D6.6: Development-Time and On-Device Interplay

Arnaud Fontaine (INR), Olga Gadyatskaya (UNITN), Frank Piessens (KUL), Isabelle Simplot-Ryl (INR), Jan Tobias Mühlberg (KUL), Fabio Massacci (UNITN), Anton Phillipov (UNITN), Pedro Capelastegui (TID), Bart Jacobs (KUL), Pieter Philippaerts (KUL)

Document Information

Document Number	D6.6
Document Title	Development-Time and On-Device Interplay
Version	1.2
Status	Final
Work Package	WP 6
Deliverable Type	Report
Contractual Date of Delivery	M36
Actual Date of Delivery	M36
Responsible Unit	KUL
Contributors	INR, KUL, UNITN, TID
Keyword List	Verification, On-device, Off-device
Dissemination	PU

Document Change Record

Version	Date	Status	Author (Unit)	Description
0.1	2011/12/02	Working	Jan Tobias Mühlberg (KUL)	Document outline and style
0.2	2011/12/08	Working	A. Fontaine (INR)	Raw material integration in section 3.2
0.3	2011/12/20	Working	Jan Tobias Mühlberg, Bart Jacobs, Pieter Philippaerts (KUL)	Integration of most of KUL's work on VeriFast
0.4	2011/12/21	Working	A. Fontaine (INR)	Splitted section 3.2 in 3.2 and 3.3; updated 3.2 and 3.3
0.5	2011/12/23	Working	Jan Tobias Mühlberg (KUL)	More material on off-device verification, added appendix, improved document layout
0.6	2011/12/23	Working	A. Fontaine (INR)	Updated sections 4.2, 4.3 and appendices
0.7	2011/12/26	Working	Olga Gadyatskaya (UNITN)	Added contents for 4.1 and appendix E
0.8	2011/12/26	Working	Olga Gadyatskaya (UNITN)	Added contents for chapter 2, updated appendix E
0.9	2011/12/27	Working	A. Fontaine (INR)	Added appendix on POPS scenario, updates to sections 4.2 and 4.3
0.10	2011/12/27	Working	Jan Tobias Mühlberg (KUL)	Added appendices A and B. Todo: condense chapter 3.
0.11	2011/12/28	Working	Frank Piessens (KUL)	First draft of executive summary
0.12	2011/12/28	Working	Frank Piessens (KUL)	First draft of Introduction
0.13	2011/12/29	Working	Jan Tobias Mühlberg (KUL)	Shortened sections 3.1 and 3.3.
0.14	2011/12/31	Working	Jan Tobias Mühlberg (KUL)	Integrated section 3.2: Verifying PEP
0.15	2012/01/02	Working	Jan Tobias Mühlberg (KUL)	Added conclusions on PEP verification
0.16	2012/01/03	Working	Jan Tobias Mühlberg (KUL)	Some cleanup, spell checking, etc.
0.17	2012/01/03	Working	Frank Piessens (KUL)	Extended Ch. 2, minor fixes throughout.

Version	Date	Status	Author (Unit)	Description
0.18	2012/01/08	Working	Olga Gadyatskaya (UNITN)	Added conclusions for HOMES case study.
1.0	2012/01/09	Final	Frank Piessens (KUL)	Minor fixes.
1.1	2012/01/11	Final	Michela Angeli (UNITN)	First quality check completed – minor remarks.
1.2	2012/01/12	Final	Jan Tobias Mühlberg (KUL)	Implemented minor remarks.

Executive Summary

This document summarizes the work performed in Task 6.6 of Work Package 6 of the SecureChange project funded by the European Commission within the Seventh Framework Programme. The overall objective of Work Package 6 is the development of verification techniques for evolving systems, with a strong focus on the development time and deployment time phases of the software lifecycle.

In the first two years of the project, WP6 developed several technologies to support verification of evolving systems. These technologies include *off-device, development-time* verification techniques, and *on-device, deployment-time* verification techniques.

For the off-device techniques, the theory was developed in the first year (reported in deliverable D6.1), and a prototype was developed in the second year (reported in deliverable D6.2). In the third and final year, work has focused on the evaluation and validation of these results in the SecureChange case studies. This deliverable D6.6 reports on the application of our off-device verifier to the POPS and HOMES case studies. In summary, our results show that our prototype verifier is ready to handle real industrial code (both JavaCard and C code), that verification is performed fast even on code bases of thousands of lines of code, and that verification finds bugs. On the downside, we find that applying our verification technique is labor intensive. In particular, we find that the annotation overhead (i.e. the amount of annotation that the developer/verification engineer has to provide) is relatively high. To counter this disadvantage, we have started to work on inference of annotations, and report on the first results in this direction.

For the on-device techniques, the theory was developed in the first two years of the project (reported in deliverables D6.3 and D6.4), and a prototype implementation of the most promising techniques for JavaCard was developed in the third and final year, and is reported on in deliverable D6.5. The prototype deliverable D6.5 (released together with this deliverable) reports both on the implementation, as well as on the evaluation and validation of the implemented techniques in the POPS case study. For some of the on-device techniques, no implementation was developed, but a rigorous on-paper analysis of the feasibility of applying these techniques to the SecureChange case studies was developed, and is reported on in this deliverable. In particular, we show that the memory consumption for the global policy and non-interference models developed in D6.3 and D6.4 in the POPS case study is acceptable, and we show the applicability of the Security-by-Contract approach developed in D6.3 and D6.4 to the HOMES case study.

Finally, an important objective of Task 6.6 is to show how all techniques developed in WP6 fit together. To this end, we develop an integrated scenario from the POPS case study, that shows how the off-device and on-device techniques work together and address complementary security properties.

In line with the reviewers recommendations, this deliverable is structured as follows: we provide a technical summary of results of approximately 50 pages, and refer to appendices for more detailed information. Three of these appendices correspond to published or submitted scientific papers.

Contents

List of Figures

List of Tables

1. Introduction

The key objective of the SecureChange project is the development of tools and techniques to ensure lifelong compliance to evolving security, privacy and dependability requirements for long-running and evolving software-based systems. To achieve this objective, the project studies and improves the state-of-the-art in several phases of the software lifecycle, including requirements engineering, architectural design, detailed design, implementation, verification and testing.

The focus of Work Package 6 of the project is on the implementation and verification phases, but it includes also the usage phase, as the question how to securely update and evolve running systems is very much in scope for Work Package 6.

Work Package 6 has two main lines of work. The first line (consisting of Tasks 6.1, 6.2 and 6.6) concerns the development of off-device, development-time verification techniques that can ensure the absence of certain classes of vulnerabilities (such as memory safety or concurrency related vulnerabilities), and the development of proof-of-concept tools that implement these techniques for the programming languages used in the SecureChange case studies (JavaCard and C). Such techniques and tools support secure and correct evolution of the code and changes to the code by making implicit developer assumptions explicit in annotations, and by checking that these assumptions are not violated during code evolution.

The second line of work (Tasks 6.3, 6.4, 6.5 and 6.6) concerns the development of on-device, deployment-time verification algorithms. This includes techniques to verify code access control and information flow security of dynamically loaded code, and extensions to the Security-by-Contract paradigm. Such techniques support the secure extensibility of open systems that support deployment of new components at run-time (such as deployment of new applets in the POPS case study or deployment of new bundles in the HOMES case study).

Task 6.6 brings these two lines of work together by showing how the techniques developed for on-device and off-device verification complement each other.

This report deliverable D6.6 is released together with prototype deliverable D6.5. D6.5 delivers the implementation of the on-device techniques for direct and transitive control flow, and reports on their evaluation and validation in the POPS case study. D6.6 reports all the other results in WP6. Hence, the main objective of this final report deliverable in WP6 is threefold:

1. We report on the evaluation and validation of the off-device verification techniques. Three important results are reported on. First, we perform an extensive experiment in the POPS case study: we verify a sizable JavaCard applet implementing an electronic identity card. An experience report is summarized in Section 3.1. Second, we report our progress on the verification of a core security module taken from the HOMES case study (Section 3.2). Finally, since an important conclusion of these first two experiments is that annotation overhead is high, we report on techniques we developed to reduce annotation overhead (Section 3.3).

2. We report on the progress on the on-device verification techniques that were not implemented. (The implemented techniques are reported in D6.5) More specifically, we evaluate the

applicability of the global policy model to POPS (Section 4.1), the applicability of the non-interference model to POPS (Section 4.2) and the applicability of the security-by-contract model for direct control flow to HOMES (Section 4.3).

3. We show how the different techniques developed in WP6 work together by developing an integrated scenario on the POPS case study that comprised four communicating JavaCard applets (Section 2), and demonstrate all implemented WP6 techniques on this scenario.

2. An Integrated Scenario on the POPS Case Study

The SecureChange POPS case study is situated in the application area of Java Card / GlobalPlatform based multi-application smart cards. A more detailed overall description of the POPS case study is given in D1.1 [4]. Work Package 6 focuses on the verification of application-level security properties, and hence focuses on the Java Card applets in the POPS case study (as opposed to the platform implementation itself). POPS is the main case study for Work Package 6, and all technologies developed in WP6 have been evaluated on the POPS case study.

In this chapter, we describe one integrated scenario, consisting of four interacting Java Card applications. In later chapters, we will show how each of the WP6 technologies contributes to verifying security properties for this scenario. This scenario was created from several components. First, we started from the POPS scenario provided by the industrial partner Gemalto/Trusted labs. This scenario includes an ePurse and a jTicket applet as described in Deliverable D1.1. Since the applets in the provided scenario are relatively small, these applets alone do not allow us to demonstrate scalability of our techniques. Hence, we extended the scenario with two significantly larger applets: an anonymized electronic purse applet MyApplet (shared by Gemalto), and an open-source implementation of an electronic identity card (the EidCard applet). Finally, in order to show the full power of the on-device verification techniques, we further increased the interactions taking place between these four applets. The modified applets are called NewEPurse, NewJTicket, NewEidCard and NewMyApplet, respectively. Figure 2.1 depicts the modified applications and interactions among them. This Figure also specifies the package AIDs (PAIDs) of the applets.

We now briefly summarize the functionality of each application used in the integrated scenario and detail how we modified them. Some more details about the content of each application/package are given in Appendix C.

- The ePurse applet comprises functionality of a GlobalPlatform-based electronic purse application. We are using the GlobalPlatform library provided by Gemalto/Trusted Labs for deployment of this applet. This application provides a service debit() in the Shareable interface IEPurseService available for other applications on the card. The Shareable interface in the new applet was renamed into IEPurseServiceDebit. We have also added another Shareable interface IEPurseServiceCredit with two services transaction() and charge() to the applet, functionality of these services is the same as of debit().
- The jTicket applet is a Java Card transport application, its functionality allows to buy tickets through usage of payment services of the ePurse application.
- EidCard is a Belgian electronic ID application. We have added to it the Shareable interface INewEidPoints with the service sharePoints(). Also the NewEidCard applet tries to invoke the service charge() of the NewEPurse applet.

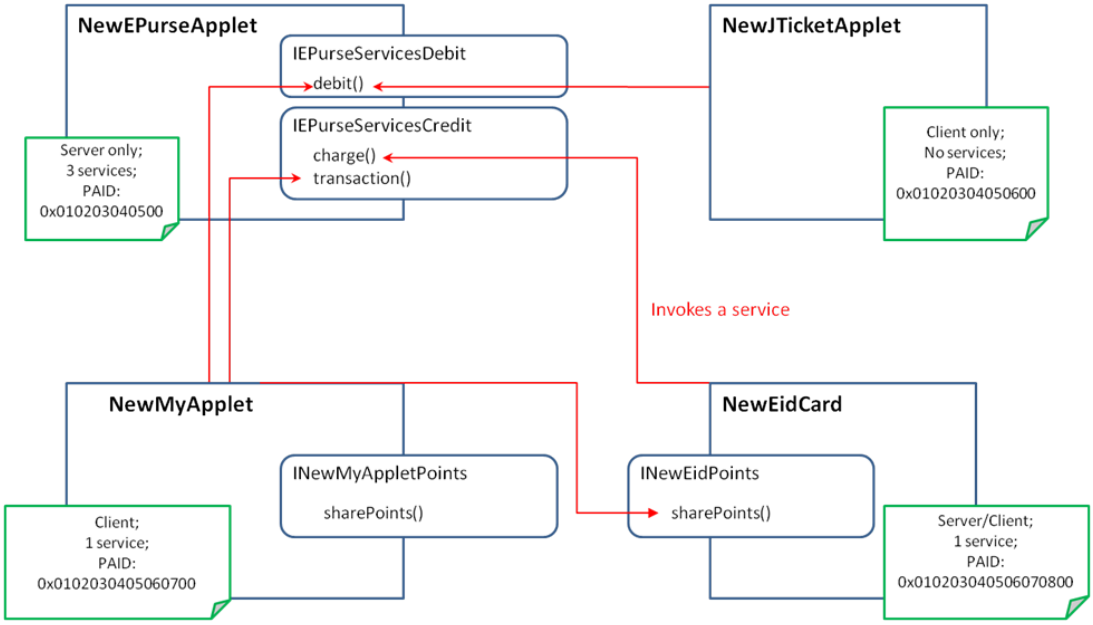

Figure 2.1: Schema of the Integrated Scenario

- MyApplet is an anonymized electronic purse applet shared by Gemalto, which has more complex structure than ePurse. It is GlobalPlatform-based and for deployment we use the GlobalPlatform library provided by Gemalto/Trused Labs. We have added the Shareable interface INewMyAppletPoints() with the method sharePoints(). NewMyApplet application as well tries to access the services transaction() and debit() of the NewEPurse applet and the service sharePoints() of the NewEidCard applet.

Since there is no existing real code yet that makes full use of the interoperability mechanisms provided by the most recent versions of JavaCard / GlobalPlatform, we constructed the scenario above so that it contains substantial challenges for the verification techniques developed in WP6: it includes both newly-developed, greenfield code, as well as existing legacy code. It contains small as well as medium sized (over 1000 lines of code) applets. And it allows for collaborations between several independently developed applets. We should acknowledge however that from an application point of view, the scenario is artificial. The scenario is intended to demonstrate all the technical challenges we want to overcome; it is not intended to be a realistic example application.

The off-device verification technique (i.e. the VeriFast verification tool, reported in D6.2) was evaluated on this scenario. All applets were verified with VeriFast, giving strong guarantees for the protection against Denial of Service security property: the successful verification guarantees the absence of run time exceptions such as array out of bounds exceptions or null pointer dereference exceptions. We discuss the verification of the largest applet (the electronic identity card applet) in more detail in Chapter 3 of this deliverable, and in appendix A we include a published paper that includes a more extensive experience report.

The on-device verification techniques presented in D6.5 were tested on the integrated scenario. For both prototypes, the SxC prototype and the EVE-TCFprototype, the testing procedures included building compliant and non-compliant specifications (contracts and policies) for each applet using the CAP file modification tools developed in WP6. Partners have tried different scenarios of application loading and removal. These extensive testing procedures were conducted in order to validate functionality of the prototypes on real-life applications. This is reported in the tool deliverable that describes these prototypes (D6.5).

Finally, the on-device verification techniques that were developed but not implemented (i.e. the global policy and non-interference models from D6.3 and D6.4) have been evaluated on this case study on paper, and the results of this evaluation are reported in Chapter 4.

3. Off-Device Verification

In this chapter we report on the application of the VeriFast ([30] and D6.2 [48]) software verification tool to the POPS and HOMES case studies as described in D1.1 [4]. We further present research on mitigating the annotation overhead imposed by VeriFast by means of automatically generating parts of the required annotations.

3.1 POPS: Verification of Java Card Applets

A high-level overview of POPS is given in D1.1 [4]. In this section we outline technical aspects of the Java Card platform and report on the successful verification of the eID applet. An extended version of the material of this section has been published in [46] (c.f. App. A).

3.1.1 Java Card Applets

The entry point of each Java Card applet is a class that extends the built-in, abstract class `javacard.framework.Applet`. This class defines a number of methods that are called by the Java Card runtime to interact with the applet. In particular, the class `Applet` defines an abstract method `process` that must be overridden by the subclass. The implementation of `process` forms the core of the applet. More specifically, `process` takes an `APDU` (i.e. a wrapper around a byte array) as input, processes it, and possibly returns an updated `APDU` to the runtime. Typically, the `APDU` contains both information on the action that should be performed by the applet and data associated with that action.

A subclass of `javacard.framework.Applet` is a valid applet only if it declares a static method called `install`. The goal of this method is to create a new applet instance and to register this instance with the runtime. The class `MyApplet` shows the prototypical structure of a Java Card applet.

Listing 3.1: The prototypical structure of an applet.

```
1  class MyApplet extends Applet {
2    public static void install(byte[] arr, short offset, byte length) {
3      MyApplet applet = new MyApplet();
4      // initialize the applet
5      applet.register();
6    }
7
8    public void process(APDU apdu) {
9      // process the apdu
10   }
11 }
```

Transactions. Java Card applets use two types of memory to store data and intermediate results. Fields and objects are stored in persistent EEPROM memory, whereas the stack (and hence local variables) are stored in volatile RAM memory. In addition, the applet can also choose to allocate arrays in RAM memory, because this type of memory is faster and is harder for attackers to read. This complicates things because the smart card may lose power at any time during the computation, which results in the RAM memory being wiped, whereas the EEPROM memory retains the intermediate results.

To preserve consistency of the data stored in persistent memory, Java Card supports transactions. More specifically, the platform defines three methods to interact with the transaction mechanism: `beginTransaction`, `commitTransaction`, and `abortTransaction`. When `beginTransaction` is called, all changes to persistent memory are made conditionally. Only when a call to `commitTransaction` is executed, the changes to the persistent memory are committed atomically. If `abortTransaction` is called instead, or if the card suddenly loses power before calling `commitTransaction`, the persistent memory is restored to its original state (on card boot-up when power is restored). Note that the transaction mechanism does not impact values stored in RAM. Incorrect use of the transaction API, for example calling `beginTransaction` while a transaction is already in progress, results in an exception.

Java Card and VeriFast. VeriFast was originally developed for C and Java programs, but has been modified to also support Java Card applications. The Java language used for Java Card applications is a precise subset of the full Java language, thus adding Java Card support to VeriFast was easy.

Java Card does, however, use a very different class library optimized for smart cards. VeriFast needs to know for every function in the library the pre- and postconditions in order to reason about code. These specifications are placed in a separate file that defines all the classes and methods in the Java Card framework. The specifications are based on the descriptions of these methods in the official Java Card documentation. The actual implementation of these library functions is not checked.

Building the specification file is an incremental process. VeriFast only needs pre- and postconditions for the methods that are actually used by the applications you want to verify. Hence, only a subset of the full Java Card class library has been annotated in the specification file. It is critical that the specifications of library functions is correct; errors in their annotations could lead to errors in the verification process. Therefore, extreme diligence is used when adding new function definitions to the specification.

3.1.2 The eID Applet

The Belgian Electronic Identity Card (eID) was introduced in 2003 as a replacement for the existing non-electronic identity card. Its purpose is to enable e-government and e-business scenarios where strong authentication is necessary. The card has the size of a standard credit card and features an embedded chip. In addition to containing a machine readable version of the information printed on the card, the chip also contains the address of the owner and two RSA key pairs with the corresponding X509 certificates. One key pair is used for authentication, whereas the other key pair can be used to generate legally binding electronic signatures.

The card is implemented on top of the Java Card platform (Classic Edition) and implements the smart card commands as defined in the ISO7816 standard. Unfortunately, the actual code that runs on the eID cards is not publicly available. For our case study, we used an open source, cloned version of the eID applet that implements the same functionality as the real eID card. The source code of the applet can be downloaded from http://code.google.com/

p/eid-quick-key-toolset/. The applet is aimed at developers who wish to interact with eID cards as an easy to use and customizable testing platform.

The eID implementation consists of one large class called `EidCard` and a few other small helper classes. The `EidCard` class inherits from the `Applet` class and encapsulates about 80% of the entire code base. It is a complex class of about 900 lines of code and no less than 38 fields.

Specification of Transaction Correctness. Java Card offers transactions to preserve consistency of the data stored in persistent memory. However, what does it mean for an applet to be consistent? In VeriFast, developers can explicitly write down what fields are part of the persistent state together with the desired consistency conditions. More specifically, the class `Applet` defines an instance predicate called `valid`. Each subclass must override this predicate. The implementation of the predicate given in the subclass defines the consistency conditions for the applet at hand. For example, consider the applet class `ExampleApplet` shown below. The predicate `valid` indicates that both the fields `arr` and `i`, and the array pointed to by `arr` are part of the persistent state (line 6). Moreover, the predicate imposes the consistency condition that `i` is a valid index in `arr` (line 7).

Listing 3.2: The contract of the `process` method, using fractional permissions.

```
1  class ExampleApplet extends Applet {
2    short i;
3    short[] arr;
4    /*@
5    predicate valid() =
6      this.arr |-> ?arr &*& this.i |-> ?i &*&
7      array_slice(arr, 0, ?len, _) &*&
8      0 <= i &*& i < len;
9    @*/
10 }
```

While reading fields is possible at any time, updates to persistent memory should be made inside of a transaction. The permission system used by VeriFast is the key to enforcing this property. More specifically, at the start of the `process` method, no transaction is in progress. As shown in Lst. 3.3, the precondition of `process` contains $1/2$ of the `valid` predicate. This means that the method can read but not update fields included in `valid` (as the method only has one half of the permissions included in `valid`). The predicate `current_applet` is simply a token describing the currently active applet.

