new CAP file (ex: `output.cap`) with the same content plus the TCF component (Section 1.2.3) thanks to the following command:

```
convert --policy policy.tcf --output output.cap input.cap
```

The above command shows the two mandatory command line parameters to be specified, but `convert` admits other optional parameters:

- `--compress` compresses entries in the output CAP file using the 'deflate' routine (default is no compression, *i.e.* 'store');
- `--keep-debug` copies the Debug component of the input CAP file in the output CAP file (default is no);
- `--override` overrides the output CAP file if it already exists (default is error if the output CAP file already exists);
- `--lazy` disables the check of policies consistency (inheritance, overriding, invocation) within the package (default is enabled);
- `--system-policy spolicy.tcf` admits that `spolicy.tcf` is the control flow policy of the API and external classes and interfaces, which is especially useful to properly deal with inheritance and overriding rules;
- `--system-calls` in combination with the `--system-policy` parameter or `--top` parameter, takes into account the policies of external classes to infer implicit control flow policies of methods defined in the analyzed package (default is to ignore them)
- `--top` the *any domain* policy is set by default to external methods without a policy explicitly given.

1.2.3 The TCF component

The TCF component component embedded in CAP files by the `convert` tool of EVE-TCF is a *Custom Component* of the CAP file format (see JCVM 2.x specifications) with the data structures displayed on Figure 1.4. The fields of those structures have the following meaning:

- the `tcf_component` structure describes a TCF component:
  - `tag` contains the value of `TRANSITIVE_CONTROL_FLOW_TAG_CUSTOM_COMPONENT` (250), which permits to identify the TCF component;
  - `size` indicated the number of bytes in the `tcf_component` structure, excluding the `tag` and `size` items. The value of the `size` field must be greater than 0;
  - `domains_count` represents the number of entries in the `domains` field;
  - `domains` contains all the security domain AIDs mentioned in control flow policies that reference this field;

- class_policies_count represents the number of entries in the class_policies field;
- class_policies contains a class_policy entry for each class and each interface defined in this package;

- the cap_class_policy structure describes the control flow policy of a class or an interface:
  - classref contains the location (*i.e.* the offset) in the Class Component (see JCVM specifications) of the info structure corresponding to a class (or an interface) defined in this package;
  - class_token represents the class token of the current class (or interface), or 0xFF if the current class (or interface) has no token assigned;
  - method_policies_count represents the number of entries in the method_policies field;
  - method_policies maps to each method of the current class (or interface) its control flow policy;

- the cap_method_policy structure describes the control flow policy of a method:
  - bitfield is mask of modifiers used with a method with the following meaning:

Mask	0x80		0x40		0x20	
Value	0x80	is visible	0x40	is implemented	0x20	is static
	0x00	is not visible	0x00	is abstract	0x00	is not static

  - method_token represents the static method token or virtual method token or interface method token of this method if the method is visible according to bitfield;
  - method_offset represents a byte offset into the info item of the Method Component (see JCVM specifications) if the method is implemented (*i.e.* not an abstract method or a method definition in an interface) according to bitfield;
  - authorized_count represents the number of entries in the authorized field;
  - authorized contains indexes in the domains field of the tcf_component structure corresponding to the AIDs of security domains authorized to call directly or transitively this method;

- the aid structure describes an ISO AID:
  - length gives the number of bytes (from 5 to 16 included) in the value array;
  - value contains the ISO AID (see JCVM specifications) of a GlobalPlatform security domain.

1.3 On-device verification of embedded policies

On-device verification of control flow policies embedded in TCF component requires two modifications of the smart operating system/JCVM: a first one to trigger the verification at installation of new package (Section 1.3.1), and a second one to trigger update of repositories of policy on package removal (Section 1.3.2). The prototype version of EVE-TCF released to industrial partner does not implement modification of control flow policies on-device, and relies on the JavaCard applet IFCInstallerApplet to store repositories of policies. The main purpose of the two following sections is to explain how the operating system/JCVM interacts with the EVE-TCF verifier (native code) and the IFCInstallerApplet (JavaCard 2.x code) at installation-time of a new package (Section 1.3.1) and at removal-time (Section 1.3.2).

```
tcf_component {
    u1 tag;
    u2 size;
    u1 domains_count;
    aid domains[domains_count];
    u1 class_policies_count;
    cap_class_policy class_policies[class_policies_count];
}

cap_class_policy {
    u2 classref;
    u1 class_token;
    u2 method_policies_count;
    cap_method_policy method_policies[method_policies_count];
}

cap_method_policy {
    u1 bitfield;
    u1 method_token; /* present according to bitfield */
    u2 method_offset; /* present according to bitfield */
    u1 authorized_count;
    u1 authorized[authorized_count];
}

aid {
    u1 length;
    u1 value[length];
}
```

Figure 1.4: Data structures of the TCF component.

1.3.1 Installation of a new package

After a new CAP file is uploaded on-device, it must then be installed if and only if it is successfully checked by the EVE-TCF on-device verifier. The verification of a freshly uploaded package not yet installed follows the following steps also described on Figure 1.5:

1. the system calls the `get_policy` method of the `IFCInstallerApplet` to copy the repository of control flow policies of already installed packages in the APDU buffer at offset CDATA (see ISO7816);

2. if the copy succeeds (returns a non-null positive integer), the system calls the

   `verif_transitive_control_flow`

   function that processes the verification of the last loaded cap relying on APDU buffer content as its repository of verified policies; the verification uses a volatile RAM buffer of 255 bytes on-device to store temporary data that can be erased when verification is over;

3. if the verification succeeds (returns a non-null positive integer), then new package policy is stored by the verification procedure in the APDU buffer at the offset given in parameter;

4. the system calls the `update_policy` method of the `IFCInstallerApplet` that will add the new control flow policy to the repository of verified policies;

5. the TCF component is now useless and can thus be dropped be the system.

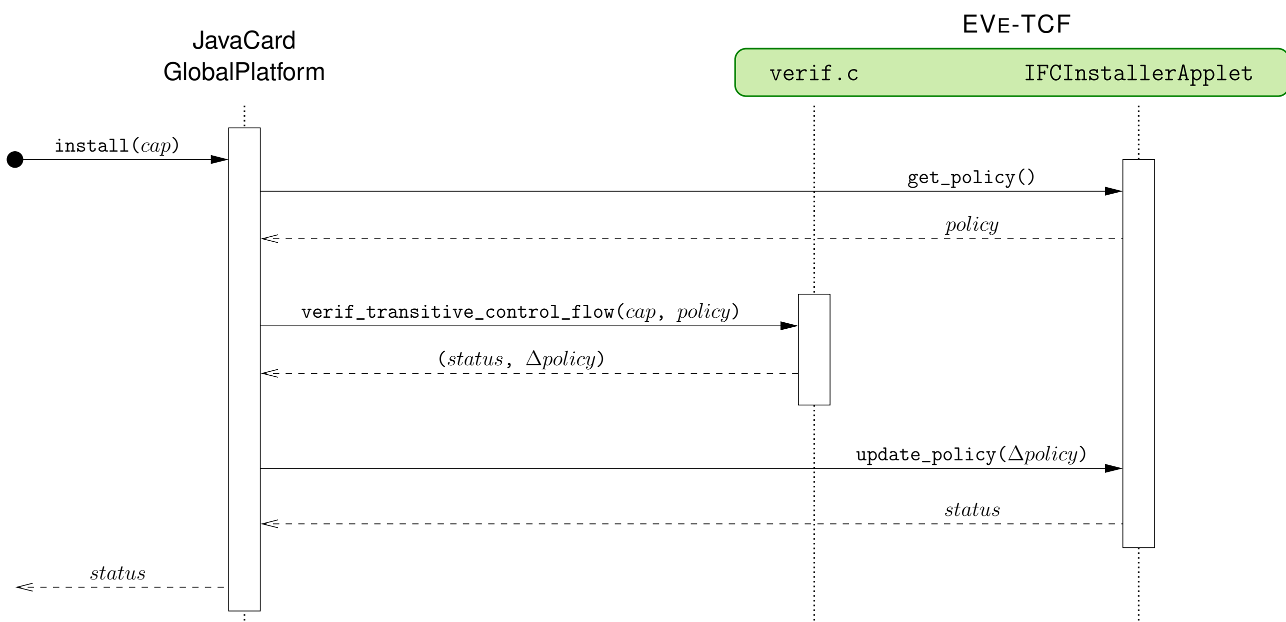


Figure 1.5: Schema of the on-device verification process at installation of a new package.

1.3.2 Removal of an installed package

As described in Section 1.1.2, implementation of package removal is straightforward for the targeted JavaCard 2.x device. It simply consists in removing the control flow policies related to the removed package once the system is sure no other package depends on it. In this case, the removal of package policies cannot fail, and only involves the `IFCInstallerApplet`, as depicted on Figure 1.6.