Listing 3.3: The contract of the `process` method, using fractional permissions.

```
1  public void process(...)
2     //@ requires current_applet(this) &*& [1/2]valid() &*& ...;
3     //@ ensures current_applet(this) &*& [1/2]valid() &*& ...;
4  {
5     ...
6  }
```

To update the fields of the applet, the method should somehow gain additional permissions (namely the other half of the `valid` predicate). These additional permissions can be acquired by calling `beginTransaction`. In particular, the postcondition of `beginTransaction` shown in Figure 3.4 gives $1/2$ of the `valid` predicate. The `process` method can then merge `[1/2]valid()`

(gained from the precondition of `process`) and `[1/2]valid()` (gained from the postcondition of `beginTransaction`) into `[1]valid()`. The full permission to `valid` gives the applet the right to modify the applet's fields for the duration of the transaction. When calling `commitTransaction`, half of the permissions included in the `valid()` predicate return to the system again. Note that it is impossible to call `commitTransaction` if the applet is in an invalid state (according to the conditions described by `valid`), as the precondition of `commitTransaction` requires the consistency conditions to hold.

Listing 3.4: The declaration of the `beginTransaction` and `commitTransaction` methods

```
1  public static void beginTransaction();
2     //@ requires current_applet(?a) &*& ...;
3     //@ ensures current_applet(a) &*& [1/2]a.valid() &*& ...;
4
5  public static void commitTransaction();
6     //@ requires current_applet(?a) &*& a.valid() &*& ...;
7     //@ ensures current_applet(a) &*& [1/2]a.valid() &*& ...;
```

Inheritance. The ISO7816 standard specifies a mechanism to access files that are stored on a smart card. Three types of files are defined:

1. **Master files** represent the root of the file system. Each smart card contains at most one master file.
2. **Elementary files** contain actual data.
3. **Dedicated files** behave like directories. They can contain other dedicated or elementary files.

To represent this structure, the eID implementation uses helper classes that form a class hierarchy. The root of the hierarchy is the abstract `File` class. This class has two subclasses: `DedicatedFile` and `ElementaryFile`. And finally, the `MasterFile` class inherits from `DedicatedFile`.

When a class is defined in the source code, it can be annotated with a predicate that represents an instance of that class. These predicates can then be used elsewhere to represent a fully initialized instance of that class. Lst. 3.5 shows how a `File` predicate can be defined for the corresponding `File` class. The class consists of two fields, which are also represented in the predicate. The predicate can also contain other information about the class such as invariants.

Listing 3.5: A first definition of the *File* class and predicate.

```
1  public abstract class File {
2     /*@ predicate File(short theFileID, boolean activeState) =
3         this.fileID |-> theFileID &*&
4         this.active |-> activeState; @*/
5
6     private short fileID;
7     protected boolean active;
8
9     ...
10 }
```

The `ElementaryFile` class redefines the `File` predicate as shown in lines 2-4 of Lst. 3.6. A `File` predicate that is associated with an `ElementaryFile` class is defined as an `ElementaryFile` predicate where three of the five parameters are undefined.

The definition of the `ElementaryFile` predicate (lines 5-13) consists of a link to the `File` predicate defined in Lst. 3.5 and some extra fields and information that are specific to elementary files.

Listing 3.6: A first definition of the *ElementaryFile* class and predicate.

```
1  public final class ElementaryFile extends File {
2     /*@ predicate File(short theFileID, boolean activeState) =
3           ElementaryFile(theFileID, ?dedFile, ?dta,
4             activeState, ?sz); @*/
5     /*@ predicate ElementaryFile(short fileID,
6             DedicatedFile parentFile, byte[] data,
7             boolean activeState, short size) =
8           this.File(File.class)(fileID, activeState) &*&
9           this.parentFile |-> parentFile &*&
10          this.data |-> data &*& data != null &*&
11          this.size |-> size &*&
12          array_slice(data, 0, data.length, _) &*&
13          size >= 0 &*& size <= data.length; @*/
14
15    private DedicatedFile parentFile;
16    private byte[] data;
17    private short size;
18
19    ...
20 }
```

When an object is cast from the `File` to the `ElementaryFile` class (or vice versa), the corresponding predicate on the symbolic heap must be changed as well. We 'annotated' this by adding the methods that are defined in Lst. 3.7 to the `ElementaryFile` class and calling these methods when required. Obviously, this solution is far from elegant because it requires adding calls to stub functions in the code of the applet. The most recent version of VeriFast supports annotating this behavior as lemma methods (i.e. methods defined inside an annotation), removing the requirement of modifying the applet's code.

Listing 3.7: Functions to cast predicates.

```
1  public void castFileToElementary()
2     //@ requires [?f]File(?fid, ?state);
3     //@ ensures [f]ElementaryFile(fid, _, _, state, _);
4  {
5     //@ open [f]File(fid, state);
6  }
7
8  public void castElementaryToFile()
9     //@ requires [?f]ElementaryFile(?fid, ?dedFile, ?dta, ?state, ?sz);
10    //@ ensures [f]File(fid, state);
11 {
12    //@ close [f]File(fid, state);
13 }
```

One problem that occurs with the methods presented in Lst. 3.7 is that information is lost when an `ElementaryFile` is cast to a `File` and then back again to an `ElementaryFile`. This

loss of information happens in the `castFileToElementary` method where three parameters are left undefined.

There are some instances in the eID applet where this loss of information is problematic. The solution was to extend the `File` and `ElementaryFile` predicates to contain an extra parameter that can store any information. The result can be seen in Lst. 3.8. Line 3 shows the definition of this extra parameter. In the case of the `File` class, no extra information is kept and the parameter is defined to be empty (denoted as 'unit' on line 5). Similarly, line 22 defines the parameter to be empty for the `ElementaryFile` predicate, because all state information that can be stored in the predicate is fully defined by the other parameters.

Line 14 shows the case where the predicate needs the extra parameter to store additional information about the object. In this case, the `info` parameter stores a quad-tuple of extra information that can be used to correctly initialize the embedded `ElementaryFile` predicate without losing information.

Listing 3.8: A more complete definition of the *File* and *ElementaryFile* predicates that supports downcasting.

```
1  public abstract class File {
2     /*@ predicate File(short theFileID, boolean activeState,
3              any info) =
4           this.fileID |-> theFileID &*&
5           this.active |-> activeState &*& info == unit; @*/
6
7     ...
8  }
9
10 public final class ElementaryFile extends File {
11    /*@ predicate File(short theFileID, boolean activeState,
12             quad<DedicatedFile, byte[], short, any> info) =
13          ElementaryFile(theFileID, ?dedFile, ?dta, activeState,
14          ?sz, ?ifo) &*& info == quad(dedFile, dta, sz, ifo); @*/
15    /*@ predicate ElementaryFile(short fileID,
16             DedicatedFile parentFile, byte[] data, boolean activeState,
17             short size, any info) =
18          this.File(File.class)(fileID, activeState, _) &*&
19          this.parentFile |-> parentFile &*&
20          this.data |-> data &*& data != null &*& this.size |-> size
21          &*& array_slice(data, 0, data.length, _) &*&
22          size >= 0 &*& size <= data.length &*& info == unit; @*/
23
24    ...
25 }
```

3.1.3 Evaluation

The main goal of this case study was to see how practical it is to use VeriFast to annotate a Java Card applet that is more than a toy project. It gives us an idea of how much the annotation overhead is, where we can improve the tool, and whether we can actually find bugs in existing code using this approach.

Annotation Overhead. The more information the developer gives in the annotations about how the applet should behave, the more VeriFast can prove about it. It is up to the developer to choose whether he wants to use VeriFast as a tool to only detect certain kinds of errors (unexpected exceptions and incorrect use of the API), or whether he wants to prove full functional correctness of the applet. Both operation modes are supported by the tool. For the Java Card applets, we

used the annotations to prove that the applet does not contain transaction errors, performs no out of bounds operations on buffers, and never dereferences null pointers. While we have used VeriFast to verify full functional correctness of sequential and fine-grained concurrent data structures [29], specifying and verifying full functional correctness of JavaCard applets is future work.

The eID applet and helper classes consist of 1,004 lines of Java Card code. In order to verify the project, we added 802 lines of VeriFast annotations (or about four lines of annotations for every five lines of code). The majority of these annotations were `requires/ensures` pairs (88 pairs, one for each method) and `open` and `close` statements (99 and 112 instances respectively). Remarkably, only 8 predicates are defined throughout the entire code base, reflecting the design decision of the authors of the applet to write most of it as one huge class file.

Another type of annotation overhead is the time required to actually write the annotations. The verification of the eID applet was performed by a senior software engineer without prior experience with the VeriFast tool, but with regular opportunities to consult VeriFast expert users during the verification effort. We did not keep detailed effort logs. An estimate of the effort that was required is 20 man-days. This includes time spent learning the VeriFast tool and the Java Card API specifications.

Bugs and Other Problems. Because the eID applet in our case study is aimed at developers, the authors did not spend a lot of time worrying about card tearing. This is demonstrated by the fact that they did not use the Java Card transaction system at all. Using VeriFast, we found 25 locations where a card tear could cause the persistent memory to enter an inconsistent state.

Three locations were found where a null pointer dereference could occur. An additional three class casting problems were found, where a variable holding a reference to the selected file (of type `File`) was cast to an `ElementaryFile` instance. These bugs could be triggered by sending messages with invalid file identifiers to the smart card. Seven potential out of bounds operations were also found in the code. These bugs could be triggered by sending illegal messages to the smart card.

VeriFast Strengths. Compared to other program verifiers that target Java Card [37, 19], VeriFast has two advantages: speed and soundness. That is, VeriFast usually reports in only a couple of seconds (usually less) whether the applet is correct or whether it contains a potential bug. Secondly, if VeriFast deems a program to be correct, then that program is guaranteed to be free from null pointer and array index out of bounds exceptions, and API usage and assertion violations.

A feature that proved to be crucial in understanding failed verification attempts is VeriFast's symbolic debugger. The symbolic debugger can be used to diagnose verification errors by inspecting the symbolic states encountered on the path to the error. For example, if the tool reports an array indexing error, one can look at the symbolic states to find out why the index is incorrect. This stands in stark contrast to most verification condition generation-based tools that simply report an error, but do not provide any help to understand the cause of the error.

3.2 HOMES: Verification of a Core Security Module in C

In this section we report on the verification of the C implementation of a Policy Enforcement Point (PEP) with VeriFast. PEP is a program intended to run on Home Gateways. The case study is provided by TID as part of the HOMES scenario described in D1.1 [4]. Since WP6 obtained the case study only in November 2010 and substantial development effort was required to improve VeriFast's support for the C language, the case study is still incomplete.

3.2.1 The PEP Case Study

The PEP program consists of approximately 1700 lines of C code. The program is designed to run on embedded Linux-based home gateways and facilitates the application of security policies in Network Admission Control scenarios. More specifically, for an authenticated network device, PEP will receive an access policy from a Policy Decision Point. This policy is then put in place by configuring the gateway's network interfaces and firewall rules accordingly. This scenario is extensively described in D1.1 [4].

Progress Time-line. The case study is released to WP6 in November 2010. An initial assessment of the feasibility of applying VeriFast to PEP is carried out in the following month. We conclude that verifying PEP is viable. Yet, VeriFast's support for C needs to be extended substantially to support C language constructs that were not available in VeriFast back then. Work on extending VeriFast starts in March 2011, and extended case studies on PEP are carried out in September and October 2011, followed by an attempt to verify the unmodified PEP sources. This effort is still ongoing.

Case Study and Properties. The PEP implementation is split into 9 C source files and 8 C header files. In total, 53 functions are implemented. The core module of PEP is the file `pep.c`, which comprises of 658 lines of code in 13 functions. Although PEP itself is relatively small, it involves a range of Linux libraries that increase the complexity of the verification effort substantially. These libraries are:

libpcap provides a network packet filtering mechanism

libdumbnet provides a simplified, portable interface to several low-level networking routines, including creation and submission of network packages, and the configuration of network interfaces and firewall rules

libssl provides Secure Socket Layer encryption and authentication of network traffic

For a thorough verification of the absence of runtime errors and for proving functional correctness, the entire PEP code and the APIs of the above libraries would have to be annotated. Since this was infeasible given the short period of time available for conducting the case study, we focus on annotating `pep.c` and the PEP header files only. In order to integrate the libraries we have created a single header file `sys_includes.h` that contains a 250-lines excerpt of the libraries' APIs and the Linux system header files.

An important goal of the exercise is to conduct the verification with as few modifications to the sources as possible. This is important for communicating bugs reported by WP6 to the developers of the PEP program, i.e. TID. Since verifying PEP for full functional correctness is not feasible in the time available, we currently aim at proving that the PEP implementation does not perform illegal operations such as dividing by zero or illegal memory access. PEP is also multi-threaded. Thus, a second objective is to verify that the PEP implementation is free of data races. In the following we present and discuss the necessary extensions to VeriFast and the progress of our verification case study.

3.2.2 VeriFast Extensions

The VeriFast distribution (c.f. D6.2 [48] and [30]) is accompanied by a tutorial document and a range of examples that illustrate how C programs are annotated and verified. To enable the verification of the PEP program, substantial development effort went into extending VeriFast so as to support arrays and structs that are not dynamically allocated. In this section we outline annotations for these programming constructs.

Arrays. The PEP program employs a lot of arrays as buffers for constructing strings and network packets. In an initial attempts to verify PEP, we replaced these arrays with dynamically allocated heap objects. This, however, implies modifications in the public header files of the system libraries (e.g. for arrays inside structs), introduces a range of new program behavior that is to be verified, and makes it impossible to compile, link and execute the code.

Support for global and local arrays with array initializers, and an extended range of C syntax for accessing arrays, including the typical `array[position]`, was implemented in VeriFast. In order to access dynamically allocated chunks in a similar way as static arrays, we provide conversion lemmas such as `chars_to_chararray` and `chararray_to_chars`. Interactions between static and dynamic arrays are illustrated in Lst. 3.9. The listing presents an annotated program that initializes all elements of a dynamically allocated array with the content of one element taken from a global static array. To verify the program, a number of lemma definitions, amounting to a total of 27 lines of annotations are required. These lemmas are available in the examples section of the recent VeriFast distribution.

Listing 3.9: C Arrays in VeriFast.

```
1  static char src[20] = "Hello, world\n";
2
3  int main (int argc, char **argv) //@ : main_full(carrays)
4  //@ requires module(carrays, true);
5  //@ ensures result == 0;
6  { //@ open_module();
7    char *dst; int i = 0;
8    dst = malloc (42); if (dst == 0) { abort(); }
9  //@ chars_to_chararray(dst, 42);
10   while (i < 42)
11 /*@ invariant 0 <= i && i <= 42
12     &*& array<char>(src, 20, sizeof(char), character, ?srcelems)
13     &*& array<char>(dst, 42, sizeof(char), character, ?ndstelems)
14     &*& forall_eq(take(i, ndstelems), nth(0, srcelems)) == true; @*/
15    { dst[i] = src[0];
16 //@ take_plus_one(i, update(i, nth(0, srcelems), ndstelems));
17     i++; }
18 //@ chararray_to_chars(dst);
19   free (dst);
20 //@ close_module();
21 //@ leak module(carrays, _);
22   return (0); }
```

Structs. Similar to arrays, PEP and the system libraries make use of static structs. Further features previously unsupported by VeriFast include structs fields of type struct or array, and struct initializers. In initial feasibility studies on PEP, we replaced static structs by their dynamically allocated equivalents. Struct fields inside these structs were replaced by pointers and a dedicated function would allocate and initialize all variables as necessary. Further extensive modifications were necessary to avoid the use of `memset()` and `memcpy()` to modify the content of structs.

Again, the resulting code would not execute correctly due to incompatibilities with library functions. In consequence, full support for structs has been added to VeriFast, enabling a range of case studies beyond the scope of PEP.

3.2.3 Verification Progress

As outlined in the previous section, a number of feasibility studies was conducted to familiarize with the case study and its requirements. In this process, modified versions of PEP's functions were annotated and verified. The results of these case studies yield no reliable bug reports as the analyzed versions of the PEP program do neither comply with the system libraries' APIs nor with the PEP-internal APIs. TID, the stakeholder of the case study, was requested to supply WP6 with test cases (e.g. unit tests for the core functions). These may have enabled us to re-engineer a PEP program that does correctly implement the PEP functionality without employing programming constructs unsupported by VeriFast. Unfortunately, the program is reported not to be formally tested. Subsequently, VeriFast is extended to enable us to verify a largely unmodified version of the PEP program.

Ongoing Verification Effort. Currently, 8 (out of 13) functions of PEP's core file, `pep.c`, are annotated and verified with no or only minor modifications to the actual code. In particular, we removed calls to `printf()`[1] since functions with a variable number of arguments are currently unsupported. We further use a custom script to "preprocess" the source file so as to add some VeriFast-specific `#include`-directives and to perform macro expansion[2]. Importantly, we still use incomplete stub-annotations for some internal APIs and for the system libraries.

Until now, a total of 450 lines of annotations is produced for the final case study. This does not include generic lemmas and predicates. We conjecture the annotation overhead to be slightly below or on par with the number of lines of code in the PEP program. The verification of PEP is being conducted by a researcher with a background in specification techniques and software verification, but with no specific experience related to VeriFast and separation logic. The time invested in the verification project, including time to familiarize with the verification tool, conduct feasibility studies, extend the tool and carry out the current experiments, amounts to roughly 5 person months. The time dedicated exclusively to the ongoing study is less than one person month.

Verification Results. Since the PEP program is production code that implements a security module which has potentially been deployed with thousands of home gateways, our expectations to find critical bugs were relatively low.

Nevertheless, even though our verification effort is not yet finished, we already discovered 5 NULL-pointer bugs. Of those, 4 are the results of the unguarded use of `malloc()` while the 5th is due to an unchecked dereference of a pointer returned by `fopen()`. If it can be guaranteed that the home gateways PEP is deployed on, never run out of memory, the bugs related to `malloc()` may never be triggered. The `fopen()`-error, however, can cause the program to crash in practice, i.e. when the configuration file is not present or not readable.

In addition, we have a strong indication for a race condition on the global variable `brmac`. This potential bug was first discovered through manual inspection in the initial feasibility studies. It cannot be confirmed yet because the annotations of the code are still incomplete.

[1] In PEP, `printf()` is only used to output status messages and debug information.
[2] Support for preprocessor directives has been added to VeriFast after work on the final case study started.

3.3 Automatic Annotation Inference

VeriFast heavily relies on hand-crafted annotations that contribute to the tool's efficiency. Yet, the need for annotations may also render the VeriFast cumbersome to use. This issue has previously been criticized by the stakeholders of the POPS and HOMES case studies. To quantify the annotations needed, we present statistics on example programs from the VeriFast distribution in Tab. 3.1. The second (*LOC*) and third (*LOAnn*) columns denote the lines of C/Java code and annotations in the examples. The numbers given in parentheses correspond to the number of open and close statements, respectively, which will be further explained in Sec. 3.3.1. The fourth (*LoAnn/LOC*) column shows the ratio of annotation overhead. In this section we investigate three techniques to mechanize the generation of annotations. An extended abstract of our work has been published in [51] (c.f. App. B).

Table 3.1: Statistics on VeriFast's annotation overhead

	LOC	LOAnn	LoAnn/LOC
stack (C)	88	198 (18/16)	2.3
sorted binary tree (C)	125	267 (16/23)	2.1
bank example program (C)	405	127 (10/22)	0.31
chat server (C)	130	114 (20/26)	0.88
chat server (Java)	138	144 (19/28)	1.0
game server (Java)	318	225 (47/63)	0.71

We distinguish two layers in VeriFast: *(i)* VeriFast's core, which requires annotations to perform verification. This core must be as small and maintainable, as the verification's soundness relies on it. *(ii)* The automation layer generates a substantial portion of annotations to be fed to VeriFast's core. The aim of this approach is to maximize robustness such that we can freely experiment with different automation techniques without having to worry about introducing unsound elements – the annotations are fully verified by the core.

To illustrate annotations and annotation inference, we introduce a running example a fully annotated list-copying function in Lst. 3.11. The `copy` function comprises of 12 C statements and 31 annotation statements, not counting lemmas that can be shared by multiple function contracts. We now discuss how some of these annotations can be generated automatically.

We focus on verifying memory safety. Thus, we employ two simple predicates `Node` and `LSeg` (Lst. 3.10) that describe the structure of the list. The code in Lst. 3.11 makes use of a function `new()` that produces a new `Node(result, 0)` and always succeeds. The function is defined in terms of `malloc` and `abort`s on allocation failure. `NoCycle`, `Distinct`, `AppendLSeg` and `AppendNode` are lemmas whose contracts are shown in Figure 3.12.

Listing 3.10: `Node` and `LSeg` predicates

```
1  predicate Node(struct list* P, struct list* Q) =
2   P != 0 &*& malloc_block_list(P) &*& P->next  |-> Q &*& P->value |-> ?v;
3
4  predicate LSeg(struct list* P, struct list* Q) =
5   P == Q ? emp : Node(P, ?R) &*& LSeg(R, Q);
```

3.3.1 Auto-Open and Auto-Close

As can be seen, a lot of annotations perform opening and closing predicates. Our statistics in Tab. 3.1 indicate the situation is similar for other programs, rendering `open` and `close` interesting candidates for automation. Whenever the execution of a statement fails, the verifier could inspect the current execution state and attempt opening or closing predicates to identify the right heap fragments to be produced.