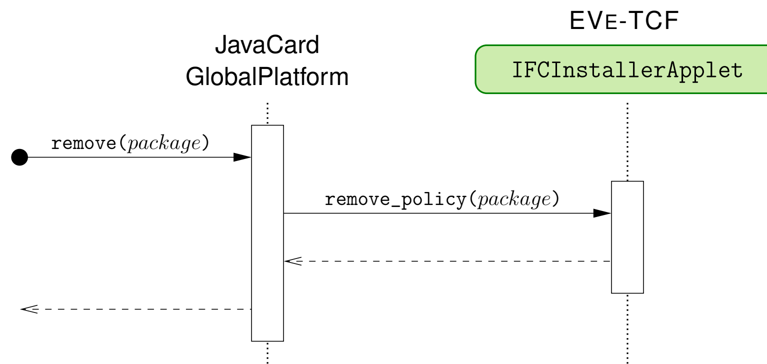


Figure 1.6: Schema of the on-device verification process at removal of an installed package.

1.4 On-device management of policies

Verified control flow policies of installed packages are stored on-device in repositories which structure is given on Figure 1.7. This structure is very similar to the one used in the TCF component (Section 1.2.3), except that method policies are now stored in a binary format, as described in the deliverable D6.3.

```
binary_repositories {
  u1 domains_count;
  u2 domains_size;
  aid domains[domains_count];
  u1 package_policies_count;
  binary_package_policy package_policies[package_policies_count];
}

binary_package_policy {
  u1 package_index;
  u2 class_policies_size;
  binary_class_policy class_policies[];
}

binary_class_policy {
  u2 classref;
  u1 class_token;
  u2 method_policies_size;
  binary_method_policy method_policies[];
}

binary_method_policy {
  u1 bitfield;
  u1 method_token;    /* present according to bitfield */
  u2 method_offset;   /* present according to bitfield */
  u1 policy;
}
```

Figure 1.7: Data structures of the control flow policies stored on-device.

The fields of the data structures displayed on Figure 1.7 have the following meaning:

- the `binary_repositories` structure describes a set of binary repositories of control flow policies:
  - `domains_count` represents the number of entries in the `domains` field;
  - `domains_size` represents the size in bytes of the `domains` field;

  - `domains` contains all the security domain AIDs mentioned in control flow policies of this repository;
  - `package_policies_count` represents the number of entries in the `package_policies` field;
  - `package_policies` contains a `package_policy` entry for each package installed[1];

- the `binary_package_policy` structure describes the control flow policy of a method:
  - `package_index` contains the internal package index of the current package;
  - `class_policies_size` represents the size in bytes of the `class_policies` field;
  - `class_policies` maps to each class (or interface) of the current package its control flow policy;

- the `binary_class_policy` structure is used to describe the control flow policy of a class or an interface:
  - `classref` contains the location (*i.e.* the offset) in the Class Component (see JCVM specifications) of the `info` structure corresponding to a class (or an interface) defined in this package;
  - `class_token` represents the class token of the current class (or interface), or `0xFF` if the current class (or interface) has no token assigned;
  - `method_policies_size` represents the size in bytes of the `method_policies` field;
  - `method_policies` maps to each method of the current class (or interface) its control flow policy;

- the `binary_method_policy` structure describes the control flow policy of a method:
  - `bitfield` is mask of modifiers used with a method with the following meaning[2]:

Mask	0x80		0x40		0x20	
Value	0x80	is visible	0x40	is implemented	0x20	is static
	0x00	is not visible	0x00	is abstract	0x00	is not static

Mask	0x03	
Value	0x00	repository of verified policies ($rules$ repository in the model)
	0x01	repository of verified policies of methods in packages with unsolved dependencies ($ruleswait$ repository in the model)
	0x02	repository of minimal expected policies of methods not yet installed ($unsolved$ repository in the model)

  - `method_token` represents the static method token or virtual method token or interface method token of this method if the method is visible according to `bitfield`;
  - `method_offset` represents a byte offset into the info item of the Method Component (see JCVM specifications) if the method is implemented (*i.e.* not an abstract method or a method definition in an interface) according to `bitfield`;
  - `policy` contains the bit-wise encoded policy of the current method where each bit of the seven highest bits corresponds to a security domain (a bit set to 1 means the domain is authorized) and the lowest bit encodes the special $\top$.

The binary encoding of a policy relies on domain indexes in the `domains` field of the `binary_repository` structure. If a method is authorized to be called from the i-th domain referenced in the `domains` structure, then its `policy` field has the bit masked by the value 2^i is set to 1.

This encoding on one byte allows to deal with up to seven different security domains on-device, which is enough for the targets of the SecureChange project. The size of the `policy` field can easily be increased for other targets where more security domains can exist.

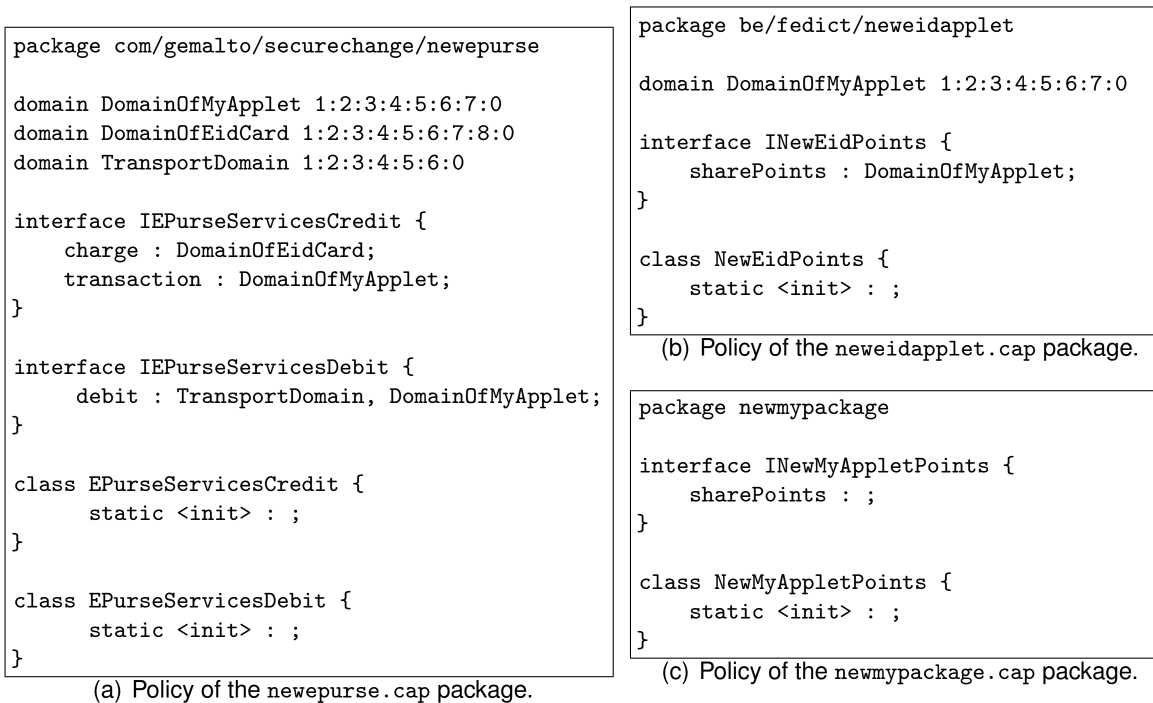
[1] Only non-safe packages have a control flow policy (Section 1.1.2).

[2] Only one repository is used in this prototype (Section 1.1.2).

1.5 The POPS case study

In the context of the SecureChange project, EVE-TCF prototype implementation of the transitive control flow model is applied to the integrated scenario of the POPS case study presented in the deliverable D6.6.

Figure 1.8 gives the control flow policies of the `newepurse.cap`, the `neweidapplet.cap` and the `newmypackage.cap` packages. The policy of the `newjticket.cap` is actually empty as it does not provide shared services to other packages. For this package, the default implicit policy (*i.e.* its methods can only be called from the security domain in which it is installed) is sufficient and is automatically infered by the `convert` tool.



```
package com/gemalto/securechange/newepurse

domain DomainOfMyApplet 1:2:3:4:5:6:7:0
domain DomainOfEidCard 1:2:3:4:5:6:7:8:0
domain TransportDomain 1:2:3:4:5:6:0

interface IEPurseServicesCredit {
    charge : DomainOfEidCard;
    transaction : DomainOfMyApplet;
}

interface IEPurseServicesDebit {
     debit : TransportDomain, DomainOfMyApplet;
}

class EPurseServicesCredit {
      static <init> : ;
}

class EPurseServicesDebit {
      static <init> : ;
}
```

(a) Policy of the `newepurse.cap` package.

```
package be/fedict/neweidapplet

domain DomainOfMyApplet 1:2:3:4:5:6:7:0

interface INewEidPoints {
    sharePoints : DomainOfMyApplet;
}

class NewEidPoints {
    static <init> : ;
}
```

(b) Policy of the `neweidapplet.cap` package.

```
package newmypackage

interface INewMyAppletPoints {
    sharePoints : ;
}

class NewMyAppletPoints {
    static <init> : ;
}
```

(c) Policy of the `newmypackage.cap` package.

Figure 1.8: Control flow policies of packages of the POPS scenario for `convert`.

Table 1.1 gives the size (in bytes) of each package without the TCF component, the size of each TCF component computed by the `convert` tool for the control flow policies given in Figure 1.8 and the overhead of embedding each component in the original package.