Listing 3.11: Copying linked lists

```
1  struct list* copy(struct list* xs)
2   //@ requires LSeg(xs, 0);
3   //@ ensures LSeg(xs, 0) &*& LSeg(result, 0);
4  {
5   if ( xs == 0 ) {
6    //@ close LSeg(0, 0);                                  // a
7    return 0; }
8   else {
9    struct list* ys = new();
10   //@ open LSeg(xs, 0);
11   //@ open Node(xs, _);                                  // a
12   //@ open Node(ys, 0);                                  // a
13   ys->value = xs->value;
14   struct list *p = xs->next, *q = ys;
15   //@ close Node(ys, 0);                                 // a
16   //@ close Node(xs, p);                                 // a
17   //@ NoCycle(xs, p);
18   //@ close LSeg(p, p);                                  // a
19   //@ close LSeg(xs, p);                                 // a
20   //@ close LSeg(ys, q);                                 // a
21   while ( p != 0 )
22    //@ invariant LSeg(xs,p) &*& LSeg(p,0) &*& LSeg(ys,q) &*& Node(q,0);
23   {
24    //@ struct list *oldp = p, *oldq = q;
25    struct list* next = new();
26    //@ Distinct(q, next);
27    //@ open Node(q, 0);                                   // a
28    q->next = next; q = q->next;
29    //@ close Node(oldq, q);                               // a
30    //@ open LSeg(p, 0);
31    //@ assert Node(p, ?pn);
32    //@ NoCycle(p, pn);
33    //@ open Node(p, _);                                   // a
34    //@ open Node(q, 0);                                   // a
35    q->value = p->value; p = p->next;
36    //@ close Node(q, 0);                                  // a
37    //@ close Node(oldp, p);                               // a
38    //@ AppendLSeg(xs, oldp); AppendNode(ys, oldq);
39   }
40   //@ open LSeg(p, 0);                                   // a
41   //@ NotNull(q);                                        // b
42   //@ close LSeg(0, 0);                                  // a
43   //@ AppendLSeg(ys, q);
44   //@ open LSeg(0, 0);                                   // a
45   return ys;
46  }
47 }
```

Let us assume that we are reading from the `next` field of a variable `x`, which requires a heap fragment matching `x->next |-> v`. However, only `Node(x, y)` is available. Without automation, verification would fail, but instead, the verifier could try opening `Node(x, y)` and find out that this results in the required heap fragment. Of course, given that predicates can be defined recursively, this "search" may not terminate. Therefore, heuristics are needed to guide automation.

We implemented automatic opening and closing of predicates in VeriFast: VeriFast keeps a directed graph whose nodes are predicates and whose arcs indicate how predicates are related to each other. For example, there exists an arc from `LSeg` to `Node`, meaning that opening an `LSeg` yields a `Node`. However, this depends on whether the `LSeg` does represent the empty list.

Listing 3.12: Lemmas

```
1  lemma void NoCycle(struct list* P, struct list* Q)
2    requires Node(P, Q) &*& LSeg(Q, 0);
3    ensures  Node(P, Q) &*& LSeg(Q, 0) &*& P != Q;
4  lemma void Distinct(struct list* P, struct list* Q)
5    requires Node(P, ?PN) &*& Node(Q, ?QN);
6    ensures  Node(P, PN)  &*& Node(Q, QN) &*& P != Q;
7  lemma void AppendLSeg(struct list* P, struct list* Q)
8    requires LSeg(P, Q) &*& Node(Q, ?R) &*& Q != R &*& LSeg(R, 0);
9    ensures  LSeg(P, R) &*& LSeg(R, 0);
10 lemma void AppendNode(struct list* P, struct list* Q)
11   requires LSeg(P, Q) &*& Node(Q, ?R) &*& Node(R, ?S);
12   ensures  LSeg(P, R) &*& Node(R, S);
```

To express this dependency, we label the arcs with the required conditions. The same conditions can be used to encode the relationships between the arguments of both predicates. For the predicate definitions from Lst. 3.10, the graph would contain the following:

$$\mathrm{LSeg}(a,b) \xrightarrow[a=p]{a\neq b} \mathrm{Node}(p,q) \xrightarrow{p=x} x \to \mathrm{next} \mapsto y$$

During verification, some operation may require the presence of a `Node(p, q)` heap fragment. If this is missing, two possible solutions are considered: we can either attempt to perform an auto-open on an `LSeg(p, b)` for which we know that `p != b`, or try to close `Node(p, q)` if there happens to be a `p->next |-> ?` on the current heap. This yields a considerable decrease of annotations in Lst. 3.11: each `open` or `close` indicated by `// a` (17 out of 31) is inferred automatically by VeriFast.

3.3.2 Autolemmas

Lemmas only have to be defined once. Thus, automatic generation would only yield a limited reduction in annotations. Yet, lemma application occurs multiple times, which is where our focus lies. Currently, we have implemented one very specific and admittedly somewhat limited way to automate lemma application. While automatic opening and closing of predicates is only done when the need arises, VeriFast will try to apply all lemmas regarding a predicate `P` each time `P` is produced, in an attempt to accumulate as much extra information as possible. This immediately gives rise to limitations with respect to efficiency and potentially making the execution state unusable. The latter happens, for example, if the `AppendLSeg` lemma were applied indiscriminately, `Node`s would be absorbed by `LSeg`s, effectively throwing away potentially crucial information (in this case, we "forget" that the list segment has length 1.) To prevent this, autolemmas are not allowed to modify the symbolic state, but instead may only extend it with extra information. To counteract performance issues, lemmas need to be explicitly declared to qualify for automatic application, and they may only depend on one heap fragment.

In the case of our example, only one lemma qualifies for automation: `NotNull`. Thus, every time a `Node(p, q)` heap fragment is added to the heap, VeriFast will immediately infer that `p != 0`. Since we only needed to apply this lemma once, we can only decrease the number of annotations by the line indicated with `// b` in Lst. 3.11.

3.3.3 Automatic Shape Analysis

Our third approach for reducing annotations focuses solely on shape analysis [17] and has the potential to automatically generate all necessary annotations for a number of programming constructs, including postcondition and loop invariants. To apply shape analysis, we need to determine the initial program state. We achieve this by requiring the verification engineer to provide preconditions, effectively putting barriers on how far a bug's influence can reach.

Table 3.2: Automatic shape analysis: finding a fixed point

without abstraction	with abstraction
`Node(p', p) &*& LSeg(p, 0)`	`LSeg(p', p) &*& LSeg(p, 0)`
`Node(p', p1) &*& Node(p1, p) &*& LSeg(p, 0)`	`LSeg(p', p) &*& LSeg(p, 0)`

Table 3.3: Reduction of annotation overhead

C-code	#code	A	B	C	D	lemma	A	B	C
`length`	10	12	9	9	1	`Distinct`	9	7	7
`sum`	11	11	7	7	1	`NotNull`	7	6	6
`destroy`	9	6	4	4	1	`AppendNode`	19	16	16
`copy`	23	32	15	14	1	`AppendLSeg`	27	19	18
`reverse`	12	9	5	5	1	`AppendNil`	9	7	6
`drop_last`	28	28	13	13	1	`NoCycle`	11	10	9
`prepend`	7	5	3	3	1				
`append`	13	20	11	11	1				

	#code	A	B	C	D
total	113	205	132	128	8

Our implementation of shape analysis is based on [17]. The idea is very similar to what has been explained earlier in Sec. 3.3.1: during symbolic execution of a function, it will open and close the predicates as necessary to satisfy the precondition of the operations it encounters. However, the analysis has a more thorough understanding of the lemmas. That is, it will know in what circumstances lemmas need to be applied. A good example of this is the inference of the loop invariant where shape analysis uses the lemmas to abstract the state, which is necessary to prevent the symbolic heap from growing indefinitely while looking for a fixpoint. Consider the following pseudocode: $p' := p; \mathbf{while}\ p \neq 0\ \mathbf{do}\ p := p{\rightarrow}\mathrm{next}\ \mathbf{end}$. Initially, the symbolic heap contains `LSeg(p, 0)`. To enter the loop, `p` needs to be non-null, hence it is a non-empty list and can be opened up to `Node(p', p1) &*& LSeg(p1, 0)`. During the next iteration, `p1` can be null or non-null. Thus, every iteration adds the possibility of an extra node. This way, we'll never find a fixed point. Performing abstraction will fold nodes back into `LSeg`s. The difference is shown in Tab. 3.2. One might wonder why the abstraction doesn't also merge both `LSeg`s into a single `LSeg`. The reason for this is that the local variable `p` points to the start of the second `LSeg`: folding would throw away information deemed important.

For our purposes, the algorithms from [17] have been extended so as to also generate annotations to facilitate integration with VeriFast. To prove properties other than memory safety, further annotations may be added. In our example, shape analysis is able to deduce all `open` and `close` annotations, the lemma applications, the loop invariant and the postcondition. Hence, the number of necessary annotations for Lst. 3.11 is reduced to 1, namely the precondition.

3.3.4 Evaluation

In order to get a better idea of by how much we managed to decrease the number of annotations, we wrote a number of list manipulation functions. There are four versions of the code: *(A)* A version with all annotations present. *(B)* An adaptation of (A) where we enabled auto-open and auto-close. *(C)* A version where we take (B) and make `NotNull` an autolemma (Sec. 3.3.2). *(D)* Finally, a minimal version with only the required annotations to make our shape analysis (Sec. 3.3.3) able to verify the code. Tab. 3.3 shows how the annotation line counts relate.

4. On-Device Verification

In this chapter we report on the on-device information protection techniques developed by WP6, but not yet implemented in the prototypes. First we estimate the costs of the implementations of the *global policy* model (Section 4.1) and the *non-interference* model (Section 4.2) in the POPS case study. Finally, we discuss the application of SxC to the HOMES case study (Section 4.3).

4.1 Toward Implementing the Global Policy Model for Smart-Cards

In the deliverables D6.3 (Chapter 5) and D6.4 (Chapter 4), we described the *global policy* model. The purpose of this section is to give some details on its practicable implementation for JavaCard smart-cards, especially regarding content and size of the required meta-data to be embedded in CAP files (Section 4.1.1) and those persistently stored on-device (Section 4.1.2). Evaluation of memory consumption on the POPS scenario is described in Section 4.1.3.

4.1.1 Off-Device Computation and Embedding of Methods Footprints in CAP Files

Figure 4.1 gives an overview of how methods footprints (Definitions 5.4.2 and 5.5.1 of the deliverable D6.3) can be embedded in their related CAP file thanks to a *Custom Component*. The process is completely analog to the one implemented for the transitive control flow model (*c.f.*deliverable D6.5): an off-device tool analyzes the content of an input CAP file according to an input policy written in a DSL language, and produces a new CAP file with the same application content plus a *Custom Component* containing methods footprints, intermediate footprints and footprints of external methods referenced (invoked, overridden, implemented) needed for on-device verification (Section 5.6.2 of the deliverable D6.3).

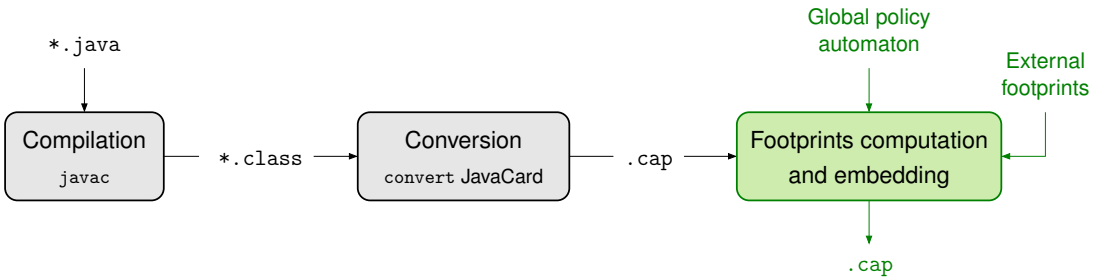

Figure 4.1: Schema of the off-device process for embedding methods footprints in a CAP file.

A Custom Component for Embedded Methods Footprints

The *Global Policy Custom Component* (GPCC) to be embedded in CAP files is a *Custom Component* of the CAP file format (*c.f.*JCVM 2.x specifications) with the data structures displayed

in Figure 4.2. The fields of those structures have the following meaning.

The `global_policy_component` structure describes a GPCC:

- `tag` contains the tag value (between 128 and 255 inclusive, as described in the ISO 7816-5) which permits to identify the GPCC;
- `size` indicated the number of bytes in the `global_policy_component` structure, excluding the `tag` and `size` items. The value of the `size` field must be greater than 0;
- `external_footprints_size` represents the size in bytes of the `external_footprints` field;
- `external_footprints` contains all the *believed footprints* of external methods (*i.e.*defined in other packages) directly referenced (*i.e.*invoked, overridden or implemented) in this package (Section 5.6.2 of the deliverable D6.3);
- `class_footprints_size` represents the size in bytes of the `class_footprints` field;
- `class_footprints` contains a `internal_class_footprint` entry for each class and each interface defined in this package.

The `external_package_footprint` structure describes the methods footprints coming from an imported package and explicitly referenced (*i.e.*invoked, overridden or implemented) in this package:

- `package_aid` contains the package AID that permits to identify the package from which external footprints are coming from;
- `class_footprints_size` contains the size in bytes of the `class_footprints` field;
- `class_footprints` contains an `external_class_footprint` entry for each class and each interface of this package containing a method referenced in the current package.

The `external_class_footprint` structure describes the methods footprints of a class or an interface defined in an imported package:

- `class_token` represents the class token of the current class (or interface); its value cannot be `0xFF` as it must correspond to a class or to an interface visible outside of the package in which it is defined;
- `method_footprints_size` contains the size in bytes of the `method_footprints` field;
- `method_footprints` maps a believed footprint to each method of the current external class (or interface) that is referenced in this package.

The `external_method_footprint` structure describes the footprint of an external method referenced in this package:

- `method_token` represents the static/virtual/interface method token of this method;
- `complete_footprint` contains the binary encoded footprint of this method (Section 5.6.2 of the deliverable D6.3)[1].

The `internal_class_footprint` structure describes the methods footprints of a class or an interface defined in this package:

[1] Footprints of external methods must be the ones used to compute footprints of methods defined in this package.

```
global_policy_component {
    u1 tag;
    u2 size;
    u2 external_footprints_size;
    external_package_footprint external_footprints[];
    u2 class_footprints_size;
    internal_class_footprint class_footprints[];
}

external_package_footprint {
    struct {
        u1 length;
        u1 value[length];
    } package_aid;
    u2 class_footprints_size;
    external_class_footprint class_footprints[];
}

external_class_footprint {
    u1 class_token;
    u2 method_footprints_size;
    external_method_footprint method_footprints[];
}

external_method_footprint {
    u1 method_token;
    footprint_t complete_footprint;
}

internal_class_footprint {
    u2 classref;
    u1 class_token;
    u2 method_footprints_size;
    internal_method_footprint method_footprints[];
}

internal_method_footprint {
    u1 bitfield;
    u1 method_token; /* present according to bitfield */
    u2 method_offset; /* present according to bitfield */
    footprint_t complete_footprint;
    u2 intermediate_footprints_size;
    intermediate_footprint intermediate_footprints[];
}

intermediate_footprint {
    u2 pc;
    footprint_t footprint;
}
```

Figure 4.2: Data structures of the *Global Policy Custom component* (GPCC), a custom component of the CAP file format for the implementation of the global policy model on JavaCard smart cards.

- `classref` contains the location (*i.e.*the offset) in the Class Component (*c.f.*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_footprints_size` contains the number of entries in the `method_footprints` field;
- `method_footprints` maps to each method of the current class (or interface) its footprint.

The `internal_method_footprint` structure describes the footprint of a method defined in this package:

- `bitfield` is a mask of modifiers describing the current method:

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/virtual/interface method token of this method; this item is present iff the method is visible, according to `bitfield`;
- `method_offset` represents a byte offset into the info item of the Method Component (*c.f.*JCVM specifications) if the method is implemented (*i.e.*not an abstract method or a method definition in an interface), according to `bitfield`;
- `complete_footprint` contains the binary encoded footprint of the current method;
- `intermediate_footprints_size` contains the size in bytes of the `intermediate_footprints` field;
- `intermediate_footprints` contains all the intermediate footprints (*i.e.proof annotations* described in Section 5.6.2 of the D6.3) of the current method used to compute the complete footprint of this method, but also to verify its complete footprint on-device.

The `intermediate_footprint` structure describes an intermediate footprint of a method, that is an incomplete footprint of a method attached to an instruction of the method targeted by a jump (instruction following an `invoke`, a conditional block, ...):

- `pc` is the offset of the instruction in the bytecode of the method to which this intermediate footprint is attached;
- `footprint` is the binary encoded footprint.

The structure `footprint_t` is a flat bit array which contains a binary encoded footprint (Section 5.6.2 of the deliverable D6.3). Its size depends on the number of states in the automaton describing the set of forbidden sequences of method calls.

4.1.2 On-Device Verification and Storage of Methods Footprints

The on-device enforcement of a global policy requires to verify at loading-time that all methods footprints of each incoming package (application, library) are correct with respect to their bytecode, and respect the global policy. In order not to be bypassed, the verification process must occur before the package is concretely installed and used by other packages installed on the card, as depicted on Figure 4.3. Methods footprints of successfully verified and installed packages are stored persistently into dedicated repositories on-device across installations, and cleaned upon successful uninstallations. Integration on-device is rigorously similar to the one implemented for the transitive control flow model described in the deliverable D6.5.

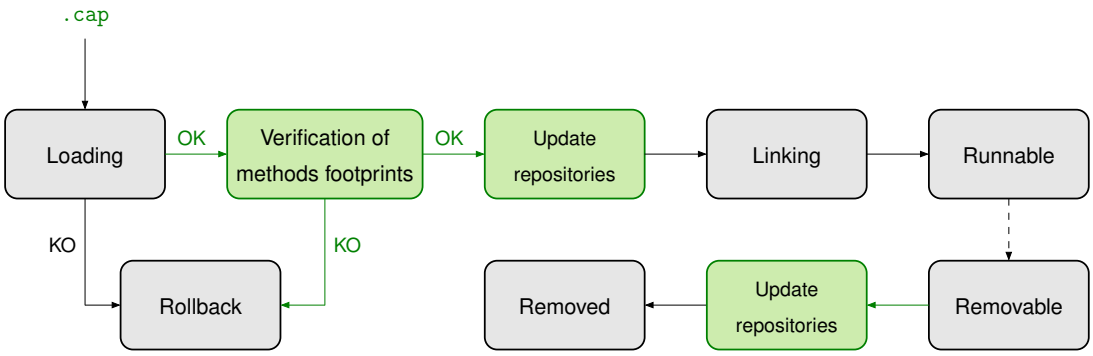

Figure 4.3: Schema of the on-device process triggered at loading-time of a new CAP file for the global policy model.

The structure of on-device repositories of methods footprints, given in Appendix D, is almost identical to the structure of the GPCC to simplify on-device treatments. It is actually simpler as no `external_*` structures are needed, and intermediate footprints should not be stored on-device after successful verification and installation.

4.1.3 Evaluation on the POPS Case Study

In this section, we estimate the size of the GPCC to be embedded in the POPS scenario packages as well as the size of repositories managed on-device at each step of the deployment scenario.

The size of the GPCC and the size of repositories depend on the size of the `footprint_t` structure, which depends on the number of states of the automaton that defines the set of forbidden sequences (Section 5.3.1 in the deliverable D6.3). Let n be the number of states of this automaton. So, according to the definition of a binary encoded footprint (Section 5.6.2 of the deliverable D6.3), the size of `footprint_t` is $N = \lceil \frac{n(n-1)}{8} \rceil$ bytes.

The GPCC Instances

Since the size of a GPCC depends on several variables, we proceed per data structure to compute the size of the GPCC for each package, according to the details of their content given in Appendix C.

The GPCC of the `newepurse.cap` Package. This package does not import any package[2], so there is no external footprint in its GPCC. The size of the GPCC structure instance, excluding the `class_footprints` array, is thus $1 + 2 + 2 + 2 = 7$ bytes. This package contains 3

[2]Import and use of JavaCard and GlobalPlatform APIs are not taken into account as they are considered safe and no footprint is attached to their methods.

classes and 2 interfaces. The sum of all `internal_class_footprint` instances sizes, excluding the `method_footprints` array, is thus $5 * (2 + 1 + 2) = 25$ bytes. This package contains 20 methods: 17 have bytecode instructions (*i.e.*are implemented), and 14 are visible (*i.e.*have a token set). The sum of all `internal_method_footprint` instances sizes, excluding the `intermediate_footprints` array, is thus $20+14+2*17+20N+20*2 = 108+20N$ bytes. In this package, the total count of instructions targeted by a jump in the 17 implemented methods is 157, so 157 intermediate footprints appear in the GPCC. The sum of all `intermediate_footprint` instances sizes is thus $157 * (2 + N) = 314 + 157N$ bytes.

The total size of the GPCC is the sum of all instantiated structures: $(314 + 157N) + (108 + 20N) + 25 + 7 = 454 + 177N$ bytes. The following table gives some instances of the GPCC size in bytes for the `newepurse.cap` package when n (and thus N) varies:

n	<4	4	5	6	7	8	9	10
N	1	2	3	4	6	7	9	12
Size	631	808	985	1162	1516	1693	2047	2578

The GPCC of the `newjticket.cap` Package. This package imports only the `newepurse.cap` package, and uses only 1 method of this package. So there is only one external method's footprint in its GPCC, and there is exactly one instance of each of the `external_*` structures. Assuming the AID of the `newepurse.cap` package is made of 16 bytes (the maximal possible size), the size of the `external_package_footprint`, excluding the `class_footprints` array, is $(1+16)+2 = 19$ bytes. The size of the `external_class_footprint`, including the `method_footprints` array which size is $1 + N$, is $3 + (1 + N) = 4 + N$ bytes. So, the size of the GPCC, excluding the `class_footprints` array, is $1+2+2+2+(19+4+N) = 30+N$ bytes. This package contains only 1 class. The size of the corresponding `internal_class_footprint` structure, excluding the `method_footprints` array, is thus $2 + 1 + 2 = 5$ bytes. This package contains 7 methods: 7 are implemented, and 3 are visible. The sum of all `internal_method_footprint` instances sizes, excluding the `intermediate_footprints` array, is thus $7 + 3 + 2 * 7 + 7N + 7 * 2 = 38 + 7N$ bytes. In this package, the total count of instructions targeted by a jump in the 7 implemented methods of the package is 53, so 53 intermediate footprints appear in the GPCC. The sum of all `intermediate_footprint` instances sizes is thus $53 * (2 + N) = 106 + 53N$ bytes.

The total size of the GPCC is the sum of all instantiated structures: $(106 + 53N) + (38 + 7N) + 5 + (30 + N) = 179 + 61N$ bytes. The following table gives some instances of the GPCC size in bytes for the `newjticket.cap` package when n (and thus N) varies:

n	<4	4	5	6	7	8	9	10
N	1	2	3	4	6	7	9	12
Size	240	301	362	423	545	606	728	911

The GPCC of the `neweidapplet.cap` Package. This package imports only the `newepurse.cap` package, and uses only 1 method of this package. So there is only one external method's footprint in its GPCC, and there is exactly one instance of each of the `external_*` structures. Assuming the AID of the `newepurse.cap` package is made of 16 bytes (this is the maximal possible size for an AID), the size of the `external_package_footprint`, excluding the `class_footprints` array, is $(1 + 16) + 2 = 19$ bytes. The size of the `external_class_footprint`, including the `method_footprints` array which size is $1 + N$, is $3 + (1 + N) = 4 + N$ bytes. So, the size of the GPCC, excluding the `class_footprints` array, is $1 + 2 + 2 + 2 + (19 + 4 + N) = 30 + N$ bytes. This package contains 6 classes and 1 interface. The sum of all `internal_class_footprint` instances sizes, excluding the `method_footprints` array, is thus $7 * (2 + 1 + 2) = 35$ bytes. This package contains 82 methods: 81 are implemented, and 38 are visible. The sum of all

`internal_method_footprint` instances sizes, excluding the `intermediate_footprints` array, is thus $82 + 38 + 2 * 81 + 82N + 82 * 2 = 446 + 82N$ bytes. In this package, the total count of instructions targeted by a jump in the 81 implemented methods of the package is 733, so 733 intermediate footprints appear in the GPCC. The sum of all `intermediate_footprint` instances sizes is thus $733 * (2 + N) = 1466 + 733N$ bytes.

The total size of the GPCC is the sum of all instantiated structures: $(1466 + 733N) + (446 + 82N) + 35 + (30 + N) = 1977 + 816N$ bytes. The following table gives some instances of the GPCC size in bytes for the `neweidapplet.cap` package when n (and thus N) varies:

n	<4	4	5	6	7	8	9	10
N	1	2	3	4	6	7	9	12
Size	2793	3609	4425	5241	6873	7689	9321	11769

The GPCC of the `newmypackage.cap` Package. This package imports the `newepurse.cap` package and the `neweidapplet.cap` package. Two methods defined in the same interface of the `newepurse.cap` package are invoked, while only one method in one interface of the `neweidapplet.cap` package is invoked. The GPCC of the `newmypackage.cap` package thus contains 2 instances of the `external_package_footprint` structure, plus 2 instances of the `external_class_footprint` structure, and three instances of the `external_method_footprint` structure. Assuming the two AIDs of the imported packages are each made of 16 bytes (this is the maximal possible size for an AID), the total size of the two `external_package_footprint` instances, excluding their `class_footprints` array, is $2 * ((1 + 16) + 2) = 38$ bytes. The total size of the two `external_class_footprint` instances, excluding their `method_footprints` array, is $2 * (1 + 2) = 6$ bytes. The total size of the three `external_method_footprint` instances is $3 * (1 + N) = 3 + 3N$ bytes. So, the size of the GPCC, excluding the `class_footprints` array, is $1+2+2+2+38+6+(3+3N) = 54+3N$ bytes. This package contains 2 classes and 1 interface. The sum of all `internal_class_footprint` instances sizes, excluding the `method_footprints` array, is thus $3 * (2 + 1 + 2) = 15$ bytes. This package contains 17 methods: 16 are implemented, and 6 are visible. The sum of all `internal_method_footprint` instances sizes, excluding the `intermediate_footprints` array, is thus $17 + 6 + 2 * 16 + 17N + 17 * 2 = 89 + 17N$ bytes. In this package, the total count of instructions targeted by a jump in the 16 implemented methods of the package is 187, so 187 intermediate footprints appear in the GPCC. The sum of all `intermediate_footprint` instances sizes is thus $187 * (2 + N) = 374 + 187N$ bytes.

The total size of the GPCC is the sum of all instantiated structures: $(374 + 187N) + (89 + 17N) + 15 + (54 + 3N) = 532 + 207N$ bytes. The following table gives some instances of the GPCC size in bytes for the `neweidapplet.cap` package when n (and thus N) varies:

n	<4	4	5	6	7	8	9	10
N	1	2	3	4	6	7	9	12
Size	739	946	1153	1360	1774	1981	2395	3016

On-Device Repositories

The size of on-device repositories is given according to the deployment scenario, after the installation of the each new package.

After the Installation of the `newepurse.cap` Package. There exists a single instance of the `ondevice_footprint_repositories` structure which has a size of 1 byte, excluding its `package_footprint` array. There exists only one instance of `ondevice_package_footprint` corresponding to the `newepurse.cap` package, and its size, excluding the `class_footprints`

array, is 3 bytes. The `newepurse.cap` package contains 5 classes/interfaces, so the sum of the five `ondevice_class_footprint` instances sizes, excluding the `intermediate_footprints` array, is thus $5 * (2 + 1 + 2) = 35$ bytes. The `newepurse.cap` package contains 20 methods: 17 implemented and 14 visible. So the sum of all corresponding `ondevice_method_footprint` instances sizes is $20 + 14 + 2 * 17 + 20N = 68 + 20N$ bytes. The total size of the single instance of `ondevice_footprint_repositories` structure after the installation of the `newepurse.cap` package is the sum of all the aforementioned structures: $1 + 3 + 35 + (68 + 20N) = 107 + 20N$ bytes. The following table gives some instances of the size in bytes of on-device repositories after only the `newepurse.cap` package is installed when n (and thus N) varies:

n	<4	4	5	6	7	8	9	10
N	1	2	3	4	6	7	9	12
Size	127	147	167	187	227	247	287	347

After the Installation of the `newepurse.cap` and the `newjticket.cap` Packages. The single instance of `ondevice_footprint_repositories` structure, excluding the `package_footprint` array, has a size of 1 byte. There are two instances of the `ondevice_package_footprint` structure, each instance corresponding to an installed package. The sum of the sizes of these two instances, excluding their `class_footprints` arrays, is $2 * 3 = 6$ bytes. There are 6 classes/interfaces installed (5 for `newepurse.cap`, 1 for `newjticket.cap`), so the sum of the four `ondevice_class_footprint` instances sizes, excluding their `methods_footprints` arrays, is thus $6 * (2 + 1 + 2) = 30$ bytes. There are 27 methods installed: 24 implemented and 17 visible. So the sum of all corresponding `ondevice_method_footprint` instances sizes is $27 + 17 + 2 * 24 + 27N = 92 + 27N$ bytes. The total size of the single instance of `ondevice_footprint_repositories` structure after the installation of these packages is the sum of all the aforementioned structures: $1 + 6 + 30 + (92 + 27N) = 129 + 27N$ bytes. The following table gives some instances of the size in bytes of on-device repositories after installation of the `newepurse.cap` and the `newjticket.cap` packages when n (and thus N) varies:

n	<4	4	5	6	7	8	9	10
N	1	2	3	4	6	7	9	12
Size	156	183	210	237	291	318	372	453

After the Installations of the First Three Packages. There exists a single instance of the `ondevice_footprint_repositories` structure which has a size of 1 byte, if we exclude its `package_footprint` array. There are three instances of the `ondevice_package_footprint` structure, each instance corresponding to an installed package. The sum of the sizes of these three instances, excluding their `class_footprints` arrays, is $3 * 3 = 9$ bytes. There are 13 classes/interfaces installed (5 for `newepurse.cap`, 1 for `newjticket.cap`, 7 for `neweidapplet.cap`), so the sum of the four `ondevice_class_footprint` instances sizes, excluding their `methods_footprints` arrays, is thus $13 * (2 + 1 + 2) = 65$ bytes. There are 109 methods installed: 105 implemented and 55 visible. So the sum of all corresponding `ondevice_method_footprint` instances sizes is $109 + 55 + 2 * 105 + 109N = 374 + 109N$ bytes. The total size of the single instance of `ondevice_footprint_repositories` structure after the installation of these packages is the sum of all the aforementioned structures: $1 + 9 + 65 + (374 + 109N) = 449 + 109N$ bytes. The following table gives some instances of the size in bytes of on-device repositories after installation of the `newepurse.cap`, the `newjticket.cap` and the `neweidapplet.cap` packages when n (and thus N) varies:

n	<4	4	5	6	7	8	9	10
N	1	2	3	4	6	7	9	12
Size	558	667	776	885	1103	1212	1430	1757

At the End of the Scenario. The single instance of the `ondevice_footprint_repositories` structure has a size of 1 byte, excluding its `package_footprint` array. There are four instances of the `ondevice_package_footprint` structure, each instance corresponding to an installed package. The sum of the sizes of these four instances, excluding their `class_footprints` arrays, is $4 * 3 = 12$ bytes. There are 16 classes/interfaces installed (5 for `newepurse.cap`, 1 for `newjticket.cap`, 7 for `neweidapplet.cap`, 3 for `newmypackage.cap`), so the sum of the four `ondevice_class_footprint` instances sizes, excluding their `methods_footprints` arrays, is thus $16 * (2 + 1 + 2) = 80$ bytes. There are 126 methods installed: 121 implemented and 61 visible. So the sum of all corresponding `ondevice_method_footprint` instances sizes is $126 + 61 + 2 * 121 + 126N = 555 + 126N$ bytes. The total size of the single instance of `ondevice_footprint_repositories` structure after the installation of these packages is the sum of all the aforementioned structures: $1 + 12 + 80 + (555 + 126N) = 648 + 126N$ bytes. The following table gives some instances of the size in bytes of on-device repositories after installation of the `newepurse.cap`, the `newjticket.cap`, the `neweidapplet.cap` and the `newmypackage.cap` packages when n (and thus N) varies:

n	<4	4	5	6	7	8	9	10
N	1	2	3	4	6	7	9	12
Size	774	900	1026	1152	1404	1530	1782	2160

4.2 Toward Implementing the Non-Interference Model for Smart-Cards

In the deliverables D6.3 (Chapter 6) and D6.4 (Chapter 5), we described the *non-interference* model. The purpose of this section is to give some details on its practicable implementation for JavaCard smart-cards, especially regarding content and size of the required meta-data to be embedded in CAP files (Section 4.2.1) and those persistently stored on-device (Section 4.2.2). Evaluation of memory consumption on the POPS scenario is described in Section 4.2.3.

4.2.1 Off-Device Computation and Embedding of Methods Signatures in CAP Files

Figure 4.4 gives an overview of how methods signatures are computed by the `STAN` tool[3] and how they can be embedded in their related CAP file thanks to a *Custom Component*. The process is quite different from the one described in Section 4.1.1 for the global policy model as it relies on an the `STAN` tool that computes methods signatures from `.class` files rather than a `.cap` file. However, using the methods signatures embedded in `.class` files by the `STAN` tool, an off-device tool for the non-interference model can produce a new CAP file with the same application content plus a *Custom Component* corresponding to methods signatures and intermediate signatures needed for on-device checking of complete signatures.

[3] http://stan-project.gforge.inria.fr/index.php/Main/HomePage

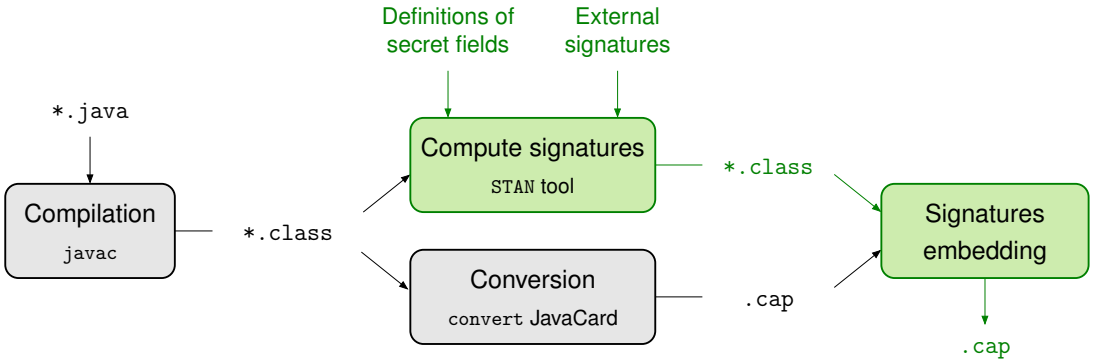

Figure 4.4: Schema of the off-device process for embedding methods signatures.

A Custom Component for Embedded Methods Signatures

The *Non-Interference Custom Component* (NICP) to be embedded in CAP files is a *Custom Component* of the CAP file format (*c.f.*JCVM 2.x specifications) with the data structures displayed in Figure 4.5. The fields of those structures have the following meaning.

The `non_interference_component` structure describes a NICP:

- `tag` contains the tag value (between 128 and 255 inclusive, as described in the ISO 7816-5) which permits to identify the NICP;
- `size` indicated the number of bytes in the `non_interference_component` structure, excluding the `tag` and `size` items. The value of the `size` field must be greater than 0;
- `external_signatures_size` represents the size in bytes of the `external_signatures` field;
- `external_signatures` contains all the signatures of external methods (*i.e.*defined in other packages) directly referenced (*i.e.*invoked, overridden, implemented) in this package;
- `class_signatures_size` represents the size in bytes of the `class_signatures` field;
- `class_signatures` contains a `internal_class_signature` entry for each class and each interface defined in this package.

The `external_package_signature` structure describes the methods signatures coming from an imported package and explicitly referenced (*i.e.*invoked, overridden, implemented) in this package:

- `package_aid` contains the package AID that permits to identify the package from which external signatures are coming from;
- `class_signatures_size` contains the size in bytes of the `class_signatures` field;
- `class_signatures` contains an `external_class_signature` entry for each class and each interface of this package containing a method referenced in the current package.

The `external_class_signature` structure describes the methods signatures of a class or an interface:

- `class_token` represents the class token of the current class (or interface); its value cannot be `0xFF` as it must correspond to a class or to an interface visible outside of the package in which it is defined;

```
non_interference_component {
    u1 tag;
    u2 size;
    u2 external_signatures_size;
    external_package_signature external_signatures[];
    u2 class_signatures_size;
    internal_class_signature class_signatures[];
}

external_package_signature {
    struct {
        u1 length;
        u1 value[length];
    } package_aid;
    u2 class_signatures_size;
    external_class_signature class_signatures[];
}

external_class_signature {
    u1 class_token;
    u2 method_signatures_size;
    external_method_signature method_signatures[];
}

external_method_signature {
    u1 method_token;
    signature_t complete_signature;
}

internal_class_signature {
    u2 classref;
    u1 class_token;
    u2 method_signatures_size;
    internal_method_signature method_signatures[];
}

internal_method_signature {
    u1 bitfield;
    u1 method_token; /* present according to bitfield */
    u2 method_offset; /* present according to bitfield */
    signature_t complete_signature;
    u2 intermediate_signatures_size;
    intermediate_signature intermediate_signatures[];
}

intermediate_signature {
    u2 pc;
    signature_t signature;
}
```

Figure 4.5: Data structures of the *Non-Interference Custom component* (NICP), a custom component of the CAP file format for the implementation of the non-interference model on JavaCard smart cards.

- `method_signatures_size` contains the size in bytes of the `method_signatures` field;
- `method_signatures` maps the signature to each method of the current class (or interface) that is referenced (invoked, overridden, implemented) in this package.

The `external_method_signature` structure describes the signature of an external method referenced in this package:

- `method_token` represents the static/virtual/interface method token of this method;
- `complete_signature` contains the expected signature of this method (Section 6.1.3 of the deliverable D6.3)[4].

The `internal_class_signature` structure describes the methods signatures of a class or an interface defined in this package:

- `classref` contains the location (*i.e.*the offset) in the Class Component (*c.f.*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_signatures_size` represents the number of entries in the `method_signatures` field;
- `method_signatures` maps to each method of the current class (or interface) its signature.

The `internal_method_signature` structure describes the signature of a method defined in this package:

- `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/virtual/interface method token of this method; this item is present iff the method is visible according to `bitfield`;
- `method_offset` represents a byte offset into the info item of the Method Component (*c.f.*JCVM specifications) if the method is implemented (*i.e.*not an abstract method or a method definition in an interface) according to `bitfield`;
- `complete_signature` contains binary encoded signature of the current method;
- `intermediate_signatures_size` contains the size in bytes of the `intermediate_signatures` field;
- `intermediate_signatures` contains the intermediate signatures of the current method used for on-device verification of the complete signature of the method.

[4]Signatures of external methods must be the ones used to compute signatures for this package.

The `intermediate_signature` structure describes an intermediate signature of a method, that is an incomplete signature of a method attached to an instruction of the method targeted by a jump (instruction following an `invoke`, a conditional block, ...):

- `pc` the offset of the instruction in the bytecode of the method to which this intermediate signature is attached;
- `signature` the binary encoded intermediate signature.

The structure `signature_t` is a two dimensional byte array which contains a binary encoded signature (Section 6.1.3 of the deliverable D6.3). Its size is variable: it is at least 25 bytes, and it grows in a polynomial way with the number of parameters of the analyzed method, but it does not depend on the policy applied to class attributes (Section 6.1.2 of the deliverable D6.3).

4.2.2 On-Device Verification and Storage of Methods Signatures

The on-device enforcement of a non-interference policy requires to verify at loading-time that all methods signatures of each incoming package (application, library) are correct with respect to their bytecode and respect the secret flow policy between security domains. In order not to be bypassed, the verification process must occur before the package is concretely installed and used by other packages installed on the card, as depicted on Figure 4.6, in the exact same way as the global policy model (Section 4.1.2).

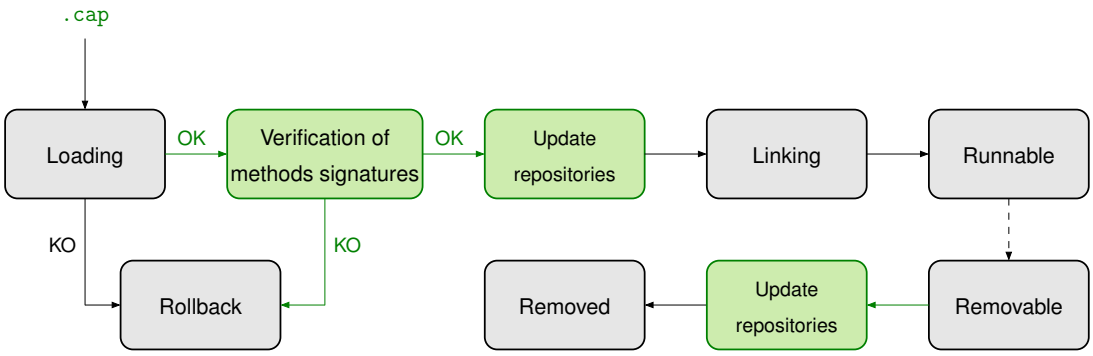

Figure 4.6: Schema of the on-device process triggered at loading-time of a new CAP file for the non-interference model.

The structure of on-device repositories of methods footprints, given in Appendix E, is almost identical to the structure of the GPCC to simplify on-device treatments. It is actually simpler as no `external_*` structures are needed, and intermediate signatures should not be stored on-device after successful verification and installation.

4.2.3 Evaluation on the POPS Case Study

In this section, we estimate the size of the NICP to be embedded in the POPS scenario packages as well as the size of repositories managed on-device at each step of the deployment scenario.

The size of the NICP depends on the size of the `signature_t` structure, which depends on the number of parameters of each method. Let $\mathcal{S}$ be a function that maps to a number of parameters of a method the size in bytes of its signature defined as:

$$\begin{aligned} \mathcal{S} : \mathbb{N} &\longrightarrow \mathbb{N} \\ p &\longmapsto 25 + p(10 + p) \end{aligned}$$

The NICP Instances

Since the size of a NICP depends on several variables, we proceed per data structure to compute the size of the NICP for each package, according to the details of their content given in Appendix C.