Package	Size of the original package	Size of the TCF component	Overhead
`newepurse.cap`	4614	171	+3.7%
`newjticket.cap`	3262	41	+1.3%
`neweidapplet.cap`	11540	415	+3.6%
`newmypackage.cap`	4778	92	+1.9%

Table 1.1: Size of TCF components embedded in packages of the POPS scenario.

Figure 1.9 gives the simulation script for the `simu` tool corresponding to the deployment scenario of the POPS case study. A security domain is created for each package just before its deployment, and the content of on-device policies repositories is dumped as is after each package installation.

Figure 1.10 shows the output of the `simu` tool executed on the simulation script of Figure 1.9.

Table 1.2 gives the size (in bytes) of on-device policies repositories after each installation of a package in the deployment scenario.

```
create domain BankDomain 1:2:3:4:5:0
install package "newepurse_tcf.cap" in BankDomain

dump policy "BinaryComponentAfterNewEPurse.bin"

create domain TransportDomain 1:2:3:4:5:6:0
install package "newjticket_tcf.cap" in TransportDomain

dump policy "BinaryComponentAfterNewJTicket.bin"

create domain DomainOfEidCard 1:2:3:4:5:6:7:8:0
install package "neweidapplet_tcf.cap" in DomainOfEidCard

dump policy "BinaryComponentAfterNewEIDApplet.bin"

create domain DomainOfMyApplet 1:2:3:4:5:6:7:0
install package "newmypackage_tcf.cap" in DomainOfMyApplet

dump policy "BinaryComponentAfterNewMyApplet.bin"
```

Figure 1.9: Simulation script of the POPS scenario for `simu`.

Installed packages	Size of the policies repositories
`newepurse.cap`	168
`newepurse.cap` + `newjticket.cap`	207
`newepurse.cap` + `newjticket.cap` + `neweidapplet.cap`	609
`newepurse.cap` + `newjticket.cap` + `neweidapplet.cap` + `newmypackage.cap`	699

Table 1.2: Size of on-device control flow policies repositories after each installation of a package of the POPS scenario.

```
Initializing RAM buffer (800 bytes)...
Initializing default platform values...
Initializing applet data buffer (800 bytes)...
Initializing APDU buffer (800 bytes, content offset is 0x14)...
Initializing system packages and security domains...
Initializing security policy...
    4 bytes -> 10 bytes
    1 new domains
    0 new package policies
Creating security domain BankDomain...

Selecting the domain BankDomain...
Parsing 'newepurse_tcf.cap'...
Importing 3 packages (GTO specific)...
Retrieving policy data from applet... 10 bytes
Verification of newepurse_tcf.cap...
Installing package in domain BankDomain...
Uploading policy data to applet...
    10 bytes -> 168 bytes
    4 new domains
    1 new package policies

Creating security domain TransportDomain...

Selecting the domain TransportDomain...
Parsing 'newjticket_tcf.cap'...
Importing 3 packages (GTO specific)...
Retrieving policy data from applet... 168 bytes
Verification of newjticket_tcf.cap...
Installing package in domain TransportDomain...
Uploading policy data to applet...
    168 bytes -> 207 bytes
    0 new domains
    1 new package policies

Creating security domain DomainOfEidCard...

Selecting the domain DomainOfEidCard...
Parsing 'neweidapplet_tcf.cap'...
Importing 6 packages (GTO specific)...
Retrieving policy data from applet... 207 bytes
Verification of neweidapplet_tcf.cap...
Installing package in domain DomainOfEidCard...
Uploading policy data to applet...
    207 bytes -> 609 bytes
    0 new domains
    1 new package policies

Creating security domain DomainOfMyApplet...

Selecting the domain DomainOfMyApplet...
Parsing 'newmypackage_tcf.cap'...
Importing 5 packages (GTO specific)...
Retrieving policy data from applet... 609 bytes
Verification of newmypackage_tcf.cap...
Installing package in domain DomainOfMyApplet...
Uploading policy data to applet...
    609 bytes -> 699 bytes
    0 new domains
    1 new package policies
Policy data size is 699 byte(s)
```

Figure 1.10: Simulation result of the POPS scenario with `simu`.

2. The Security-by-Contract Prototype

In this chapter we overview the Security-by-Contract prototype that enforces direct control flow policies on Java Card. The Security-by-Contract (S×C) approach for smart cards was introduced in D6.3 and D6.4. In the current report we present the implementation details of the prototype and an example of the S×C prototype applied to the WP6 running scenario.

The prototype exists in two versions: the embeddable S×C prototype and the testing S×C prototype. The embeddable prototypes is fully compliant with the card implementation details shared by Gemalto/Trusted Labs (presented in Appendix A) and it was shared with them for full industrial evaluation. The testing prototype is a prototype developed by UNITN for testing and demonstration purposes. It offers enhanced functionality over the embeddable prototype, because it enables the functional dependencies verification between applications. The embeddable prototype does not capture these dependencies explicitly because they are managed by the card independently from the S×C prototype. We provide more details in Appendix B.

2.1 An Overview of the SxC Prototypes

The S×C prototypes consist of three main components: the SxCInstaller, the ClaimChecker and the PolicyStore. The SxCInstaller is the main interface with the platform components, such as JavaStub or the apiobc library. It also comprises the functionality of the PolicyChecker component of the S×C framework. The ClaimChecker is the entity responsible for parsing the CAP files with the help of the apiobc library, extracting the contract from the Custom component and matching it with the bytecode. The SxCInstaller and the ClaimChecker are written in C. The PolicyStore component is responsible for storing the security policy and maintaining it across the card sessions, it is written in Java Card.

2.1.1 Embedding Contracts

Prior to loading the application contract needs to be embedded into the CAP file. This is done by the *CAP File Modifier* tool which has a user-friendly visual interface. This tool allows to create contracts, store them in files and embed them into existing CAP files. The Contract Custom components of CAP files are used as means to deliver contracts on the card. On the cards which are not equipped with the S×C framework the Contract Custom components will simply be ignored and they will not affect the functionality of the applications.

The S×C prototypes expects to receive a CAP file with the Contract Custom component, as we deliver contracts embedded into CAP files. The standard Java Card Development Kit implemented by Oracle does not support Custom components, so we have developed the *CAP Modifier* tool to embed contracts into CAP files. The process of the off-card conversion prior to loading is depicted in Figure 2.1. The white parts depict standard steps and tools in the Java Card application development. The grey parts are the new steps of the development process. After applying the standard Java Card tools (Compiler and Converter), the *CAP Modifier* tool takes as input already converted CAP file, appends the Contract Custom component and modifies the contents of the Descriptor component (by increasing the counter of the Custom components amount and specifying the length of the Contract Custom component), so the card can recognize that the CAP file contains a Custom component.