The NICP of the `newepurse.cap` Package. This package does not import any package[5], so there is no external signature in its NICP. The size of the NICP structure instance, excluding the `class_signatures` array, is thus $1 + 2 + 2 + 2 = 7$ bytes. This package contains 3 classes and 2 interfaces. The sum of all `internal_class_signature` instances sizes, excluding the `method_signatures` array, is thus $5 * (2 + 1 + 2) = 25$ bytes. This package contains 20 methods: 17 have bytecode instructions (*i.e.*are implemented), and 14 are visible (*i.e.*have a token set). The sum of all `internal_method_signature` instances sizes, excluding the `intermediate_signatures` array, is thus $20+14+2*17+\mathcal{S}(0)+17*\mathcal{S}(1)+\mathcal{S}(2)+\mathcal{S}(3)+20*2 = 966$ bytes. In this package, the total count of instructions targeted by a jump in the 17 implemented methods is 157, so 157 intermediate signatures appear in the NICP. The sum of all `intermediate_signature` instances sizes is thus $157*2+3*\mathcal{S}(0)+138*\mathcal{S}(1)+13*\mathcal{S}(2)+3*\mathcal{S}(3) = 6186$ bytes.

The total size of the NICP for the `newepurse.cap` package is the sum of all instantiated structures: $6186 + 966 + 25 + 7 = 7184$ bytes.

The NICP of the `newjticket.cap` Package. This package imports only the `newepurse.cap` package, and uses only 1 method of this package. So there is only one external method's signature needs in its NICP, and there is exactly one instance of each of the `external_*` structures. Assuming the AID of the `newepurse.cap` package is made of 16 bytes (this is the maximal possible size for an AID), the size of the `external_package_signature`, excluding the `class_signatures` array, is $(1 + 16) + 2 = 19$ bytes. The size of the `external_class_signature`, including the `method_signatures` array which size is $1 + \mathcal{S}(1)$, is $3 + (1 + \mathcal{S}(1)) = 40$ bytes. So, the size of the NICP, excluding the `class_signatures` array, is $1 + 2 + 2 + 2 + (19 + 40) = 66$ bytes. This package contains only 1 class. The size of the corresponding `internal_class_signature` structure, excluding the `method_signatures` array, is thus $2 + 1 + 2 = 5$ bytes. This package contains 7 methods: 7 are implemented, and 3 are visible. The sum of all `internal_method_signature` instances sizes, excluding the `intermediate_signatures` array, is thus $7 + 3 + 2 * 7 + \mathcal{S}(0) + 5 * \mathcal{S}(1) + \mathcal{S}(3) + 7 * 2 = 307$ bytes. In this package, the total count of instructions targeted by a jump in the 7 implemented methods of the package is 53, so 53 intermediate signatures appear in the NICP. The sum of all `intermediate_signature` instances sizes is thus $53 * 2 + 3 * \mathcal{S}(0) + 47 * \mathcal{S}(1) + 3 * \mathcal{S}(3) = 2065$ bytes.

The total size of the NICP of the `newjticket.cap` package is the sum of all instantiated structures: $2065 + 300 + 5 + 66 = 2436$ bytes.

The NICP of the `neweidapplet.cap` Package. This package imports only the `newepurse.cap` package, and uses only 1 method of this package. So there is only one external method's signature in its NICP, and there is exactly one instance of each of the `external_*` structures. Assuming the AID of the `newepurse.cap` package is made of 16 bytes (this is the maximal possible size for an AID), the size of the `external_package_signature`, excluding the `class_signatures` array, is $(1 + 16) + 2 = 19$ bytes. The size of the `external_class_signature`, including the `method_signatures` array which size is $1 + \mathcal{S}(1)$, is $3 + (1 + \mathcal{S}(1)) = 40$ bytes. So, the size

[5]Import and use of JavaCard and GlobalPlatform APIs are not taken into account as they are considered safe and no signature is attached to their methods.

of the NICP, excluding the `class_signatures` array, is $1 + 2 + 2 + 2 + (19 + 40) = 66$ bytes. This package contains 6 classes and 1 interface. The sum of all `internal_class_signature` instances sizes, excluding the `method_signatures` array, is thus $7 * (2 + 1 + 2) = 35$ bytes. This package contains 82 methods: 81 are implemented, and 38 are visible. The sum of all `internal_method_signature` instances sizes, excluding the `intermediate_signatures` array, is thus $82 + 38 + 2 * 81 + 25 * \mathcal{S}(0) + 20 * \mathcal{S}(1) + 32 * \mathcal{S}(2) + 4 * \mathcal{S}(3) + \mathcal{S}(4) + 82 * 2 = 3696$ bytes. In this package, the total count of instructions targeted by a jump in the 81 implemented methods of the package is 733, so 733 intermediate signatures appear in the NICP. The sum of all `intermediate_signature` instances sizes is thus $733 * 2 + 97 * \mathcal{S}(0) + 169 * \mathcal{S}(1) + 448 * \mathcal{S}(2) + 17 * \mathcal{S}(3) + 2 * \mathcal{S}(4) = 33177$ bytes.

The total size of the NICP of the `neweidapplet.cap` is the sum of all instantiated structures: $33177 + 3696 + 35 + 66 = 36974$ bytes.

The NICP of the `newmypackage.cap` Package. This package imports the `newepurse.cap` package and the `neweidapplet.cap` package. Two methods defined in the same interface of the `newepurse.cap` package are invoked, while only one method in one interface of the `neweidapplet.cap` package is invoked. The NICP of this package thus contains 2 instances of the `external_package_signature` structure, plus 2 instances of the `external_class_signature` structure, and 3 instances of the `external_method_signature` structure. Assuming the two AIDs of the imported packages are each made of 16 bytes (this is the maximal possible size for an AID), the total size of the two `external_package_signature` instances, excluding their `class_signatures` array, is $2 * ((1 + 16) + 2) = 38$ bytes. The total size of the two `external_class_signature` instances, excluding their `method_signatures` array, is $2*(1+2) = 6$ bytes. The total size of the three `external_method_signature` instances is $3 + 3 * \mathcal{S}(1) = 111$ bytes. So, the size of the NICP, excluding the `class_signatures` array, is $1 + 2 + 2 + 2 + 38 + 6 + 111 = 162$ bytes. This package contains 2 classes and 1 interface. The sum of all `internal_class_signature` instances sizes, excluding the `method_signatures` array, is thus $3 * (2 + 1 + 2) = 15$ bytes. This package contains 17 methods: 16 are implemented, and 6 are visible. The sum of all `internal_method_signature` instances sizes, excluding the `intermediate_signatures` array, is thus $17 + 6 + 2 * 16 + 3 * \mathcal{S}(0) + 10 * \mathcal{S}(1) + \mathcal{S}(2) + 2 * \mathcal{S}(3) + \mathcal{S}(5) + 17 * 2 = 801$ bytes. In this package, the total count of instructions targeted by a jump in the 16 implemented methods of the package is 187, so 187 intermediate signatures appear in the NICP. The sum of all `intermediate_signature` instances sizes is thus $187 * 2 + 16 * \mathcal{S}(0) + 141 * \mathcal{S}(1) + 4 * \mathcal{S}(2) + 19 * \mathcal{S}(3) + 7 * \mathcal{S}(5) = 7962$ bytes.

The total size of the NICP of the `newmypackage.cap` is the sum of all instantiated structures: $7962 + 801 + 15 + 162 = 8940$ bytes.

On-Device Repositories

The size of on-device repositories is given according to the deployment scenario, after the installation of the each new package.

After the Installation of the `newepurse.cap` Package. There exists a single instance of the `ondevice_signature_repositories` structure which has a size of 1 byte, excluding its `package_signature` array. There exists only one instance of `ondevice_package_signature` corresponding to the `newepurse.cap` package, and its size, excluding the `class_signatures` array, is 3 bytes. The `newepurse.cap` package contains 5 classes/interfaces, so the sum of the five `ondevice_class_signature` instances sizes, excluding the `intermediate_signatures` array, is thus $5 * (2 + 1 + 2) = 35$ bytes. The `newepurse.cap` package contains 20 methods:

17 implemented and 14 visible. So the sum of all corresponding `ondevice_method_signature` instances sizes is $20 + 14 + 2 * 17 + \mathcal{S}(0) + 17 * \mathcal{S}(1) + \mathcal{S}(2) + \mathcal{S}(3) + 20 * 2 = 966$ bytes. The total size of the single instance of `ondevice_signature_repositories` structure after the installation of the `newepurse.cap` package is the sum of all the aforementioned structures, including the `dataflow`: $N + 1 + 3 + 35 + 966 = 1005 + N$ bytes. The following table gives some instances of the size in bytes of on-device repositories after only the `newepurse.cap` package is installed when n (and thus N) varies:

n	<4	4	5	6	7
N	1	2	3	4	6
Size	1006	1007	1008	1009	1011

After the Installation of the `newepurse.cap` and the `newjticket.cap` Packages. The single instance of `ondevice_signature_repositories` structure has a size of 1 byte, excluding the `package_signature` array. There are two instances of the `ondevice_package_signature` structure, each instance corresponding to an installed package. The sum of the sizes of these two instances, excluding their `class_signatures` arrays, is $2 * 3 = 6$ bytes. There are 6 classes/interfaces installed (5 for `newepurse.cap`, 1 for `newjticket.cap`), so the sum of the four `ondevice_class_signature` instances sizes, excluding their `methods_signatures` arrays, is thus $6 * (2 + 1 + 2) = 30$ bytes. There are 27 methods installed: 24 implemented and 17 visible. So the sum of all corresponding `ondevice_method_signature` instances sizes is $27 + 17 + 2 * 24 + 2 * \mathcal{S}(0) + 22 * \mathcal{S}(1) + \mathcal{S}(2) + 2 * \mathcal{S}(3) + 27 * 2 = 1165$ bytes. The total size of the single instance of `ondevice_signature_repositories` structure after the installation of these packages is the sum of all the aforementioned structures, including the `dataflow`: $N + 1 + 6 + 30 + 1165 = 1202 + N$ bytes. The following table gives some instances of the size in bytes of on-device repositories after installation of the `newepurse.cap` and the `newjticket.cap` packages when n (and thus N) varies:

n	<4	4	5	6	7
N	1	2	3	4	6
Size	1203	1204	1205	1206	1208

After the Installations of the First Three Packages. There exists a single instance of the `ondevice_signature_repositories` structure which has a size of 1 byte, if we exclude its `package_signature` array. There are three instances of the `ondevice_package_signature` structure, each instance corresponding to an installed package. The sum of the sizes of these three instances, excluding their `class_signatures` arrays, is $3 * 3 = 9$ bytes. There are 13 classes/interfaces installed (5 for `newepurse.cap`, 1 for `newjticket.cap`, 7 for `neweidapplet.cap`), so the sum of the four `ondevice_class_signature` instances sizes, excluding their `methods_signatures` arrays, is thus $13 * (2 + 1 + 2) = 65$ bytes. There are 109 methods installed: 105 implemented and 55 visible. So the sum of all corresponding `ondevice_method_signature` instances sizes is $109 + 55 + 2 * 105 + 27 * \mathcal{S}(0) + 42 * \mathcal{S}(1) + 33 * \mathcal{S}(2) + 6 * \mathcal{S}(3) + \mathcal{S}(4) + 109 * 2 = 4861$ bytes. The total size of the single instance of `ondevice_signature_repositories` structure after the installation of these packages is the sum of all the aforementioned structures, including the `dataflow`: $N + 1 + 9 + 65 + 4861 = 4936 + N$ bytes. The following table gives some instances of the size in bytes of on-device repositories after installation of the `newepurse.cap`, the `newjticket.cap` and the `neweidapplet.cap` packages when n (and thus N) varies:

n	<4	4	5	6	7
N	1	2	3	4	6
Size	4937	4938	4939	4940	4942

At the End of the Scenario. The single instance of the `ondevice_signature_repositories` structure has a size of 1 byte, excluding its `package_signature` array. There are four instances of the `ondevice_package_signature` structure, each instance corresponding to an installed package. The sum of the sizes of these four instances, excluding their `class_signatures` arrays, is $4 * 3 = 12$ bytes. There are 16 classes/interfaces installed (5 for `newepurse.cap`, 1 for `newjticket.cap`, 7 for `neweidapplet.cap`, 3 for `newmypackage.cap`), so the sum of the four `ondevice_class_signature` instances sizes, excluding their `methods_signatures` arrays, is thus $16 * (2 + 1 + 2) = 80$ bytes. There are 126 methods installed: 121 implemented and 61 visible. So the sum of all corresponding `ondevice_method_signature` instances sizes is $126 + 61 + 2 * 121 + 30 * \mathcal{S}(0) + 52 * \mathcal{S}(1) + 34 * \mathcal{S}(2) + 8 * \mathcal{S}(3) + \mathcal{S}(4) + \mathcal{S}(5) + 126 * 2 = 5662$ bytes. The total size of the single instance of `ondevice_signature_repositories` structure after the installation of these packages is the sum of all the aforementioned structures, including the `dataflow`: $N + 1 + 12 + 80 + 5662 = 5755 + N$ bytes. The following table gives some instances of the size in bytes of on-device repositories after installation of the `newepurse.cap`, the `newjticket.cap`, the `neweidapplet.cap` and the `newmypackage.cap` packages when n (and thus N) varies:

n	<4	4	5	6	7
N	1	2	3	4	6
Size	5756	5757	5758	5759	5761

4.3 Security-by-Contract for HOMES

Work Package 6 (On-device verification part of it) of the Secure Change project focuses on the verification methodologies which can be used on resource-constrained Java-based devices to ensure that entities (for example, applications) are interacting in compliance with some pre-defined security policies.

The HOMES case study is a case study of the Secure Change project, that was chosen as a secondary case study for WP6 (due to unavailability of code the ATM case study is not applicable for WP6). In the current deliverable we propose the Security-by-Contract methodology applied for the HOMES case study.

The HOMES case study describes the OSGi platform as a part of a smart home. The OSGi platform is therefore the focus of the current study. In a nutshell, the SxC methodology enables each bundle coming onto the OSGi platform with a contract embedded into its manifest file. The contract contains details about bundle's functional requirements, and also it lists permissions for access to the services and packages of the current bundle. The PolicyChecker component embedded on the platform checks during installation that the requirements of the bundle are satisfied. Thus the SxC enables dynamic functionality and security enforcement in a changing environment when bundles from different providers are installed or removed while various provided services can be launched or stopped.

4.3.1 The HOMES Case Study

The HOMES case study of the Secure Change project describes a smart home that explores the OSGi technology (more information is available in D1.1.1 and D1.2 [2, 3]). OSGi is, in essence, a specification of a Java-based dynamic module system that is developed and maintained by the OSGi Alliance [43]. The OSGi Service Platform encompasses functions to change the provided functionality dynamically on the device on a variety of networks, without requiring restarts. To minimize the coupling, as well as make these couplings managed, the OSGi technology provides

a service-oriented architecture that enables the modules to dynamically discover each other for collaboration.

The Change Requirement and the Security Property

The chosen change requirement from D1.1.1 and D1.2 is **Bundle Life Cycle Operations**. The chosen security property is

Secure extensibility: The home gateway can be extended at run time with additional general software, coming from third parties in many cases. Such extensions should be verified to be secure in the sense that they do not introduce unauthorized information leaks or the possibility of denial of service.

4.3.2 Security by Contract for HOMES

We present the SxC methodology that is able to capture interactions of bundles on the OSGi platform and ensure these interactions are compliant with the specified security policies, thus enforcing the chosen security property Secure Extensibility. The current proposal demonstrates that the on-device verification techniques developed by WP6 can in fact be used in domains different from the Java Card technology (where we have spent the most of our effort). Moreover, various platforms can experience different benefits from the SxC methodology, depending on the computations resources available and trust assumptions made.

SxC's basic idea for OSGi is that each bundle will have a contract embedded into its manifest file. The contract contains details on the functional requirements and lists access permissions for the sibling services on the platform. During installation of bundles the contract is extracted and matched with the platform security policy aggregating the contracts of all installed bundles. Thus after the check we can be sure (under reasonable assumptions) that the incoming bundle respects the security limitations posed by other bundles.

The SxC proposal for the OSGi frameworks is detailed in Appendix F. It contains the notation for bundles' policies and a general description how the SxC approach can improve the current OSGi framework security management.

5. Conclusions

The technologies developed in Secure Change Work Package 6 provide substantial additional security guarantees for evolving software systems. The off-device verification provides strong guarantees on memory safety and absence of data-races. In addition, optional programmer provided correctness criteria (up to full functional correctness) can be verified. The on-device verification techniques can check security properties of newly loaded components in open systems. A range of techniques has been developed that explore different trade-offs with respect to verification effort and security guarantees provided.

We have developed the theory behind our techniques, have provided prototype implementations for the most promising techniques, and have evaluated them in the Secure Change case studies POPS and HOMES.

The overall conclusion of these evaluations is positive: we report on successful applications of both off-device and on-device verification techniques. However, we have also identified shortcomings of the current prototypes, and future research and development is needed to address these.

In this concluding chapter, we summarize the work we did by recapitulating the results on the two WP6 case studies. We draw conclusions and outline ongoing future directions of research.

5.1 Verification in the POPS Case Study

The POPS case study was the primary case study for work package 6. All our technologies were developed with this application case study in mind, and all implemented technologies were evaluated on this case study.

We developed a sizable application scenario for the POPS case study consisting of four interacting applets (two relatively small, newly developed applets, and two relatively large, existing applets), and we have shown how both off-device verification as well as on-device verification applies to this scenario.

5.1.1 Off-Device Verification

All applets from the scenario (but most notable the two existing non-trivial Java Card applets) were annotated and verified for correctness with respect to certain common programming errors. In particular, the verification proved that the applet does not contain transaction errors, performs no out of bounds operations on buffers, and never dereferences null pointers.

The results of this experiment are encouraging: with an annotation overhead of about four lines of annotations per five lines of code we found a total of 13 bugs in the eID applet, and 25 locations where transactions were not properly used.

The case study has led to a number of useful insights and showed us some of the rougher edges of the VeriFast tool that need to be polished some more. Most of the issues were small

and were either bugs in the tool (for instance, Java parsing errors) or functionality that was easy to implement but had not been done yet due to time constraints.

Currently, only a subset of the Java Card API is supported by VeriFast. For example, we do not support multi-applet applications that communicate via the shareable interface mechanism yet. We intend to support these additional features and write specifications for all library functions in the Java Card API.

An important, missing feature that turned out to greatly reduce the annotation overhead (and hence reduce the cost of verification) is automatic inference of open and close statements and of lemma applications. For example, the eID applet contains 211 open and close statements. While VeriFast already infers some ghost statements, we believe one of the most important steps to improve the verification experience is extending this inference mechanism.

This case study has been invaluable for us to improve the tool. A number of bugs were fixed and small additions were made in order to support the verification of the applets. A longer term plan has also been established to further add improvements and optimizations to the tool. In particular, the automatic generation of `open` and `close` statements is becoming an important part of VeriFast, as well as language and technology-specific extensions to the tool.

Further development of the inference of annotations is the most important avenue for future work.

5.1.2 On-Device Verification

In this deliverable, we have shown that the two non-implemented models (*global policy* and *non-interference*) for on-device information protection can be implemented for the POPS scenario in the same way as the *transitive control flow* and the SxC models are implemented and described in the D6.5. The precise estimation of overhead memory consumptions both in CAP files of the scenario and on-device given in this deliverable are quite promising, even if, according to GTO partner, it still appears that memory consumption for the meta-data is too high for the targeted UICC smart cards of the POPS scenario. Practically, we already see different ways to improve memory consumption for this platform, such as indexing footprints/signatures of methods in order to remove footprints/signatures redundancy, or including footprints/signatures in existing components of the CAP file format to avoid a useless redundant index of packages, classes and methods in a global policy/non-interfence custom component. Integration of these models on concrete UICC smart cards requires a lot of human resources but it is a real engineering challenge.

On-device verification of non-interference policies in an open context requires twice much resources than the verification of global policies as it is for more complex. We think about two theoretical future works to investigate in order to meet the requirements of embedded systems, not only legacy smart cards: to propose a secure information flow model less refined than the non-interference for simpler on-device verification, and to propose a global policy model enriched with some information flow elements only on few relevant/critical data. Both approaches have their pros and cons but each could lead to different solutions according to integrator and vendor needs.

5.2 Verification in the HOMES Case Study

The HOMES case study was the secondary case study for work package 6. We investigated the applicability of our verification technologies to HOMES.

5.2.1 Development-Time Verification of Core Security Modules in HOMES

Sec. 3.2 describes the verification of a core security module, namely the PEP program provided by TID, with VeriFast. PEP's intended use is to install security policies on home gateway systems by configuring network interfaces. The aim of our verification effort is on proving the absence of bugs related to memory safety and race conditions.

The case study proceeds in two stages of feasibility studies revealing that extensive implementation work in VeriFast is required. After extending VeriFast, the final experiment on verifying the PEP implementation is ongoing. Since PEP is production code, a low expectancy to detect bugs was assumed. Yet, our verification work found a relatively high number of NULL-pointer errors. On the downside, we found that the effort required to verify the PEP implementation was higher than expected.

Nevertheless, our results show that VeriFast can be used to effectively find bugs in the domain of low-level network management software. VeriFast gained substantially with respect to its support of the C programming language and the VeriFast documentation can be extended to give guidance for the verification of this kind of software. A very interesting avenue for future work is to investigate in how far such guidance can make the verification of security-critical modules (such as this PEP module, or more generally kernel modules or device drivers) sufficiently cheap.

5.2.2 Security by Contract for HOMES

Sec. 4.3 and related Appendix F contain the proposal how the Security-by-Contract approach can be applied to the OSGi framework that constitutes the core of the HOMES case study. We have reviewed the current drawbacks of the OSGi platforms in an open setting when bundles from third-party providers can be deployed or removed from the platform. In this scenario the bundle providers have very limited abilities to express their requirements on interactions with their bundles. Our proposal is to place the bundle's contract containing these requirements in the bundle's manifest file and to compare these requirements with the status of the platform during the bundle deployment. We have developed the notation for bundle contracts compliant with the manifest file syntaxis, with the proposed notation the bundle providers can also express some functional requirements of their bundles, such as presence of a certain bundle, package or service on the platform.

We did not have enough resources to implement the SxC bundle that could perform the contract extraction and comparison with the current status of the platform (platform policy). We expect, however, that the overhead of the deployment time SxC verification is acceptable, based on our estimations and evaluation of the industrial partner (TID). For the future work we are intended to pursue the following directions. It is interesting to enhance the proposed definition of bundle interactions to capture more precise details of the interaction (service usage, method invocation, information flow, etc). Also the scope of the bundle behavior can be extended to include sensitive system API calls. All these extensions of the contracts will require to develop methods to compare the contracts with the bundle's bytecode (the ClaimChecker component).

Bibliography

[1] Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., HÃď hnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., and Schmitt, P. H. The KeY tool. *Software and Systems Modeling*, 4:32–54, 2005.

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

[3] Armenteros, A., Chetali, B., Felici, M., Massacci, F., Meduri, V., and Tedeschi, A. Selected change requirements and security properties. Public project deliverable, SecureChange EU Project, www.securechange.eu, 2010.

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

[5] Balfe, B. Will Java 8 bring in OSGi?, http://blog.balfes.net/2011/08/09/will-java-8-bring-in-osgi/.

[6] Barnett, M., Leino, K., and Schulte, W. The spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, vol. 3362 of *LNCS*, pp. 49–69, Heidelberg, 2005. Springer.

[7] Barnett, M., yuh Evan Chang, B., Deline, R., Jacobs, B., and Leino, K. R. M. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO '05*, vol. 4111 of *LNCS*, pp. 364–387, Heidelberg, 2006. Springer.

[8] Belimpasakis, P., Michael, M., and Moloney, S. The home as a content provider for mash-ups with external services. In *CCNC'2009*, pp. 798–802, Piscataway, USA, 2009. IEEE.

[9] Berdine, J., Calcagno, C., and O'Hearn, P. Symbolic execution with separation logic. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2005.

[10] Boyland, J. Checking interference with fractional permissions. In *Static Analysis: 10th International Symposium*, 2003.

[11] Brotherston, J., Bornat, R., and Calcagno, C. Cyclic proofs of program termination in separation logic. *SIGPLAN Not.*, 43:101–112, 2008.

[12] Calcagno, C., Distefano, D., O'Hearn, P., and Yang, H. Compositional shape analysis by means of bi-abduction. *SIGPLAN Not.*, 44(1):289–300, 2009.

[13] Calcagno, C., Parkinson, M., and Vafeiadis, V. Modular safety checking for fine-grained concurrency. In *SAS '07*, vol. 4634 of *LNCS*, pp. 233–248, Heidelberg, 2007. Springer.

[14] Cataño, N. and Huisman, M. Formal specification of Gemplus' electronic purse case study using ESC/Java. In *Formal Methods Europe*, vol. 2391 of *LNCS*, pp. 272–289, Heidelberg, 2002. Springer.

[15] Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., and Tobies, S. VCC: A practical system for verifying concurrent C. In *TPHOLs '09*, vol. 5674 of *LNCS*, pp. 23–42, Heidelberg, 2009. Springer.

[16] Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A., and Vardi, M. Y. Proving that programs eventually do something good. *SIGPLAN Not.*, 42:265–276, 2007.

[17] Distefano, D., O'Hearn, P., and Yang, H. A local shape analysis based on separation logic. In *TACAS 2006*, vol. 3920 of *LNCS*, pp. 287–302, Heidelberg, 2006. Springer.

[18] Distefano, D. and Parkinson J, M. J. jstar: towards practical verification for java. In *OOPSLA '08*, pp. 213–226, New York, 2008. ACM.

[19] Flanagan, C., Rustan, K., Leino, M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI)*, 2002.

[20] Fontaine, A., Hym, S., Simplot-Ryl, I., Gadyatskaya, O., Massacci, F., Bernet, J., Nguyen, Q.-H., Bouquet, F., Piessens, F., and Angeli, M. D6.4: Impact analysis of new security requirements. Public project deliverable, SecureChange EU Project, www.securechange.eu, 2011.

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

[22] Geoffray, N., Thomas, G., Muller, G., Parrend, P., Frénot, S., and Folliot, B. I-JVM: a Java Virtual Machine for Component Isolation in OSGi. In *DSN'2009*. IEEE, 2009.

[23] Gomes, B. E. G., Moreira, A. M., and Déharbe, D. Developing Java card applications with B. In *Brazilian Symposium on Formal Methods (SBMF)*, 2005.

[24] Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., and Sagiv, M. Local reasoning for storable locks and threads. In *APLAS '07*, vol. 4807 of *LNCS*, pp. 19–37, Heidelberg, 2007. Springer.

[25] Gu, T., Pung, H., and Zhang, D. Toward an OSGi-based infrastructure for context-aware applications. *IEEE Perv. Computing*, 3:66–74, 2004.

[26] Harel, D., Kozen, D., and Tiuryn, J. Dynamic logic. In *Handbook of Philosophical Logic*. Springer, Heidelberg, 1984.

[27] Huisman, M., Gurov, D., Sprenger, C., and Chugunov, G. Checking absence of illicit applet interactions: A case study. In *Formal Aspects of Software Engineering*, 2004.

[28] Innerhofer-Oberperfler, F., Löw, S., Breu, R., Michael Breu, M. H., Agreiter, B., Felderer, M., Kalb, P., Scandariato, R., and Solhaug, B. D2.2: A configuration management process for lifelong adaptable systems. Public project deliverable, SecureChange EU Project, www.securechange.eu, 2011.

[29] Jacobs, B. and Piessens, F. Expressive modular fine-grained concurrency specification. *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2011)*, 46(1):271–282, 2011.

[30] Jacobs, B., Smans, J., and Piessens, F. A quick tour of the VeriFast program verifier. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2010.

[31] Jacobs, B., Smans, J., and Piessens, F. The VeriFast program verifier: A tutorial, 2011. http://people.cs.kuleuven.be/~bart.jacobs/verifast/tutorial.pdf.

[32] Lee, C., Nordstedt, D., and Helal, S. Enabling smart spaces with OSGi. *IEEE Perv. Computing*, 2(3):89 – 94, 2003.

[33] Leino, K. and Rümmer, P. A polymorphic intermediate verification language: Design and logical encoding. In *TACAS '10*, vol. 6015 of *LNCS*, pp. 312–327, Heidelberg, 2010. Springer.

[34] Leino, K. R. and Müller, P. A basis for verifying multi-threaded programs. In *ESOP '09*, vol. 5502 of *LNCS*, pp. 378–393, Heidelberg, 2009. Springer.

[35] Leino, K. R. M. Dafny: An automatic program verifier for functional correctness. In *LPAR-16 '10*, vol. 6355 of *LNCS*, pp. 348–370, Heidelberg, 2010. Springer.

[36] Leino, K. R. M., Nelson, G., and Saxe, J. B. Esc/java user's manual. Technical report, Compaq Systems Research Center, 2000.

[37] Mostowski, W. Fully verified Java Card API reference implementation. In *International Verification Workshop (VERIFY)*, 2007.

[38] Mostowski, W. and Poll, E. Midlet navigation graphs in JML. *Computer Networks*, 36(4), 2001.

[39] Ngu, A., Carlson, M., Sheng, Q., and Paik, H. Semantic-based mashup of composite applications. *IEEE Tran. on Services Computing*, 99:2–15, 2010.

[40] O'Hearn, P., Reynolds, J., and Yang, H. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic (CSL)*, 2001.

[41] Oracle. Java card technology, 2011. http://www.oracle.com/technetwork/java/javacard/overview/.

[42] OSGi. OSGi service platform core specification. Technical Report Version 4.3, The OSGi Alliance, 2011.

[43] OSGi Alliance Web Site, T. http://www.osgi.org.

[44] Parkinson, M. and Bierman, G. Separation logic and abstraction. *SIGPLAN Not.*, 40:247–258, 2005.

[45] Patrick, P. System and method for an infrastructure that rnables provisioning of dynamic business applications. Patent US 2009/0249287 A1, 2009.

[46] Philippaerts, P., Vogels, F., Smans, J., Jacobs, B., and Piessens, F. The belgian electronic identity card: a verification case study. In *AVoCS 2011*, 2011. To appear.

[47] Phung, P. and Sands, D. Security policy enforcement in the OSGi framework using aspect-oriented programming. In *COMPSAC'2008*, pp. 1076–1082, 2008.

[48] Piessens, F., Jacobs, B., Smans, J., Massacci, P. P. F., and Angeli, M. D6.2: Tool support for the programming model. Public project deliverable, SecureChange EU Project, www.securechange.eu, 2011.

[49] Roşu, G., Ellison, C., and Schulte, W. Matching logic: an alternative to hoare/floyd logic. In *AMAST'10*, vol. 6486 of *LNCS*, pp. 142–162, Heidelberg, 2010. Springer.

[50] Smans, J., Jacobs, B., and Piessens, F. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP 2009*, vol. 5653 of *LNCS*, pp. 148–172, Heidelberg, 2009. Springer.

[51] Vogels, F., Jacobs, B., Piessens, F., and Smans, J. Annotation inference for separation logic based verifiers. In *FMOODS 2011*, vol. 6722 of *LNCS*, pp. 319–333, Heidelberg, 2011. Springer.

[52] Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., and O'Hearn, P. Scalable shape analysis for systems code. In *CAV'08*, vol. 5123 of *LNCS*, pp. 385–398, Heidelberg, 2008. Springer.

A. The Belgian Electronic Identity Card: A Verification Case Study[1]

Authors: Pieter Philippaerts, Frédéric Vogels, Jan Smans[2], Bart Jacobs and Frank Piessens

Abstract: In the field of annotation-based source code level program verification for Java-like languages, separation-logic based verifiers offer a promising alternative to classic JML based verifiers such as ESC/Java2, the Mobius tool or Spec#. Researchers have demonstrated the advantages of separation logic based verification by showing that it is feasible to verify very challenging (though very small) sample code, such as design patterns, or highly concurrent code. However, there is little experience in using this new breed of verifiers on real code. In this paper we report on our experience of verifying several thousands of lines of Java Card code using VeriFast, one of the state-of-the-art separation logic based verifiers. We quantify annotation overhead, verification performance, and impact on code quality (number of bugs found). Finally, our experiments suggest a number of potential improvements to the VeriFast tool.

A.1 Introduction

Software verification is finally reaching a point where it is possible to verify relatively complex applications written in popular programming languages. Even though it is still often a significant effort to annotate applications in order to help them get verified automatically, the benefits outweigh the cost for a number of software markets. In particular, software with a very high cost of failure (for example, airplane controllers) or software for systems that are difficult to update after deployment (for example, smart cards) are perfect candidates for software verification.

VeriFast [30] is a verifier for single-threaded and multithreaded C and Java programs annotated with separation logic specifications. The approach enables programmers to ascertain the absence of invalid memory accesses, including null pointer dereferences and out-of-bounds array accesses, as well as compliance with programmer-specified method preconditions and postconditions.

This paper will assess the applicability of verification of Java Card applets using the VeriFast approach. Two non-trivial applets are annotated and an analysis of the verification effort and results is made. Section A.2 introduces the VeriFast tool and gives a short introduction to Java Card technology. Section A.3 describes the applets that were used in this case study and gives a short overview of some of the solutions we used to annotate certain features. Section A.4 evaluates the results of the case study and Section A.5 summarizes the future work. Finally, Section A.6 concludes the paper.

[1] To appear in: P. Philippaerts, F. Vogels, J. Smans, B. Jacobs, F. Piessens. The Belgian electronic identity card: a verification case study. AVoCS 2011, 2011.

[2] Jan Smans is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO)

A.2 Background

The results in this paper build on two technologies that will be presented in this section. Section A.2.1 presents a short overview of the verification technology used in this case study, and Section A.2.2 introduces the features of the Java Card platform relevant to the applets being verified.

A.2.1 VeriFast

VeriFast [30] is a verifier for Java and C programs annotated with separation logic [40] specifications. The tool modularly checks via symbolic execution [9] that each method in the program satisfies its specification. If VeriFast deems a Java program to be correct, then that program does contain neither null dereferences, array indexing errors, API usage violations nor data races. Moreover, all user-specified assertions are guaranteed to hold.

At the heart of the separation logic lies the concept of permissions. In particular, a method can only access a field if it has permission to do so. For example, consider the class `Interval` shown below. `o.low |-> v` denotes (1) the permission to access (read and write) the field `low` of the `Interval` object `o` and (2) that the current value of that field is `v`.

Listing A.1: VeriFast annotations in Java code

```
1  /*@ predicate interval(Interval i, int l, int h) =
2    i.low |-> l &*& i.high |-> h &*& l <= h;
3  @*/
4
5  public class Interval {
6    int low, high;
7
8    void shift(int amount)
9      //@ requires interval(this, ?low, ?high);
10     //@ ensures interval(this, low + amount, high + amount);
11   {
12     //@ open interval(this, low, high);
13     this.low += amount;
14     this.high += amount;
15     //@ close interval(this, low + amount, high + amount);
16   }
17 }
```

To distinguish full (read and write) from read-only access, permissions are qualified with fractions [10] between 0 (exclusive) and 1 (inclusive), where `1` corresponds to full access and any other fraction represents read-only access. For example, `[f]o.low |-> v` denotes read-only access if `f` is less than `1` and full access if `f` equals `1`. We typically omit explicitly writing `[1]` for full permissions. Permissions can be split and merged as required during the proof. For example, two read-only permissions `[1/2]o.low |-> v` and `[1/2]o.low |-> v` can be combined into a single full permission `[1]o.low |-> v` and vice versa. In the context of Java Card, we rely on fractional permissions to check that fields are not assigned to outside of transactions.

To abstract over the set of permissions required by a method, permissions can be grouped and hidden via predicates. For example, consider the predicate `interval` shown above. This predicate groups the permissions to access `low` and `high`, and additionally states that the value of `low` must be less or equal to the value of `high`. Just like basic permissions, predicates can be split and merged as required during the proof.

Each method in the program has a corresponding method contract consisting of a pre- and

postcondition that respectively describe the permissions required and returned by the method. More specifically, the permissions described by the precondition conceptually transfer from the caller to the callee on entry to the method, and vice versa for the postcondition when the method returns. For example, consider the method `shift` in the class `Interval`. `shift`'s precondition states that the method may only be called if `this` points to a valid interval (where the meaning of "valid interval" is determined by the predicate `interval`). The precondition imposes no restriction on the interval's bounds, but binds the lower bound to the variable `low` (indicated by the question mark) and the upper bound to `high`. The postcondition ensures that `this` is still a valid interval, and its bounds have been shifted by `amount` with respect to the method pre-state. Note that our verification tool requires all annotations to be written inside special comments (`/*@ ... @*/`) which are ignored by the Java compiler but recognized by our verifier.

VeriFast does not automatically fold and unfold predicate definitions. Instead, folding and unfolding must be done explicitly by developers via ghost commands (unless the predicate is marked precise). For example, the open statement in the body of shift unfolds the definition of the predicate `interval`, and similarly the close statement folds the definition. Verification of the code snippet shown above fails if any of the ghost statements is removed.

In addition to static predicates (placed outside of a class), VeriFast also supports instance predicates (placed inside of a class). Just like instance methods, instance predicates are dynamically bound. That is, the variable `this` is an implicit argument to each instance predicate, and its dynamic type determines the exact meaning of the predicate. For example, the interface `Vehicle` shown below defines the instance predicate `valid`. The meaning of valid depends on the subclass at hand. For example, `o.valid()` denotes the permission to read the field `maxspeed` if the dynamic type of `o` is `Car`. In the context of Java Card, we use instance predicates to describe consistency conditions for applets (i.e. the invariant that must be preserved by each transaction).

Listing A.2: Instance predicates.

```
1  interface Vehicle {
2    //@ predicate valid();
3  }
4  class Car implements Vehicle {
5    int maxspeed;
6
7    //@ predicate valid() = [1/2]this.maxspeed |-> _;
8  }
```

The extended static checker for Java (Esc/Java) [19] is another program verifier that has been used to verify Java Card programs [38, 14]. However, Esc/Java is unsound [36, Appendix C.0]. This means that Esc/Java can fail to detect certain bugs. For example, the extended static checker reasons incorrectly about object invariants in the presence of reentrant calls. Unlike Esc/Java, the VeriFast methodology has been proven to be sound [29].

Gomes *et al.* [23] have investigated using the B method to generate correct Java Card implementations from abstract models via refinement. Contrary to the B method, VeriFast does not start from an abstract model, but instead reasons directly about the applet's source code. The advantage of our approach is that we can retroactively prove correctness of existing implementations.

VeriFast performs *modular* verification. This means that the verifier analyzes each method in isolation using only method contracts (not the callees' implementations) to reason about method calls. An advantage of modular reasoning is that verification scales (verification times remain low) and that deep properties can be specified and checked. A disadvantage however

is that the developer must write annotations at method boundaries. To avoid having to write annotations, Huisman *et al.* [27] have used model checking to find bugs in Java Card applications. Unlike VeriFast, Huisman *et al.* do not aim to prove the absence of all errors, but only of certain undesired applet interactions.

Mostowski [37] has written a specification for the Java Card API in dynamic logic. In addition, he has used this specification to verify a number of applets using the Key verifier. A recurring problem encountered during these case studies was bad prover performance. For example, Mostowski states that "*it is not uncommon for the prover to run over an hour to finish the proof of one method*". Contrary to [37], we use separation logic to specify the Java Card API. While separation logic has proven to be a powerful specification formalism for reasoning about complex (but small) examples such as design patterns and highly concurrent code, there is only limited experience in applying separation logic to larger, realistic programs. This paper reports on our experience in applying separation logic to verify realistic Java Card code. An explicit goal of VeriFast is to keep verification times low. For example, the time needed to verify full functional correctness of a single method is typically under one second.

A.2.2 Java Card

The Java Card platform [41] was initially launched by Sun in 1996 and aimed to simplify the development of smart card applications. Until then, smart card code was largely written in C, which is difficult to write in the first place, and also has distinct disadvantages in terms of security and reliability.

The Java Card platform allowed developers to write smart card applets in a subset of the Java language that targets a specifically optimized Java framework for smart cards. The older (and most popular) platform, now called Java Card Classic Edition, does not support floating point operations, strings, multi-threading, garbage collection, stack inspection, multidimensional arrays, reflection, etc. The newest Java Card 3.0 Connected Edition supports more features but is still lacking compared to the full Java language and framework.

Java Card is now the dominant platform for smart cards, with applications for GSM, 3G, finance, PKI, e-commerce and e-government. Due to the absence of serious competition and the improvements of the latest incarnation of the Java Card platform, it can be expected that this will remain the case in the near future.

Applets

The entry point of each Java Card applet is a class that extends the built-in, abstract class `javacard.framework.Applet`. This class defines a number of methods that are called by the Java Card runtime to interact with the applet. In particular, the class `Applet` defines an abstract method `process` that must be overridden by the subclass. The implementation of `process` forms the core of the applet. More specifically, `process` takes an apdu (i.e. a wrapper around a byte array) as input, processes it, and possibly returns an updated apdu to the runtime. Typically, the apdu contains both information on the action that should be performed by the applet and data associated with that action.

A subclass of `javacard.framework.Applet` is a valid applet only if it declares a static method called `install`. The goal of this method is to create a new applet instance and to register this instance with the runtime. The class `MyApplet` shows the prototypical structure of a Java Card applet.

Listing A.3: The prototypical structure of an applet.

```
1  class MyApplet extends Applet {
2    public static void install(byte[] arr, short offset, byte length) {
3      MyApplet applet = new MyApplet();
4      // initialize the applet
5      applet.register();
6    }
7
8    public void process(APDU apdu) {
9      // process the apdu
10   }
11 }
```

Transactions

Java Card applets use two types of memory to store data and intermediate results. Fields and objects are stored in persistent EEPROM memory, whereas the stack (and hence local variables) are stored in volatile RAM memory. In addition, the applet can also choose to allocate arrays in RAM memory, because this type of memory is faster and is harder for attackers to read. This complicates things because the smart card may lose power at any time during the computation, which results in the RAM memory being wiped, whereas the EEPROM memory retains the intermediate results.