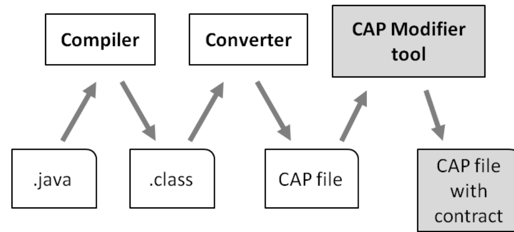


Figure 2.1: Embedding Contracts

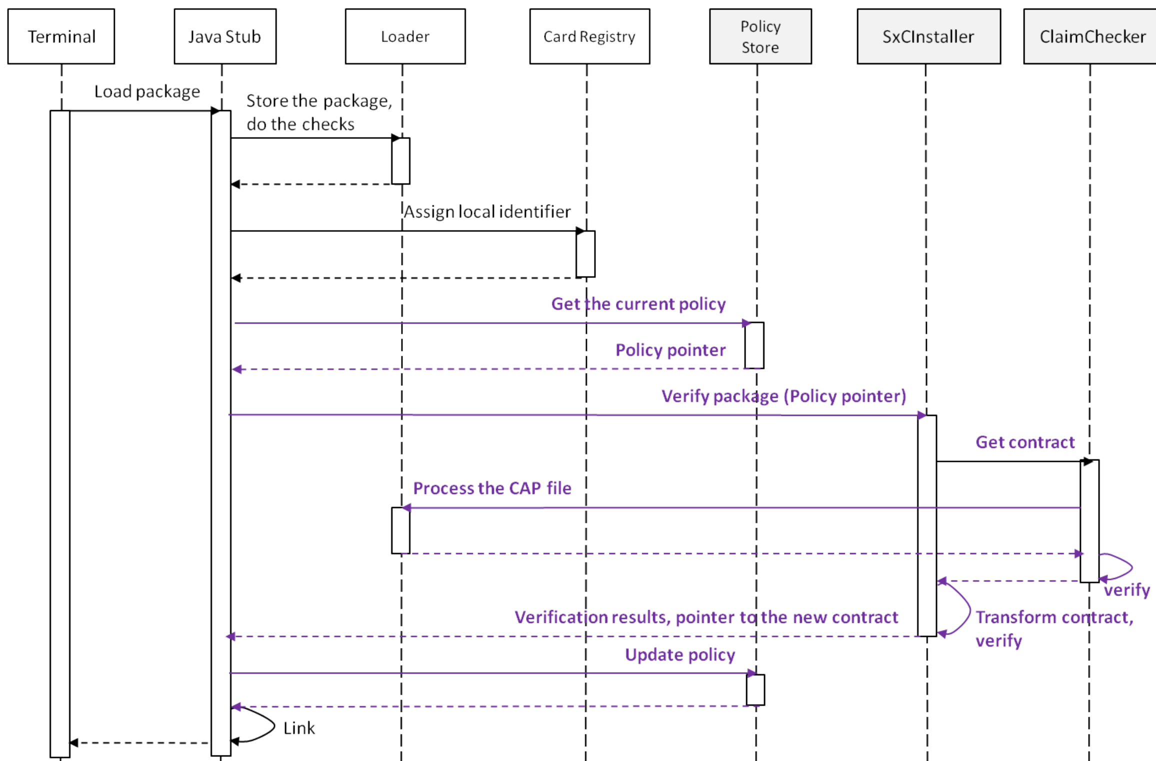


Figure 2.2: Loading Process in Presence of S×C Framework

2.1.2 The S×C Workflows on Device

The S×C prototypes support the following types of change on the platform: loading of a package, removal of a package and update of application policy of already loaded application package. As requested by Gemalto/Trusted Labs we allow some packages to avoid the S×C verification process. It is necessary because some packages have to be loaded by the smart card vendors after the card issuance for card personalization purposes (for example, the GlobalPlatform library). These packages are extensively verified within the smart card vendors premises and they are generally not needed to be declared in the smart card policy. We avoid verification of these packages by setting a comilation-time variable PrefixesToVerify and checking that the package requested for verification is within the specified range of the AID prefixes.

Figure 2.2 presents the application loading process for the cards with the S×C integrated prototype in a form of a sequence diagram. The grey components are the S×C components and the new steps in the loading are colored in violet. If the device does not have the embedded S×C prototype integrated these new steps are omitted.

The package removal process in presence of the embedded S×C prototype is presented in Figure 2.3. Again, the new verification steps are in violet. And Figure 2.4 presents the sequence diagram

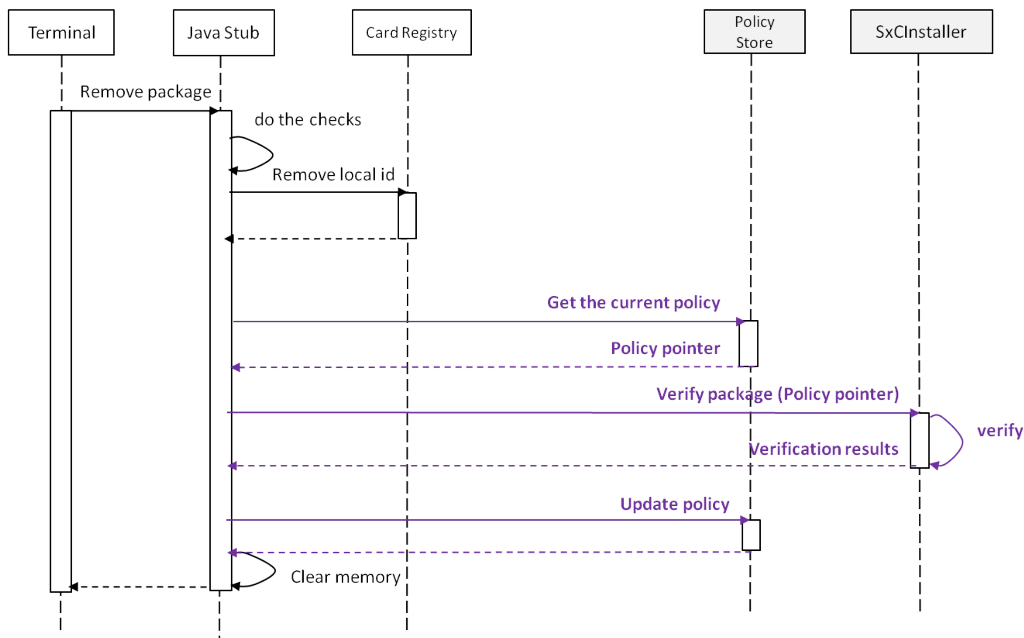


Figure 2.3: Removal Process in Presence of S×C Framework

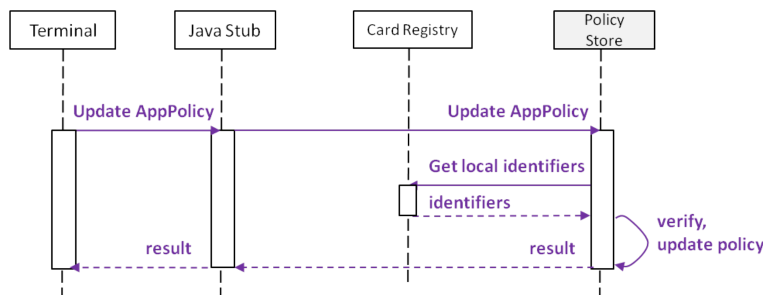


Figure 2.4: Application Policy Update Process

for the application policy update. The PolicyStore component is a class in the Java Stub, thus the terminal can contact the PolicyStore directly, once the new APDU command for the policy update is established. The application policy update process is completely new, it does not exist for standard Java Cards (because the access control policies to the services can only be embedded within the application code, thus the application has to be deleted and reinstalled if any access control rule has to be modified).

2.2 The S×C Embeddable Prototype Architecture

Figure 2.5 presents the Java Card architecture enhanced with the S×C framework. More details on the Java Card architecture are available in Appendix A. We also present a brief summary on the application contracts and more details on the contracts population in Appendix B.

2.2.1 The Claim Checker Algorithm

The ClaimChecker component is responsible for verification of the contract and the bytecode compliance. Thus it has to establish that the services from $\mathsf{Provides}_A$ exist in package A and the services from Calls_A are indeed the only services that A can try to invoke in its bytecode. The goal of the ClaimChecker algorithm is to collect for each `invokeinterface` opcode the method index t and the Constant Pool

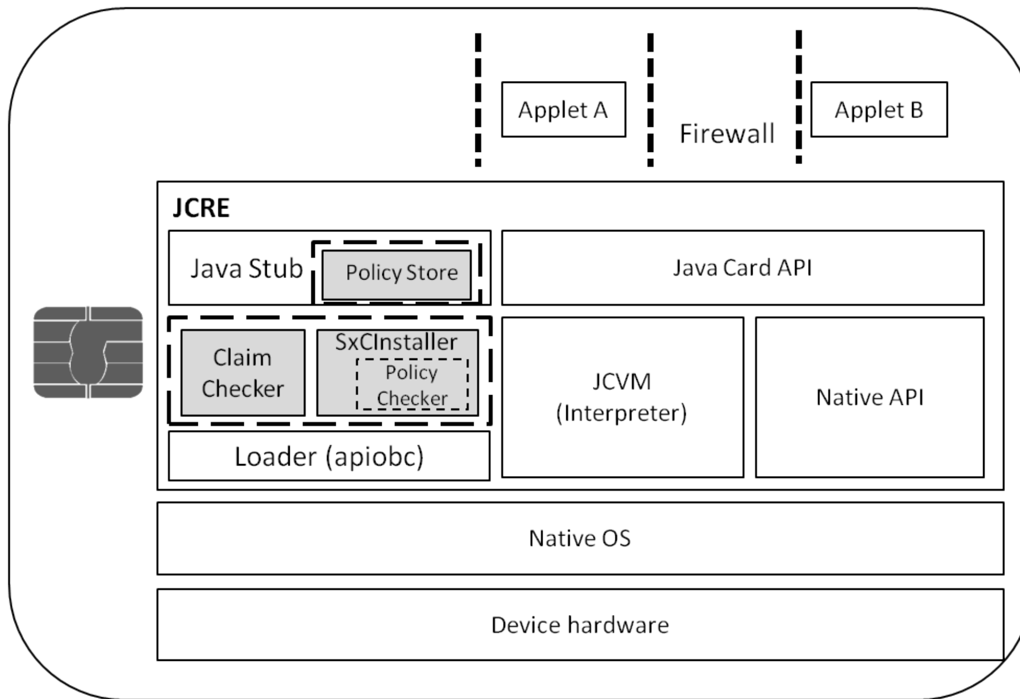


Figure 2.5: The SxC Embeddable Prototype Architecture

index I, because the JCRE specification allows the context switches through the Firewall only by this opcode. Then we can compare the collected set with the set Calls of the contract. We emphasize that operands of the `invokeinterface` opcode are known at the time of conversion into a CAP file and thus are available directly in the bytecode. All methods of the application are provided in the Method Component of the application's CAP file, an entry for each method contains an array of its bytecodes. Exported shareable interfaces are listed in the Export component of the CAP file and flagged in the Class component. The strategy for the ClaimChecker is to ensure that each service listed in the Provides set is meaningful and no other provided services exist.

2.2.2 The On-card Policy Store

The PolicyStore is responsible for storing the security policy of the card. It needs to be organized efficiently, so that the PolicyChecker algorithms are fast, but the space occupied by the security policy data structures is small. We used the bit vectors format to store the policy data and to restrict the number of applets and services admitted for loading. This restriction allows us to use the fixed size format of bit vectors. Thus we have developed the prototype assuming up to 4 loaded applets at each moment of time (the 5th will be rejected by the current implementation, but it is possible to free the space by removing something loaded), each applet can provide up to 8 services.

The data structures that are maintained in the Policy Store are: Policy, Mapping, MayCallObj and WishListObj. The Mapping object maintains a mapping between on-card S×C package identifiers (from 0 to 10) and the AIDs of the application packages and a mapping between the on-card service identifiers (from 0 to 7) and the service tokens (interface and method token)). The S×C package identifiers are generally different from the local package identifiers that are assigned by the JCRE and maintained by the card registry. This approach was chosen because it allows the S×C prototype to assign the identifiers independently from the JCRE assignment of the local identifiers, thus it minimizes the disclosure of the platform implementation details and the dependence of the S×C prototype on a specific platform implementation, ensuring interoperability.

The Policy object is a set of byte arrays corresponding to the contract structure, the arrays contain the data about provided and called services, the security rules and the functionally necessary services,

all the applications and services are referenced by their on-card identifiers. The MayCallObj object stores the data about authorizations for applications that are not loaded on the card and the WishListObj object stores services that a loaded application tries to invoke, but they are not yet present on the card. These two objects and the Mapping object are space-consuming, because they store the AIDs, but they are used rarely with respect to the Policy object. The Policy object is used for the contract-policy compliance checks, so a lot of bit vector operations can be applied to it. The Mapping, MayCallObj and WishListObj objects are used only for contract transformation into internal format, when the delivered Provides set is mapped into internal format and the services obtain the on-card identifiers, the delivered sec.rules set is mapped into either a Policy security rules set or the MayCallObj object depending on the AID of the trusted client, etc.

2.2.3 The Policy Checker

The PolicyChecker is the component responsible for contract-policy compliance checks. It needs to retrieve the security policy of the card from the Policy Store and the received contract from the ClaimChecker. The contract is then converted into the internal on-card format compliant with the security policy structure. The PolicyChecker algorithms are applied to the Policy data structures and the transformed contract. Intuitively, during loading of application B the PolicyChecker checks that (1) for all the services from Calls_B B is authorized by their providers to call them; (2) for all services from Provides_B all the applications that can invoke these services are authorized by B; (3) all the services from func.rules_B are provided. We refer the interested readers to D6.3, D6.4 for more details.

2.3 The Prototype for Testing

The S×C prototype for testing comprises the same components: the SxCInstaller, the ClaimChecker and the PolicyStore. However, it differs with the embeddable prototype in the following.

- The embeddable S×C prototype in fact does not make use of the functional dependencies among applications. This is due to the fact that the JCRE implementation requires that all the imported packages are present at the moment of application loading. If this is not the case, the application is rejected. Similarly, a package imported by some applications on the card cannot be deleted. Thus the JCRE executes the functional dependencies checks defined by the S×C approach prior to the S×C framework invocation and therefore the S×C embeddable prototype relies on these checks. The WishListObj object is not maintained by the embeddable prototype. The testing prototype does not follow these restrictions and it performs the functional dependencies checks as defined in D6.3, D6.4, thus it maintains the WishListObj object if a loaded package contains a call to a package that is not (yet) loaded.
- After the first round of integration of the S×C prototype with an actual device the industrial partner Gemalto/Trusted Labs concluded that the APDU buffer, previously defined as means for Java Card-C components interactions, cannot be used during loading process. This required a modification of the previously agreed architecture, where the PolicyStore was a dedicated applet, to the new architecture, where the PolicyStore is a class in JavaStub. Since the main goal of the testing prototype implementation was independent functionality testing, we did not modify the testing prototype after the discovery that the APDU buffer solution does not work. The communication means can be validated only on real industrial prototypes, because the full JCRE implementation is not possible for the academia partners. We emphasize that the functionality of the components of the S×C embeddable and testing prototypes is the same, except for the functionally necessary services.

The architecture of the S×C testing prototype is depicted in Figure 2.6. For the purposes of independent functionality testing we have implemented the apiobc library and the necessary Java Stub functionality following the JCRE specifications.

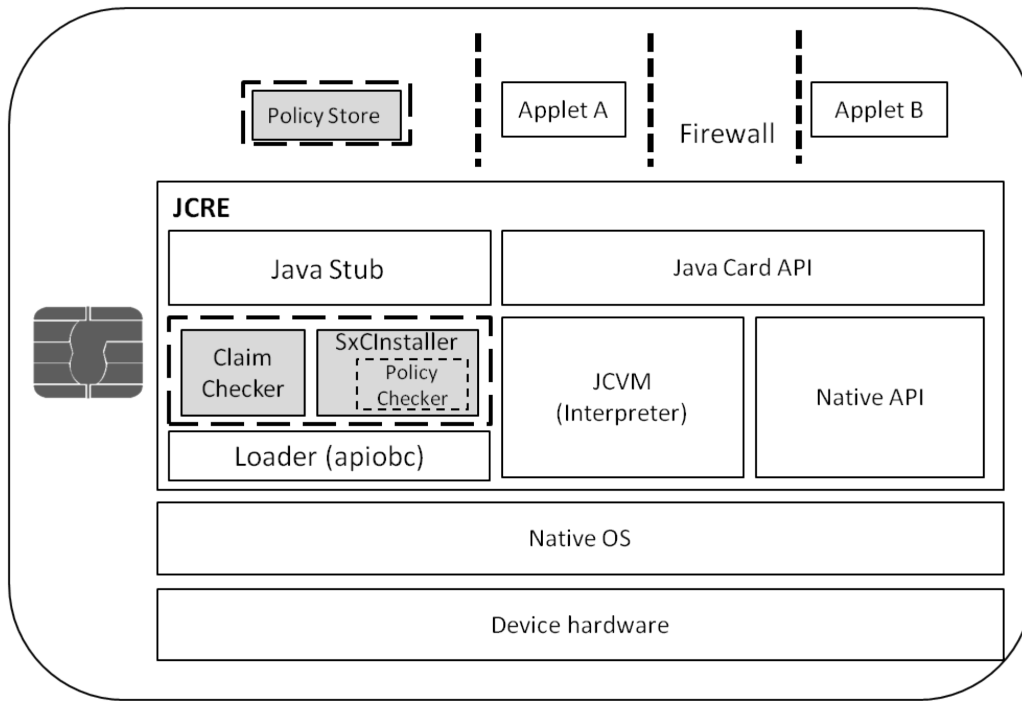


Figure 2.6: The SxC Testing Prototype Architecture

We present an overview of the S×C prototype testing using the integrated WP6 scenario in Appendix B.

Conclusion

The deliverable D6.5 consists of two prototypes for on-device information protection on JavaCard smart cards. These prototypes implement the *transitive control flow* model (Chapter 1) and the *Security-by-Contract* model (Chapter 2) introduced for the SecureChange project in the deliverables D6.3 and D6.4.

The first round of integration of those prototypes with an actual device by the industrial partner Gemalto/Trusted Labs concluded that the APDU buffer, previously defined as the mean to exchange information between C code and Java code, cannot be used during loading process. Furthermore, the APDU buffer also has a fixed size of 255 bytes which strongly limits communications. The off-device simulators provided with our prototypes can easily bypass this limitation by increasing the size of the APDU buffer, but this is not possible on-device. Lately, Gemlato/Trusted Labs suggested another solution not relying on the APDU buffer, *i.e.* static class directly included in the Java Stub instead of an applet, but we had no time to investigate and develop this proposal.

Whatever the concrete integration mean EVE-TCF and S×C prototypes will use, it will not alter the main results achieved. Both prototypes have shown their ability to effectively deal with real application code, and the feasibility of their integration in real smart cards.

A. The Device Architecture

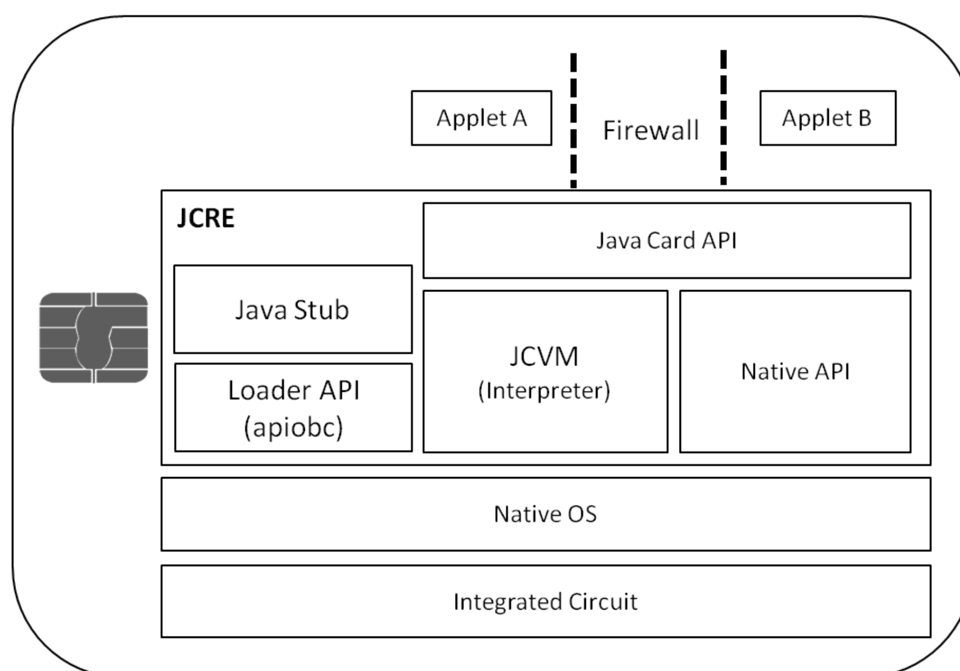


Figure A.1: The Java Card Architecture.