To preserve consistency of the data stored in persistent memory, Java Card supports transactions. More specifically, the platform defines three methods to interact with the transaction mechanism: `beginTransaction`, `commitTransaction`, and `abortTransaction`. When `beginTransaction` is called, all changes to persistent memory are made conditionally. Only when a call to `commitTransaction` is executed, the changes to the persistent memory are committed atomically. If `abortTransaction` is called instead, or if the card suddenly loses power before calling `commitTransaction`, the persistent memory is restored to its original state (on card boot-up when power is restored). Note that the transaction mechanism does not impact values stored in RAM. Incorrect use of the transaction API, for example calling `beginTransaction` while a transaction is already in progress, results in an exception.

Java Card and VeriFast

VeriFast was originally developed for C and Java programs, but has been modified to also support Java Card applications. The Java language used for Java Card applications is a precise subset of the full Java language, thus adding Java Card support to VeriFast was easy.

Java Card does, however, use a very different class library optimized for smart cards. VeriFast needs to know for every function in the library the pre- and postconditions in order to reason about code. These specifications are placed in a separate file that defines all the classes and methods in the Java Card framework. The specifications are based on the descriptions of these methods in the official Java Card documentation. The actual implementation of these library functions is not checked.

Building the specification file is an incremental process. VeriFast only needs pre- and postconditions for the methods that are actually used by the applications you want to verify. Hence, only a subset of the full Java Card class library has been annotated in the specification file. It is critical that the specifications of library functions is correct; errors in their annotations could lead to errors in the verification process. Therefore, extreme diligence is used when adding new function definitions to the specification.

A.3 Case Study

Software verification is still a very time-consuming process. Existing or new source code must be annotated in order to express assumptions and invariants, and to let the verifier reason about the code. Minimizing these required annotations is an active field of research where a lot of work remains to be done. For current verification technologies the overhead of annotating code is far from negligible, so it is not (yet) economically profitable to try to annotate and verify every piece of code. Large, non-critical code bases are examples where the effort probably is not worth the hassle.

However, smart card applications have a number of properties that *do* make them ideal candidates for software verification. First of all, they are typically small, in the order of a few thousand lines of code. Secondly, they are critical, in the sense that they usually offer some kind of security service. And last but not least, it is extremely difficult to update the code once it has been deployed. If a serious bug is discovered in the code, it might be necessary to recall all the deployed smart cards and issue new ones, which could be a commercial disaster.

This paper reports on the verification of two Java Card applets: one large open source applet that implements a clone of the Belgian Electronic Identity Card, and another smaller commercial applet.

The remainder of this section focusses on the larger, open source applet. Unfortunately, due to contractual constraints we are not allowed to discuss the details of the commercial applet.

A.3.1 The Belgian Electronic Identity Card

The Belgian Electronic Identity Card (eID) was introduced in 2003 as a replacement for the existing non-electronic identity card. Its purpose is to enable e-government and e-business scenarios where strong authentication is necessary. The card has the size of a standard credit card and features an embedded chip. In addition to containing a machine readable version of the information printed on the card, the chip also contains the address of the owner and two RSA key pairs with the corresponding X509 certificates. One key pair is used for authentication, whereas the other key pair can be used to generate legally binding electronic signatures.

The card is implemented on top of the Java Card platform (Classic Edition) and implements the smart card commands as defined in the ISO7816 standard. Unfortunately, the actual code that runs on the eID cards is not publicly available. For our case study, we used an open source, cloned version of the eID applet that implements the same functionality as the real eID card[3]. It is aimed at developers who wish to interact with eID cards as an easy to use and customizable testing platform.

The eID implementation consists of one large class called `EidCard` and a few other small helper classes. The `EidCard` class inherits from the `Applet` class and encapsulates about 80% of the entire code base. It is a complex class of about 900 lines of code and no less than 38 fields.

A.3.2 Specification of Transaction Correctness

Java Card offers transactions to preserve consistency of the data stored in persistent memory. However, what does it mean for an applet to be consistent? In VeriFast, developers can explicitly write down what fields are part of the persistent state together with the desired consistency conditions. More specifically, the class `Applet` defines an instance predicate called `valid`. Each subclass must override this predicate. The implementation of the predicate given in the subclass defines the consistency conditions for the applet at hand. For example, consider the applet

[3]The code can be downloaded from http://code.google.com/p/eid-quick-key-toolset/

class `ExampleApplet` shown below. The predicate `valid` indicates that both the fields `arr` and `i`, and the array pointed to by `arr` are part of the persistent state (line 6). Moreover, the predicate imposes the consistency condition that `i` is a valid index in `arr` (line 7).

Listing A.4: The contract of the `process` method, using fractional permissions.

```
1  class ExampleApplet extends Applet {
2    short i;
3    short[] arr;
4    /*@
5    predicate valid() =
6      this.arr |-> ?arr &*& this.i |-> ?i &*&
7      array_slice(arr, 0, ?len, _) &*&
8      0 <= i &*& i < len;
9    @*/
10 }
```

While reading fields is possible at any time, updates to persistent memory should be made inside of a transaction. The permission system used by VeriFast is the key to enforcing this property. More specifically, at the start of the `process` method, no transaction is in progress. As shown in Listing A.5, the precondition of `process` contains $1/2$ of the `valid` predicate. This means that the method can read but not update fields included in `valid` (as the method only has one half of the permissions included in `valid`). The predicate `current_applet` is simply a token describing the currently active applet.

Listing A.5: The contract of the `process` method, using fractional permissions.

```
1  public void process(...)
2     //@ requires current_applet(this) &*& [1/2]valid() &*& ...;
3     //@ ensures current_applet(this) &*& [1/2]valid() &*& ...;
4  {
5     ...
6  }
```

To update the fields of the applet, the method should somehow gain additional permissions (namely the other half of the `valid` predicate). These additional permissions can be acquired by calling `beginTransaction`. In particular, the postcondition of `beginTransaction` shown in Figure A.6 gives $1/2$ of the `valid` predicate. The `process` method can then merge `[1/2]valid()` (gained from the precondition of `process`) and `[1/2]valid()` (gained from the postcondition of `beginTransaction`) into `[1]valid()`. The full permission to `valid` gives the applet the right to modify the applet's fields for the duration of the transaction. When calling `commitTransaction`, half of the permissions included in the `valid()` predicate return to the system again. Note that it is impossible to call `endTransaction` if the applet is in an invalid state (according to the conditions described by `valid`), as the precondition of `commitTransaction` requires the consistency conditions to hold.

A.3.3 Inheritance

The ISO7816 standard specifies a mechanism to access files that are stored on a smart card. Three types of files are defined:

1. **Master files** represent the root of the file system. Each smart card contains at most one master file.

Listing A.6: The declaration of the `beginTransaction` and `commitTransaction` methods

```
1  public static void beginTransaction();
2     //@ requires current_applet(?a) &*& ...;
3     //@ ensures current_applet(a) &*& [1/2]a.valid() &*& ...;
4
5  public static void commitTransaction();
6     //@ requires current_applet(?a) &*& a.valid() &*& ...;
7     //@ ensures current_applet(a) &*& [1/2]a.valid() &*& ...;
```

2. **Elementary files** contain actual data.
3. **Dedicated files** behave like directories. They can contain other dedicated or elementary files.

To represent this structure, the eID implementation uses helper classes that form a class hierarchy. The root of the hierarchy is the abstract `File` class. This class has two subclasses: `DedicatedFile` and `ElementaryFile`. And finally, the `MasterFile` class inherits from `DedicatedFile`.

When a class is defined in the source code, it can be annotated with a predicate that represents an instance of that class. These predicates can then be used elsewhere to represent a fully initialized instance of that class. Listing A.7 shows how a `File` predicate can be defined for the corresponding `File` class. The class consists of two fields, which are also represented in the predicate. The predicate can also contain other information about the class such as invariants.

Listing A.7: A first definition of the *File* class and predicate.

```
1  public abstract class File {
2     /*@ predicate File(short theFileID, boolean activeState) =
3         this.fileID |-> theFileID &*&
4         this.active |-> activeState; @*/
5
6     private short fileID;
7     protected boolean active;
8
9     ...
10 }
```

The `ElementaryFile` class redefines the `File` predicate as shown in lines 2-4 of Listing A.8. A `File` predicate that is associated with an `ElementaryFile` class is defined as an `ElementaryFile` predicate where three of the five parameters are undefined.

The definition of the `ElementaryFile` predicate (lines 5-13) consists of a link to the `File` predicate defined in Listing A.7 and some extra fields and information that are specific to elementary files.

When an object is cast from the `File` to the `ElementaryFile` class (or vice versa), the corresponding predicate on the symbolic heap must be changed as well. We 'annotated' this by adding the methods that are defined in Listing A.9 to the `ElementaryFile` class and calling these methods when required. Obviously, this solution is far from elegant because it requires adding calls to stub functions in the code of the applet. The most recent version of VeriFast supports annotating this behavior as lemma methods (i.e. methods defined inside an annotation), removing the requirement of modifying the applet's code.

One problem that occurs with the methods presented in Listing A.9 is that information is lost when an `ElementaryFile` is cast to a `File` and then back again to an `ElementaryFile`. This

Listing A.8: A first definition of the *ElementaryFile* class and predicate.

```
1  public final class ElementaryFile extends File {
2     /*@ predicate File(short theFileID, boolean activeState) =
3           ElementaryFile(theFileID, ?dedFile, ?dta,
4             activeState, ?sz); @*/
5     /*@ predicate ElementaryFile(short fileID,
6             DedicatedFile parentFile, byte[] data,
7             boolean activeState, short size) =
8           this.File(File.class)(fileID, activeState) &*&
9           this.parentFile |-> parentFile &*&
10          this.data |-> data &*& data != null &*&
11          this.size |-> size &*&
12          array_slice(data, 0, data.length, _) &*&
13          size >= 0 &*& size <= data.length; @*/
14
15    private DedicatedFile parentFile;
16    private byte[] data;
17    private short size;
18
19    ...
20 }
```

Listing A.9: Functions to cast predicates.

```
1  public void castFileToElementary()
2     //@ requires [?f]File(?fid, ?state);
3     //@ ensures [f]ElementaryFile(fid, _, _, state, _);
4  {
5     //@ open [f]File(fid, state);
6  }
7
8  public void castElementaryToFile()
9     //@ requires [?f]ElementaryFile(?fid, ?dedFile, ?dta, ?state, ?sz);
10    //@ ensures [f]File(fid, state);
11 {
12    //@ close [f]File(fid, state);
13 }
```

loss of information happens in the `castFileToElementary` method where three parameters are left undefined.

There are some instances in the eID applet where this loss of information is problematic. The solution was to extend the `File` and `ElementaryFile` predicates to contain an extra parameter that can store any information. The result can be seen in Listing A.10. Line 3 shows the definition of this extra parameter. In the case of the `File` class, no extra information is kept and the parameter is defined to be empty (denoted as 'unit' on line 5). Similarly, line 22 defines the parameter to be empty for the `ElementaryFile` predicate, because all state information that can be stored in the predicate is fully defined by the other parameters.

Line 14 shows the case where the predicate needs the extra parameter to store additional information about the object. In this case, the `info` parameter stores a quad-tuple of extra information that can be used to correctly initialize the embedded `ElementaryFile` predicate without losing information.

Listing A.10: A more complete definition of the *File* and *ElementaryFile* predicates that supports downcasting.

```
1  public abstract class File {
2     /*@ predicate File(short theFileID, boolean activeState,
3              any info) =
4           this.fileID |-> theFileID &*&
5           this.active |-> activeState &*& info == unit; @*/
6
7     ...
8  }
9
10 public final class ElementaryFile extends File {
11    /*@ predicate File(short theFileID, boolean activeState,
12             quad<DedicatedFile, byte[], short, any> info) =
13          ElementaryFile(theFileID, ?dedFile, ?dta, activeState,
14          ?sz, ?ifo) &*& info == quad(dedFile, dta, sz, ifo); @*/
15    /*@ predicate ElementaryFile(short fileID,
16             DedicatedFile parentFile, byte[] data, boolean activeState,
17             short size, any info) =
18          this.File(File.class)(fileID, activeState, _) &*&
19          this.parentFile |-> parentFile &*&
20          this.data |-> data &*& data != null &*& this.size |-> size
21          &*& array_slice(data, 0, data.length, _) &*&
22          size >= 0 &*& size <= data.length &*& info == unit; @*/
23
24    ...
25 }
```

A.4 Evaluation

The main goal of this case study was to see how practical it is to use VeriFast to annotate a Java Card applet that is more than a toy project. It gives us an idea of how much the annotation overhead is, where we can improve the tool, and whether we can actually find bugs in existing code using this approach.

A.4.1 Annotation Overhead

The more information the developer gives in the annotations about how the applet should behave, the more VeriFast can prove about it. It is up to the developer to choose whether he wants to use VeriFast as a tool to only detect certain kinds of errors (unexpected exceptions and incorrect use of the API), or whether he wants to prove full functional correctness of the applet. Both modi operandi are supported by the tool. For the Java Card applets, we used the annotations to prove that the applet does not contain transaction errors, performs no out of bounds operations on buffers, and never dereferences null pointers. While we have used VeriFast to verify full functional correctness of sequential and fine-grained concurrent data structures [29], specifying and verifying full functional correctness of JavaCard applets is future work.

The eID applet and helper classes consist of 1,004 lines of Java Card code. In order to verify the project, we added 802 lines of VeriFast annotations (or about four lines of annotations for every five lines of code). The majority of these annotations were `requires/ensures` pairs (88 pairs, one for each method) and `open` and `close` statements (99 and 112 instances respectively). Remarkably, only 8 predicates are defined throughout the entire code base, reflecting the design decision of the authors of the applet to write most of it as one huge class file.

The commercial applet consists of 251 lines of Java Card code, which we annotated with 205 lines of VeriFast annotations. There were 13 `requires/ensures` pairs, 25 `open` statements

and 29 `close` statements.

Another type of annotation overhead is the time it took to actually write the annotations. The verification of the eID applet was performed by a senior software engineer without prior experience with the VeriFast tool, but with regular opportunities to consult VeriFast expert users during the verification effort. We did not keep detailed effort logs, but a rough estimate of the effort that was required is 20 man-days. This includes time spent learning the VeriFast tool and the Java Card API specifications. The commercial applet was annotated by a VeriFast specialist and took about 5 man-days, excluding the time it took to add some new required features to the tool.

A.4.2 Bugs and Other Problems

Because the eID applet in our case study is aimed at developers, the authors did not spend a lot of time worrying about card tearing. This is demonstrated by the fact that they did not use the Java Card transaction system at all. Using VeriFast, we found 25 locations where a card tear could cause the persistent memory to enter an inconsistent state.

Three locations were found where a null pointer dereference could occur. An additional three class casting problems were found, where a variable holding a reference to the selected file (of type `File`) was cast to an `ElementaryFile` instance. These bugs could be triggered by sending messages with invalid file identifiers to the smart card. Seven potential out of bounds operations were also found in the code. These bugs could be triggered by sending illegal messages to the smart card.

We also found a number of bugs in the commercial applet, even though it had already been verified with another verification technology previously. We found an unsafe API call, a handful of unchecked assumptions about incoming apdus, and four locations where transactions were not used properly. Clearly, the tool used earlier was not sound or was not used in a sound way.

A.4.3 VeriFast Strengths

Compared to other program verifiers that target Java Card [37, 19], VeriFast has two advantages: speed and soundness. That is, VeriFast usually reports in only a couple of seconds (usually less) whether the applet is correct or whether it contains a potential bug. Secondly, if VeriFast deems a program to be correct, then that program is guaranteed to be free from null pointer and array index out of bounds exceptions, and API usage and assertion violations.

A feature that proved to be crucial in understanding failed verification attempts is VeriFast's symbolic debugger. As shown in Figure A.1, the symbolic debugger can be used to diagnose verification errors by inspecting the symbolic states encountered on the path to the error. For example, if the tool reports an array indexing error, one can look at the symbolic states to find out why the index is incorrect. This stands in stark contrast to most verification condition generation-based tools that simply report an error, but do not provide any help to understand the cause of the error.

A.5 Future Work

This case study has led to a number of useful insights and showed us some of the rougher edges of the tool that need to be polished some more. Most of the issues were small and were either bugs in the tool (for instance, Java parsing errors) or functionality that was easy to implement but hadn't been done yet due to time constraints.

An important, missing feature that would greatly reduce the annotation overhead (and hence reduce the cost of verification) is automatic inference of open and close statements and of

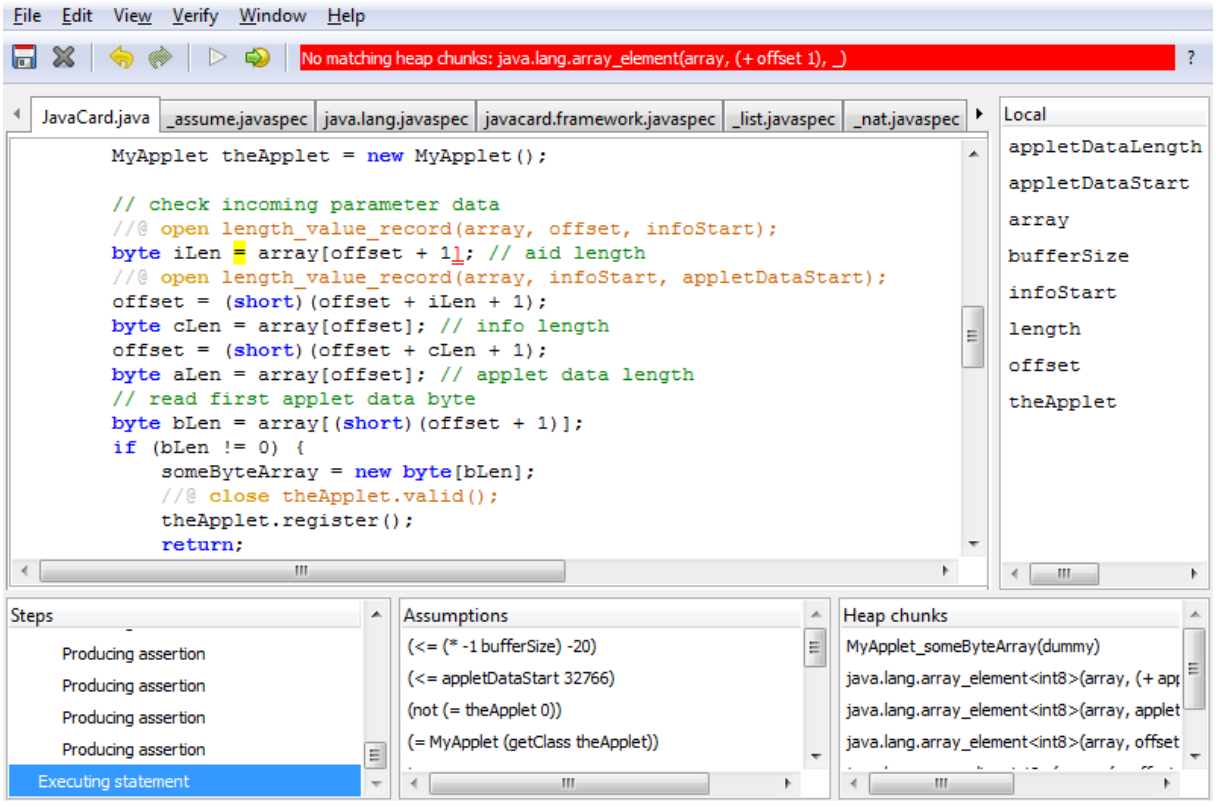

Figure A.1: The symbolic debugger of VeriFast

lemma applications. For example, the eID applet contains 211 open and close statements. While VeriFast already infers some ghost statements, we believe one of the most important steps to improve the verification experience is extending this inference mechanism.

Currently, only a subset of the Java Card API is supported by VeriFast. For example, we do not support multi-applet applications that communicate via the shareable interface mechanism yet. We intend to support these additional features and write specifications for all library functions in the Java Card API.

A.6 Conclusion

This paper reported on a case study for the VeriFast program verifier. Two non-trivial Java Card applets were annotated and verified for correctness with respect to certain common programming errors. In particular, the verification proved that the applet does not contain transaction errors, performs no out of bounds operations on buffers, and never dereferences null pointers.

The results of the case study are encouraging: with an annotation overhead of about four lines of annotations per five lines of code we found a total of 13 bugs in the eID applet, and 25 locations where transactions were not properly used.

This case study has been invaluable for us to improve the tool. A number of bugs were fixed and small additions were made in order to support the verification of the applets. A longer term plan has also been established to further add improvements and optimizations to the tool. In particular, the automatic generation of `open` and `close` statements will become an important part of the future work, as well as language and technology-specific extensions to the tool.

B. Annotation Inference for Separation Logic Based Verifiers[1]

Authors: Frédéric Vogels, Bart Jacobs, Frank Piessens and Jan Smans[2]

Abstract: With the years, program complexity has increased dramatically: ensuring program correctness has become considerably more difficult with the advent of multithreading, security has grown more prominent during the last decade, etc. As a result, static verification has become more important than ever.

Automated verification tools exist, but they are only able to prove a limited set of properties, such as memory safety. If we want to prove full functional correctness of a program, other more powerful tools are available, but they generally require a lot more input from the programmer: they often need the code to be verified to be heavily annotated.

In this paper, we attempt to combine the best of both worlds by starting off with a manual verification tool based on separation logic for which we develop techniques to automatically generate part of the required annotations. This approach provides more flexibility: for instance, it makes it possible to automatically check as large a part of the program as possible for memory errors and then manually add extra annotations only to those parts of the code where automated tools failed and/or full correctness is actually needed.