In this chapter we briefly overview the Java Card architecture in order to ease the understanding of the prototypes implementation and device integration perspectives.

Java Card is a technology enabling multi-application smart cards. In essence, the Java Card technology brings a Java Virtual Machine on integrated circuits. Figure A.1 presents the Java Card architecture. The main components are: an Integrated Circuit ("chip"), a Native Operation System (Native OS) and the Java Card Run-time Environment. The JCRE comprises a Java Card Virtual Machine (JCVM), a set of Java Card API, the JavaStub component and the Loader (the apiobc library is a part of it).

The platform components are implemented in C or in Java Card. The Loader is implemented in C, and the main components of the WP6 prototypes (EVE-TCF and the S×C prototype) are implemented in C because they have to be integrated with the Loader. The Loader API comprises a set of functions used to load applications (process the received file, perform the necessary checks, etc.).

Applications are written in Java Card and use the Java Card API. The JavaStub component is a part of the Installer/Applet Deletion Manager components of the platform (not presented on Figure A.1). It has to exhibit some functionality of applets (be selectable, for example) and it is implemented in Java Card. The communication between some of the platform components is therefore hindered.

The WP6 prototypes need to allocate persistent modifiable memory (EEPROM) in order to store the security policy/meta-data on the card. Only the Java Card components can allocate EEPROM, but not the C components. Thus we had implement the Policy Store component in Java Card. The

interactions between the C – Java Card components were initially organized through the APDU buffer. The APDU buffer is a global byte array accessible for all applications and, theoretically, all components of the JCRE. The prototypes explored 255 bytes length APDU buffer.

Java Card applications (also called *applets*) are delivered on a card in packages, that are converted into CAP files. An applet developer writes an application in Java Card (subset of Java) then the application is compiled into .class files and afterwards converted into a CAP (Converted APplet) file. The Converter is an off-card part of the JCVM. The main purpose of the conversion is to reduce the amount of memory needed for storing an applet. The structure of the CAP files are defined by the Java Card specifications, thus the JCVM can process them in an optimized manner.

The *application loading process* includes the following steps. The JavaStub component is selected on the card from a terminal and it is the entity responsible for loading, linking and installation. Upon receiving a CAP file the Loader API is used to process the file and perform some checks defined by the Java Card specification. Then the package can be linked and afterwards an application instance can be created. At some point the application may no longer be needed. Then the JavaStub starts the removal process. Upon performing the necessary checks, the application instances (if they exist) and the package can be removed and the CAP file can be deleted from the memory (the *application removal process*).

Java Card packages and applications can be uniquely identified by their AID (Application IDentifiers). An AID is a byte array, it can be 5 – 16 bytes length. On the card the packages are referred to by their *local identifiers*, these identifiers are assigned by the JCRE, which maintains (in the *card registry*) the package AID – local identifier correspondence.

The apiobc Library

The apiobc library contains a set of the Loader functions and definitions of data types and constants available on an actual device. The Loader functions access is essential for both prototypes, because the CAP file contents access on device is organized through these functions. The apiobc library contains functions that provider pointers to the beginning of each CAP file component and the length of each component, including the Custom components carrying the application policy/contract. Other functions of the apiobc library query the card registry and return the local package identifier for the given AID, inform if the current CAP file was converted with the Java Card 2.1 Converter and give a pointer to a temporary auxiliary buffer used to store the temporary date of the prototypes.

Gemalto/Trusted Labs have shared this library (function signatures and description) with INR-Lille and UNITN.

The Applet Interactions

Applications are isolated on the card by the JCRE Firewall mechanism. The Firewall confines each applet's actions to the applet's *context*. Each package has its own context, so objects can communicate freely within the same package.

The JCRE allows only methods of *Shareable interfaces* (the interfaces extending javacard.framework.Shareable) to be accessible through the Firewall. If an application desires to share some methods, it implements a Shareable interface. This application is called *a server* and the shared methods are called *services*. An application that can try to call a service is called *a client*.

B. The Security-by-Contract Prototype Details

Summary of the Chapter

In this chapter we give a short reminder on the S×C contracts and present the contracts population in more details. We also present some details of the S×C functional testing on the integrated WP6 scenario.

B.1 Embedding Contracts

We remind that the S×C approach expects that each application will bring a contract that contains the information on provided and called services and security policy of the application.

B.1.1 Application Contract

In order to make this report self-contained we now remind the structure of the application contracts initially presented in D6.3 and D6.4.

Let $A.s$ be a service s declared in a package A. The contract consists of two parts: a *claim* and a *policy*. AppClaim specifies provided (Provides set) and invoked (Calls set) services. We say that the service $A.s$ is provided if applet A is loaded and service s exists in its code. Service $B.m$ is invoked by A if A may try to invoke $B.m$ during its execution. The AppClaim will be verified for compliance with the bytecode (the CAP file) by the ClaimChecker.

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

Definition B.1.1 *Let $\Delta_{\mathcal{A}}$ be a domain of applications and $\Delta_{\mathcal{S}}$ be a domain of services.* $\mathsf{AppClaim}_\mathsf{A}$ *of an application A is a tuple $\langle \mathsf{Provides}_\mathsf{A}, \mathsf{Calls}_\mathsf{A} \rangle$, where $\mathsf{Provides}_\mathsf{A} \subseteq \Delta_{\mathcal{S}}$ is a set of the services A provides on the card and $\mathsf{Calls}_\mathsf{A} \subseteq \Delta_{\mathcal{S}}$ is a set of services that A may call during its execution.*

$\mathsf{AppPolicy}_\mathsf{A}$ *of an application A is a tuple $\langle \mathsf{sec.rules}_\mathsf{A}, \mathsf{func.rules}_\mathsf{A} \rangle$, where a relation $\mathsf{sec.rules}_\mathsf{A} \subseteq \mathsf{Provides}_\mathsf{A} \times \Delta_{\mathcal{A}}$ defines which applications are authorized to use services of A, $\mathsf{func.rules}_\mathsf{A} \subseteq \Delta_{\mathcal{S}}$ is a set of services functionally necessary for application A.*

$\mathsf{Contract}_\mathsf{A}$ *is a tuple $\langle \mathsf{AppClaim}_\mathsf{A}, \mathsf{AppPolicy}_\mathsf{A} \rangle$.*

A service s can be identified as a tuple $\langle A, I, t \rangle$, where A is unique application identifier (AID) of the package that provides the service s, I is a token for a shareable interface where the service is defined and t is a token for the method s in the interface I. Further we will sometimes omit an AID A and will refer to a service as a tuple $\langle I, t \rangle$.

Tokens are used by the JCRE for linking on the card in the same fashion as Unicode strings are used for linking in standard Java class files. For externally visible elements, such as shareable interfaces and their methods, tokens are declared in the Export file of the package. If applet A wants to provide some services, it has to make its Export file available for all potential clients. Applet B in its

source code refers to services by their Unicode string names, but when it is converted into a CAP file these names are replaced with tokens from A's Export file. Thus it is possible to identify provided and called services in terms of tokens correctly and uniquely.

A functionally necessary service for applet A is the one which absence on the platform will crash A or make it useless. For example, a transport application normally requires some payment functionality to be available. If a customer will not be able to purchase the tickets, she would prefer not to install the ticketing application from the very beginning.

An authorization for a service access includes the package AID of the authorized client (the format of an authorization will be discussed further). The access rules have to be specified separately for each service and each client that the server wants to grant access.

B.1.2 The Contract Delivered on the Card

Contracts can be delivered on the card within Custom components of the CAP files. Custom components require to have a tag and an AID. We have defined the tag to be 0xC3 and the AID 0x010203040506C3 (but these can be easily modified). These details of the Custom component and its length are listed in the Directory component of the CAP file and are presented in Table B.1.

```
custom_component_info {
    u1 component_tag
    u2 size
    u1 AID_length
    u1 AID[AID_length] }
```

Table B.1: Details of the Custom component

```
contract {
    u2 provides_count
    provides_info  provides[provides_count]
    u2 calls_count )
    calls_infocalls[calls_count]
    u2 secrules_count
    secrules_info  secrules[secrules_count] }
```

Table B.2: Structure of the Custom component Containing Contract

The scheme of the contract is illustrated in Table B.2. The order of the contract attributes is expected to be: Provides, Calls, sec.rules. Thus we just add the number of corresponding elements before each attribute. Elements of each attribute have different structures, that are provided in Table B.3 (we use structures and naming that are similar to the ones defined for CAP files, there u1 corresponds to 1 byte and u2 corresponds to 2 bytes). The contract is just a byte array, but specifying structures corresponding to each entry allows us to perform the contract extraction efficiently.

Functionally necessary services are a subset of called services: $\mathsf{func.rules}_A \subseteq \mathsf{Calls}_A$, thus just tag necessary services among the called ones. The value of funcrules_tag is set to 0x01 if the service should be listed in func.rules. Otherwise the tag value should be 0x00.

B.1.3 Contract Population

Following are the rules for contract population.

- *Provided Services.* A service is required to be listed in the Provides set if it is a method of an interface extending Shareable. A service is listed in Provides array as a pair $\langle \mathsf{I}, \mathsf{t} \rangle$, where I is the Export file token for shareable interface and t is the Export file token for the method (1 byte each).

```
provides_info {
    u1 interface_token
    u1 service_token }
calls_info {
    u1 interface_token
    u1 service_token
    u1 server_AID[16]
    u1 funcrules_tag }
secrules_info {
    u1 client_AID[16]
    u1 secrules_applet_count
    secrules_applet_info  secrules_applet[secrules_applet_count] }
secrules_applet_info {
    u1 interface_token
    u1 service_token }
```

Table B.3: Contract Attributes Structures in the Contract Custom component

- *Called and Functionally Necessary Services.* An application provider should list a service (belonging to another package) in the Calls set, if an invocation of this service is present in the code of the applet. A service from a package with AID XXX is listed in the contract as $\langle XXX, \mathsf{I}, \mathsf{t}, \mathsf{funcrules_tag}\rangle$, where funcrules_tag tags if this service is also functionally necessary or not. For optimization purposes, the Calls set is then restructured to separate services provided by different servers. The AIDs are space-consuming objects (can take up to 16 bytes) and avoiding their repetitions where possible can bring significant space savings.

- *Authorization Rules.* An authorization rule is listed in the sec.rules set as a pair containing the service details (defined as a provided service) and the authorized client package AID. Thus the structure is the same as for a called service, with a difference that no tag for functionality is needed: $\langle AID, \mathsf{I}, \mathsf{t}\rangle$. Then the same optimization strategy as for called services is applied.

The CAP file is in fact a JAR archive with a known structure. In order to embed the contract created by these rules and in compliance with the structure from Table B.2, our *CAP Modifier* takes the CAP file generated with the standard Java Card tools and appends the Contract Custom component within it, modifying the Directory component accordingly (as the specification requires).

The *CAP Modifier* GUI screen-shot is presented on Figure B.1, it depicts an existing contract and the options that users of the *CAP Modifier* tool have. The user can choose to add services to Provides, Calls/func.rules and sec.rules sets, then the dialog will appear where the user can insert the necessary AID and tokens. When the contract is ready it can be saved for future usage. The contract can also be embedded into the chosen CAP file, and then the CAP modifier can generate the scripts necessary to communicate the CAP file to the card.

B.2 An Example

We have conducted an extensive functionality testing of the S×C prototypes using the 4 applications of the integrated WP6 scenario (see D6.6 for more details). Figure B.2 depicts an example of a contract for the NewMyApplet application created in the *CAP Modifier* tool. This contract is faithful (compliant with the application code). It declares that the NewMyApplet application provides one service, calls two services from the NewEPurse application and one service from the NewEidCard application, and authorizes the NewEPurse application to call its own provided service.

Figure B.3 presents a compliant contract for the NewEPurse applet. It contains the tokens of 3 provided services and the authorizations for the NewJTicket, NewEidCard and NewMyApplet applets to

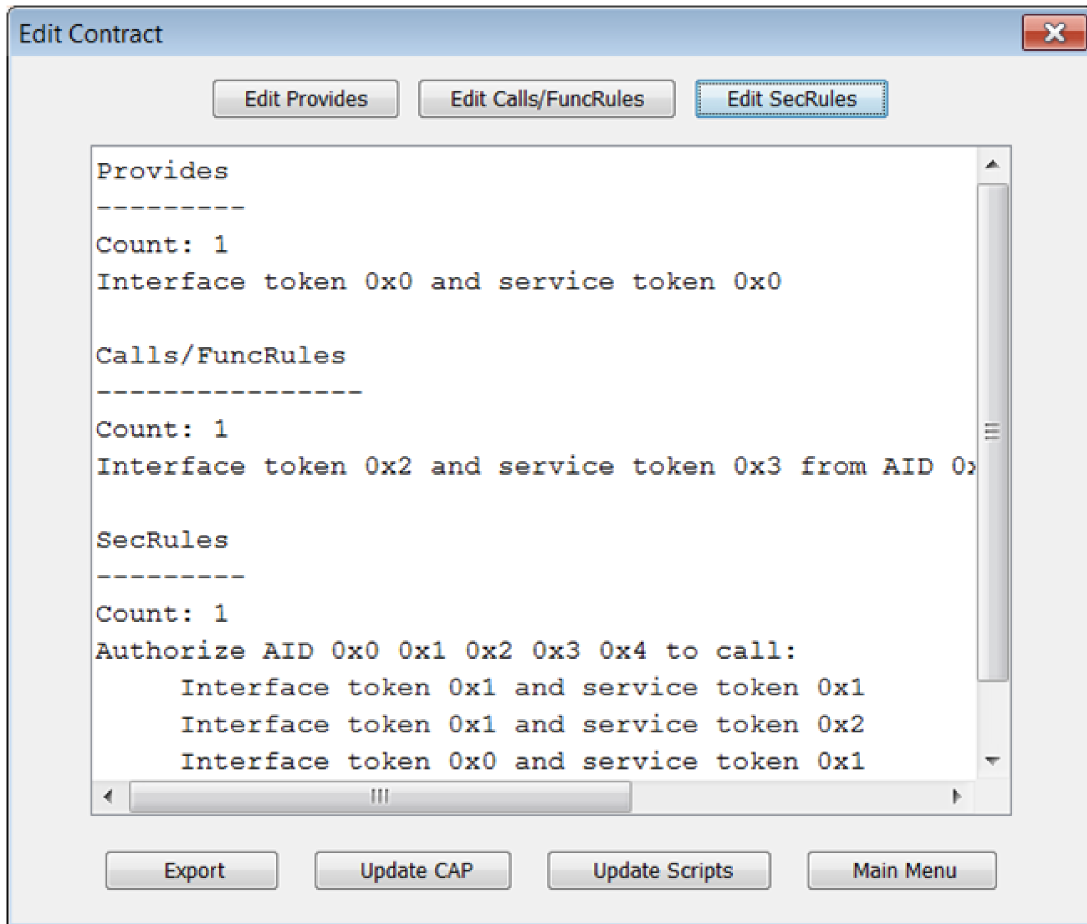


Figure B.1: User Interface of CAP modifier

access its services. Similarly, Figures B.4-B.5 present examples of contracts for applets NewJTicket and NewEidCard correspondingly. We note that the contract of the applet NewJTicket contains a service that is actually listed as a functionally necessary service.

Table B.4 presents the sizes of the CAP files used in the WP6 integrated scenario without contracts and the corresponding sizes of the Contract custom components.

Applet	Original CAP file without contract (Bytes)	Size of the contract (Bytes)	Overhead (%)
NewEPurse	4613	176	+3,8%
NewJTicket	3263	22	+0,6%
NewMyApplet	4778	58	+1,2%
NewEidCard	11541	37	+0,3%

Table B.4: CAP files sizes with and without contracts

B.3 Using the S×C Prototype for Testing

The S×C prototype for testing exists as a set of scripts that can be run on any Windows PC. The only requirement for the prototype user is to download the Java Card Development Kit 2.2.2 from the Oracle web site (it's free) and have the Java Run-time Environment installed. The variables JAVA_HOME and JC_HOME have to be set to denote the paths to the JRE and the JCRE tools correspondingly.

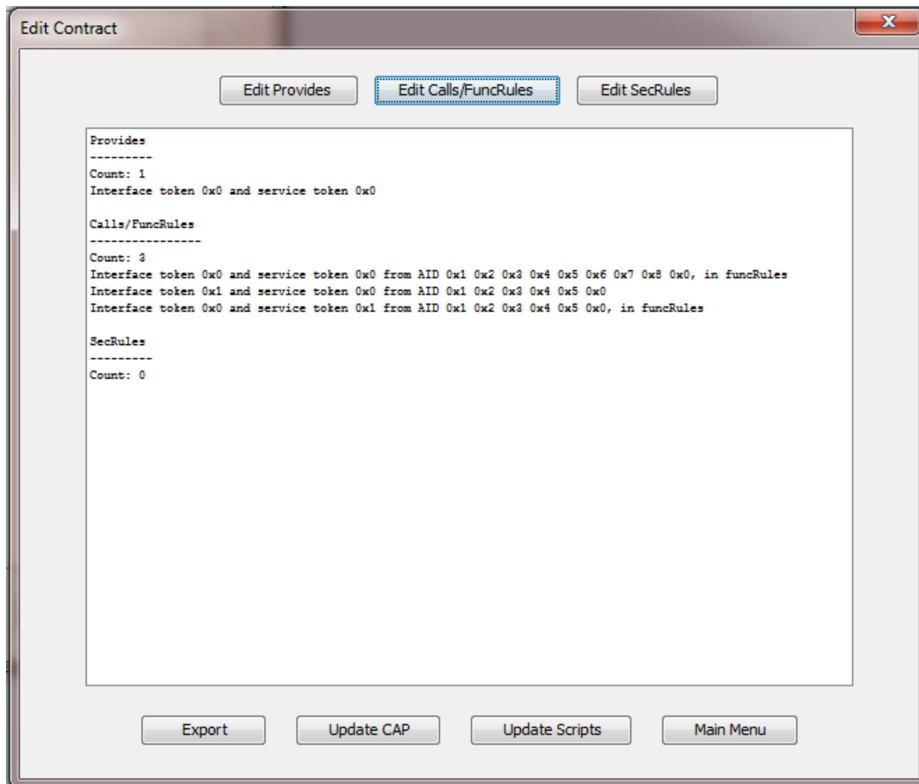


Figure B.2: A Contract for NewMyApplet

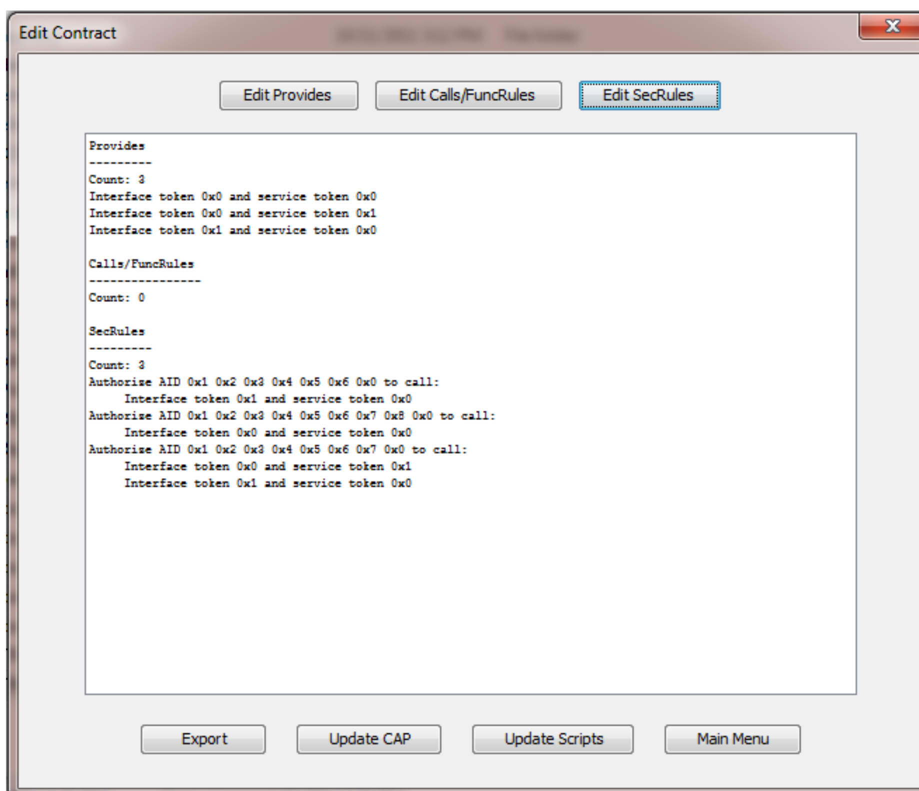


Figure B.3: A Contract for NewEPurse

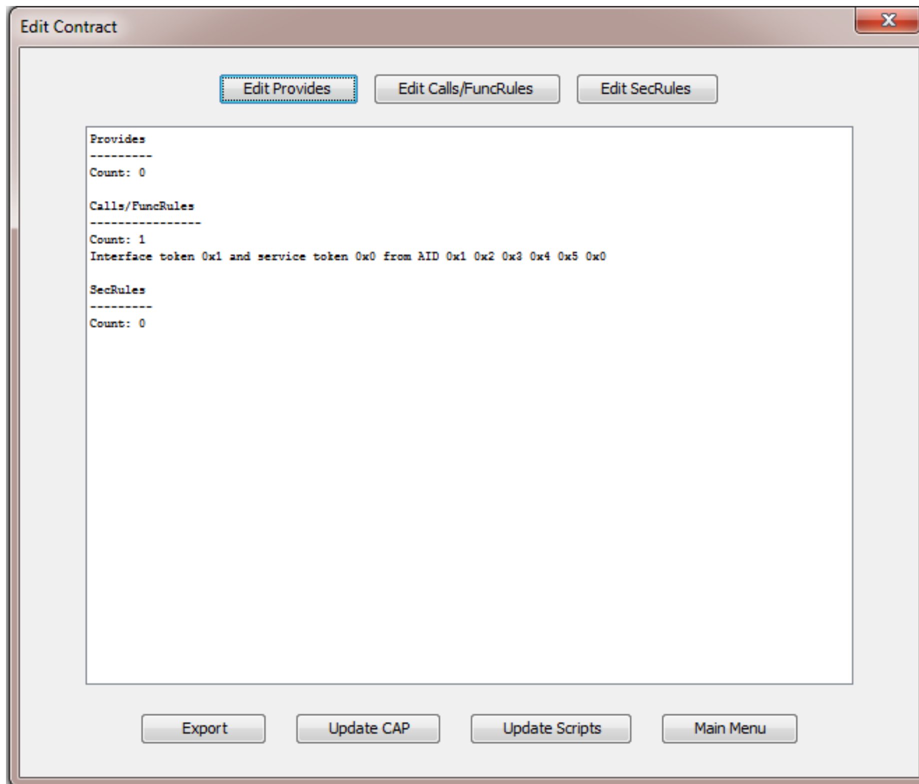


Figure B.4: A Contract for NewJTicket

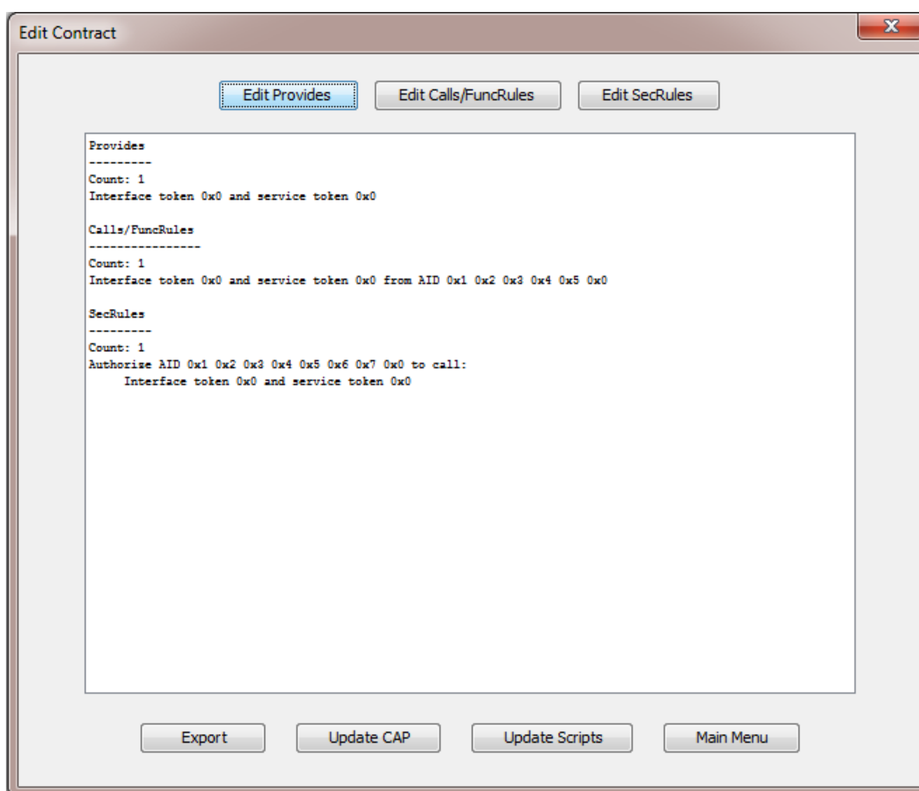


Figure B.5: A Contract for NewEidCard

Any interested person can try to use the S×C prototype for testing. The CAP files for testing the prototypes can be created with the help of the Eclipse JCWDE plug-in or the Java Card NetBeans plug-in, or the CAP files of the WP6 integrated scenario applets can be used.

The scripts to run the S×C prototype for testing run seamlessly the Java Card Development Kit tools. In the beginning the card simulation is initialized with just the PolicyStore applet deployed. The user is then required to embed the contract into the selected CAP file, place the selected CAP file into the folder with the S×C scripts and type the name of the selected CAP file in the command line. It is possible to run a simulation of a CAP file loading, removal or an application policy update. Depending on the chosen option the evolution will either be performed, or, in case of a non-compliant evolution attempt, an error will be reported to the user. We provide the user guide together with the S×C prototype for testing.