B.1 Introduction

During the last decade, program verification has made tremendous progress. However, a key issue hindering the adoption of verification is that a large amount of annotations is required for tools to be able to prove programs correct, in particular if correctness involves not just memory safety, but also program-specific properties. In this paper, we propose three annotation inference and/or reduction techniques in the context of separation logic-based verifiers: (1) automatic predicate folding and unfolding, (2) predicate information extraction lemmas and (3) automatic lemma application via shape analysis. All aforementioned contributions were developed in the context of the VeriFast program verifier in order to reduce annotation overhead for practical examples.

VeriFast [30] is a verification tool being developed at the K.U. Leuven. It is based on separation logic [40] and can currently be used to verify a multitude of correctness-related properties of C and Java programs. Example usages are

- ensuring that C code does not contain any memory-related errors, such as memory leaks and dereferencing dangling pointers;

[1] Appeared in: F. Vogels, B. Jacobs, F. Piessens, J. Smans. Annotation inference for separation logic based verifiers. FMOODS 2011, volume 6722 of LNCS, pages 319–333, Heidelberg, 2011. Springer.

[2] Jan Smans is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

	LOC	LOAnn	LoAnn/LOC
stack (C)	88	198 (18/16)	2.3
sorted binary tree (C)	125	267 (16/23)	2.1
bank example program (C)	405	127 (10/22)	0.31
chat server (C)	130	114 (20/26)	0.88
chat server (Java)	138	144 (19/28)	1.0
game server (Java)	318	225 (47/63)	0.71

Figure B.1: Some line count statistics

- checking that functions or methods satisfy contracts describing their intended semantics;
- preventing the occurrence of data races in multi-threaded code.

VeriFast heavily relies on programmer-provided annotations: this makes the tool very efficient, but the need for annotations can make its use quite cumbersome. To give the reader an idea of the quantity of annotations needed, we provide some quick statistics in Figure B.1: the first column contains a count of the number of lines of actual C code. The second column expresses the number of annotations in number of lines. The numbers between parentheses correspond to the number of open and close statements, respectively, which will be further explained in Section B.3.1. The third column shows the amount of annotation overhead.

We have developed three different techniques to (partially) automate verification by mechanizing the generation of (some of) the necessary annotations. In this paper, we describe these three approaches in detail. We will make the distinction between two layers:

- VeriFast's core, which requires all annotations and performs the actual verification. This core must be as small and uncomplicated as possible, as the verification's soundness relies on it.
- The automation layer, which generates as large a portion of the necessary annotations as possible, which will then in a second phase be fed to VeriFast's core for verification.

This approach maximizes robustness: we need only trust the core and can freely experiment with different approaches to automation without having to worry about introducing unsound elements, since the generated annotations will still be fully verified by the core, thus catching any errors.

In order to be able to discuss and compare our three annotation generation techniques, we need the reader to be familiar with VeriFast, therefore we included a small tutorial (Section B.2) which explains the basic concepts. Next, we explain our different approaches to automation in Section B.3. We then put them side by side in Section B.4 by comparing how many of the necessary annotations they are able to generate.

B.2 VeriFast: a quick tutorial

This section contains a quick introduction to VeriFast. It is not our intention to teach the reader how to become proficient in using VeriFast, but rather to provide a basic understanding of certain concepts on which VeriFast is built. A full tutorial is available at [31].

Listing B.1: A singly linked list node in C

```
1  struct list { struct list* next; int value; };
2  struct list* new(struct list* n, int v)
3   //@ requires emp;
4   /*@ ensures malloc_block_list(result) &*& result->next  |-> n &*&
5               result->value |-> v       &*& result != 0         @*/
6  {
7   struct list* node = malloc( sizeof( struct list ) );
8   if ( node == 0 ) abort();
9   node->next = n; node->value = v;
10  return node;
11 }
```

B.2.1 A singly linked list

Figure B.1 shows a `struct`-definition for a singly linked list node together with a function `new` which creates and initializes such a node. In order to verify this function in VeriFast, we need to provide a contract. The precondition, `emp`, indicates that the function does not require anything to be able to perform its task. The postcondition is a separating conjunction (`&*&`) of the "heap fragments":

- `malloc_block_list(result)` means that the returned pointer will point to a malloc'ed block of memory the size of a `list`. It is produced by a call to `malloc` and it is meant to be eventually consumed by a matching `free`. Failure to do so eventually leads to an unreachable `malloc_block_list` which corresponds to a memory leak.
- `result->next |-> n` means two things: it grants the permission to read and write to the `node`'s `next` field, and it tells us that the `next` field contains the value of the argument `n`. Idem for `result->value |-> v`.
- `result != 0` guarantees that the returned pointer is not null.

If this function verifies, it will mean that it is both memory safe and functionally correct. We defer the explanation of the separating conjunction `&*&` until later.

VeriFast uses symbolic execution [9] to perform verification. The precondition determines the initial symbolic state, which in our case is empty. VeriFast then proceeds by symbolically executing the function body.

1. `malloc` can either fail or succeed. In the case of `list`, its contract is
   ```
   requires emp;
   ensures  result == 0 ? emp
                        : malloc_block_list(result) &*&
                          result->next  |-> _       &*&
                          result->value |-> _;
   ```
   Either it will return `0`, which represents failure, and the heap is left unchanged. Otherwise, it returns a non-null pointer to a block of memory the size of a `struct list`, guaranteed by `malloc_block_list(result)`. It also provides access to both `next` and `value` fields of this newly created node, but does not make any promises about their values.
2. The `if` statement will intercept execution paths where `malloc` has failed and calls `abort`, causing VeriFast not to consider these paths any further.
3. Next, we assign `n` to the `next` field. This operation is only allowed if we have access to this field, which we do since the previous `if` statement filtered out any paths where allocation

failed, and a successful `malloc` always provides the required access rights. This statement transforms the `result->next |-> _` heap fragment to `result->next |-> n`.

4. Assigning `v` to `node->value` works analogously.

5. Finally, we return the pointer to the node. At this point, VeriFast checks that the current execution state matches the postcondition. This is the case, which concludes the successful verification of `new`.

A full explanation of the separating conjunction `&*&` can be found in [40]. In short, `P &*& Q` means that the heap consists of two disjoint subheaps where `P` and `Q` hold, respectively. It is used to express that blocks of memory do not overlap, and therefore changes to one object do not influence another. Separating conjunction enables the frame rule, which allows us to reason "locally". It expresses the fact that if an operation behaves some way in a given heap (`{P} op {Q}`), it will behave the same way in an extended heap (`{P &*& R} op {Q &*& R}`). For example, if we were to call `malloc` twice in a row, how can we be sure that it didn't return the same block of memory twice? We know this since, thanks to the frame rule, we get two `malloc_block_list` heap fragments joined by a separating conjunction, guaranteeing us that we are dealing with two different objects.

B.2.2 Predicates

As shown in the previous section, freely working with a single node requires carrying around quite a bit of information. This can become tiresome, especially if one considers larger structures with more fields whose types might be other structures. For this reason, VeriFast provides the ability to perform abstractions using predicates [44] which make it possible to "fold" (or close, in VeriFast terminology) multiple heap fragments into one.

Listing B.2: Predicates

```
1  predicate Node(struct list* n, struct list* m, int v) =
2   malloc_block_list(n) &*& n->next |-> m &*& n->value |-> v &*& n != 0;
3
4  struct list* new(struct list* n, int v)
5   //@ requires emp;
6   //@ ensures Node(result, n, v);
7  {
8   struct list* node = malloc( sizeof( struct list ) );
9   if ( node == 0 ) abort();
10  node->next = n; node->value = v;
11  //@ close Node(node, n, v);
12  return node;
13 }
```

Figure B.2 shows the definition for the `Node` predicate and an updated version of the `new` function. The `close` statement removes three heap fragments (`malloc_block_list` and the two field access permissions) and replaces them by a single `Node` fragment, on condition that it can ascertain that `node` is not `0` and the three fragments are present on the symbolic heap, otherwise verification fails. This closing is necessary in order to have the final execution state match the postcondition. Closing must happen last, for otherwise the field access permissions would be hidden when they are needed to initialize the node's fields (third line of the procedure body.) At a later time, whenever the need arises to access one of a node's fields, the `Node` fragment can be opened up (replacing `Node` by the three separate heap fragments on the symbolic heap) so as to make the field access permission available again.

B.2.3 Recursive predicates

A linked list consists of a chain of nodes each pointing to the next. Currently, we can only express linked lists of fixed maximum length:

```
p == 0 ? emp
       : Node(p, q, v1) &*& (q == 0 ? emp
                                    : Node(q, 0, v2)) // len 0-2
```

We can solve this problem using recursive predicates: we need to express that a list is either empty or a node pointing to another list. Figure B.3 shows the definition for `LSeg(p, q, xs)`, which stands for a cycleless singly linked list segment where `p` points to the first node, `q` is the one-past-the-end node and `xs` stands for the contents of this list segment.

Listing B.3: Recursive predicates

```
1  /*@ inductive List = Nil | Cons(int, List);
2      predicate LSeg(struct list* p, struct list* q, List Xs) =
3      p == q ? Xs == Nil
4             : Node(p,?t,?y) &*& LSeg(t,q,?Ys) &*& Xs == Cons(y,Ys); @*/
5
6  struct list* prepend(struct list* xs, int x)
7   //@ requires LSeg(xs, 0, ?Xs);
8   //@ ensures LSeg(result, 0, Cons(x, Xs));
9  {
10  struct list* n = new(xs, x);
11  //@ open Node(n, xs, x);
12  //@ close Node(n, xs, x);
13  //@ close LSeg(n, 0, Cons(x, Xs));
14  return n;
15 }
```

Figure B.3 also contains the definition for a `prepend` function. The contract fully describes its behavior, i.e. that a new element is added in front of the list, and that the pointer passed as argument becomes invalid; instead the returned pointer must be used.

B.2.4 Lemmas

The reader might wonder why a `Node` is consecutively opened and closed in Figure B.3. Let us first examine what happens without it. When VeriFast reaches the closing of the `LSeg`, execution forks due to the conditional in the `LSeg` predicate:

- `n` might be equal to `0`, in which case `xs` must be `Nil` instead of `Cons(x, Xs)`, so that closing fails. This needs to be prevented.
- `n` could also be a non-null pointer: the `Node` and original `LSeg` get merged into one larger `LSeg`, which is exactly what we want.

Therefore, we need to inform VeriFast of the fact that `n` cannot be equal to `0`. This fact is hidden within the `Node` predicate; opening it exposes it to VeriFast. After this we can immediately close it again in order to be able to merge it with the `LSeg` heap fragment.

The need to prove that a pointer is not null occurs often, and given the fact that an `open`/`close` pair is not very informative, it may be advisable to make use of a lemma, making our intentions clearer, as shown in Figure B.4. In Section B.3, we will encounter situations where lemmas are indispensable if we are to work with recursive data structures.

Listing B.4: The `NotNull` lemma

```
1  /*@ lemma void NotNull(struct list* p)
2        requires Node(p, ?pn, ?pv);
3        ensures  Node(p, pn, pv) &*& p != 0;
4      {
5        open Node(p, pn, pv); close Node(p, pn, pv);
6      }
7  @*/
8  struct list* prepend(struct list* xs, int x)
9   //@ requires LSeg(xs, 0, ?Xs);
10  //@ ensures  LSeg(result, 0, Cons(x, Xs));
11 {
12  struct list* n = new(xs, x);
13  //@ NotNull(n);
14  //@ close LSeg(n, 0, Cons(x, Xs));
15  return n;
16 }
```

B.3 Automation techniques

We have implemented three automation techniques which we discuss in the following sections. For this, we need a running example: Figure B.6 contains a fully annotated list-copying function, for which we will try to automatically infer as many of the required annotations as possible.

In our work, we have focused on verifying memory safety; automation for verifying functional properties is future work. Therefore, we simplify the `Node` and `LSeg` predicates we defined earlier by having the predicates throw away the data-related information, as shown in Figure B.5.

Listing B.5: Simplified `Node` and `LSeg` predicates

```
1  predicate Node(struct list* P, struct list* Q) =
2   P != 0 &*& malloc_block_list(P) &*& P->next  |-> Q &*& P->value |-> ?v;
3
4  predicate LSeg(struct list* P, struct list* Q) =
5   P == Q ? emp : Node(P, ?R) &*& LSeg(R, Q);
```

While it is not strictly necessary to understand the code in Figure B.6, we choose to clarify some key points:

- The `new()` function produces a new `Node(result, 0)` and always succeeds. It is just another function defined in terms of `malloc` and `abort`s on allocation failure (comparable to Figure B.2).
- `NoCycle`, `Distinct`, `AppendLSeg` and `AppendNode` are lemmas whose contracts are shown in Figure B.7.

The `copy` function comprises 12 statements containing actual C code, while the annotations consist of 31 statements, not counting the lemmas since these can be shared by multiple function definitions. We now proceed with a discussion of how to generate some of these annotations automatically.

B.3.1 Auto-open and auto-close

As can be seen in the examples, a lot of annotations consist of opening and closing predicates. This is generally true for any program: the values between parentheses in Figure B.1 indicate

Listing B.6: Copying linked lists

```
1  struct list* copy(struct list* xs)
2   //@ requires LSeg(xs, 0);
3   //@ ensures LSeg(xs, 0) &*& LSeg(result, 0);
4  {
5   if ( xs == 0 ) {
6    //@ close LSeg(0, 0);                                  // a
7    return 0; }
8   else {
9    struct list* ys = new();
10   //@ open LSeg(xs, 0);
11   //@ open Node(xs, _);                                  // a
12   //@ open Node(ys, 0);                                  // a
13   ys->value = xs->value;
14   struct list *p = xs->next, *q = ys;
15   //@ close Node(ys, 0);                                 // a
16   //@ close Node(xs, p);                                 // a
17   //@ NoCycle(xs, p);
18   //@ close LSeg(p, p);                                  // a
19   //@ close LSeg(xs, p);                                 // a
20   //@ close LSeg(ys, q);                                 // a
21   while ( p != 0 )
22    //@ invariant LSeg(xs,p) &*& LSeg(p,0) &*& LSeg(ys,q) &*& Node(q,0);
23   {
24    //@ struct list *oldp = p, *oldq = q;
25    struct list* next = new();
26    //@ Distinct(q, next);
27    //@ open Node(q, 0);                                   // a
28    q->next = next; q = q->next;
29    //@ close Node(oldq, q);                               // a
30    //@ open LSeg(p, 0);
31    //@ assert Node(p, ?pn);
32    //@ NoCycle(p, pn);
33    //@ open Node(p, _);                                   // a
34    //@ open Node(q, 0);                                   // a
35    q->value = p->value; p = p->next;
36    //@ close Node(q, 0);                                  // a
37    //@ close Node(oldp, p);                               // a
38    //@ AppendLSeg(xs, oldp); AppendNode(ys, oldq);
39   }
40   //@ open LSeg(p, 0);                                   // a
41   //@ NotNull(q);                                        // b
42   //@ close LSeg(0, 0);                                  // a
43   //@ AppendLSeg(ys, q);
44   //@ open LSeg(0, 0);                                   // a
45   return ys;
46  }
47 }
```

how many open and close statements respectively were necessary for the verification of other larger programs.

These annotations seem to be ideal candidates for automation: whenever the execution of a statement fails, the verifier could take a look at the current execution state and try opening or closing predicates to find out whether the right heap fragments are produced.

For example, assume we are reading from the `next` field of a variable `x`, which requires a heap fragment matching `x->next |-> v`. However, only `Node(x, y)` is available. Without automation, verification would fail, but instead, the verifier could try opening `Node(x, y)` and

Listing B.7: Lemmas

```
1  lemma void NoCycle(struct list* P, struct list* Q)
2    requires Node(P, Q) &*& LSeg(Q, 0);
3    ensures  Node(P, Q) &*& LSeg(Q, 0) &*& P != Q;
4  lemma void Distinct(struct list* P, struct list* Q)
5    requires Node(P, ?PN) &*& Node(Q, ?QN);
6    ensures  Node(P, PN)  &*& Node(Q, QN) &*& P != Q;
7  lemma void AppendLSeg(struct list* P, struct list* Q)
8    requires LSeg(P, Q) &*& Node(Q, ?R) &*& Q != R &*& LSeg(R, 0);
9    ensures  LSeg(P, R) &*& LSeg(R, 0);
10 lemma void AppendNode(struct list* P, struct list* Q)
11   requires LSeg(P, Q) &*& Node(Q, ?R) &*& Node(R, ?S);
12   ensures  LSeg(P, R) &*& Node(R, S);
```

find out that this results in the required heap fragment. Of course, this process could go on indefinitely given that predicates can be recursively defined. Therefore, some sort of heuristic is needed to guide the search.

We have added support for automatic opening and closing of predicates [44] to VeriFast. Without delving too much into technical details, VeriFast keeps a directed graph whose nodes are predicates and whose arcs indicate how predicates are related to each other. For example, there exists an arc from `LSeg` to `Node` meaning that opening an `LSeg` yields a `Node`. However, this depends on whether or not the `LSeg` does represent the empty list. To express this dependency, we label the arcs with the required conditions. These same conditions can be used to encode the relationships between the arguments of both predicates. For the predicate definitions from Figure B.5, the graph would contain the following:

$$\mathrm{LSeg}(a,b) \quad \begin{matrix} a \neq b \\ a = p \end{matrix} \longrightarrow \quad \mathrm{Node}(p,q) \quad p = x \longrightarrow \quad x \to \mathrm{next} \mapsto y$$

When, during verification, some operation requires the presence of a `Node(p, q)` heap fragment but which is missing, two possible solutions are considered: we can either attempt to perform an auto-open on an `LSeg(p, b)` for which we know that `p != b`, or try to close `Node(p, q)` if there happens to be a `p->next |-> ?` on the current heap.

Using this technique yields a considerable decrease in the amount of necessary annotations: each `open` or `close` indicated by `// a` is inferred automatically by VeriFast. Out of the 31 annotation statements, 17 can be generated, which is more than a 50% reduction.

B.3.2 Autolemmas

We now turn our attention to another part of the annotations, namely the lemmas. On the one hand, we have the lemma definitions. For the moment, we have made no efforts to automate this aspect as lemmas need only be defined once, meaning that automatic generation would only yield a limited reduction in annotations.

On the other hand we have the lemma applications, which is where our focus lies. Currently, we have only implemented one very specific and admittedly somewhat limited way to automate lemma application. While automatic opening and closing of predicates is only done when the need arises, VeriFast will try to apply all lemmas regarding a predicate `P` each time `P` is produced, in an attempt to accumulate as much extra information as possible. This immediately gives rise to some obvious limitations:

- It can become quite inefficient: there could be many lemmas to try out and many matches are possible. For example, imagine a lemma operates on a single `Node`, then it can be applied to every `Node` on the heap, so it is linear with the number of `Node`s on the heap. If however it operates on two `Node`s, matching becomes quadratic, etc. For this reason, two limitations are imposed: lemmas need to be explicitly declared to qualify for automatic application, and they may only depend on one heap fragment.
- Applying lemmas can modify the execution state so that it becomes unusable. For example, if the `AppendLSeg` lemma were applied indiscriminately, `Node`s would be absorbed by `LSeg`s, effectively throwing away potentially crucial information (in this case, we "forget" that the list segment has length 1.) To prevent this, autolemmas are not allowed to modify the symbolic state, but instead may only extend it with extra information.

Given these limitations, in the case of our example, only one lemma qualifies for automation: `NotNull`. Thus, every time a `Node(p, q)` heap fragment is added to the heap, be it by closing a `Node` or opening an `LSeg` or any other way, VeriFast will immediately infer that `p != 0`. Since we only needed to apply this lemma once, we decrease the number of annotations by just one line (Figure B.6, indicated by `// b`).

B.3.3 Automatic shape analysis

Ideally, we would like to get rid of all annotations and have the verifier just do its job without any kind of interaction from the programmer. However, as mentioned before, the verifier cannot just guess what behavior a piece of code is meant to exhibit, so that it can only check for things which are program-independent bugs, such as data races, dangling pointers, etc.

Our third approach for reducing annotations focuses solely on shape analysis [17], i.e. it is limited to checking for memory leaks and invalid pointers dereferences. Fortunately, this limitation is counterbalanced by the fact that it is potentially able to automatically generate all necessary annotations for certain functions, i.e. the postcondition, loop invariants, etc.

In order to verify a function by applying shape analysis, we need to determine the initial program state. The simplest way to achieve this is to require the programmer to make his intentions clear by providing preconditions. Even though it appears to be a concession, it has its advantages. Consider the following: the function `length` requires a list, but `last` requires a non-empty list. How does the verifier make this distinction? If `length` contains a bug which makes it fail to verify on empty lists, should the verifier just deduce it is not meant to work on empty lists?

We could have the verifier assume that the buggy `length` function is in fact correct but not supposed to work on empty lists. The verification is still sound: no memory-related errors will occur. A downside to this approach is that the `length` function will probably be used elsewhere in the program, and the unnecessary condition of non-emptiness will propagate. At some point, verification will probably fail, but far from the actual location of the bug. Requiring contracts thus puts barriers on how far a bug's influence can reach.

One could make a similar case for the postconditions: shape analysis performs symbolic execution and hence ends up with the final program state. If the programmer provides a postcondition, it can be matched against this final state. This too will prevent a bug's influence from spreading.

Our implementation of shape analysis is based on the approach proposed by Distefano et al. [17]. The idea is simple and very similar to what has been explained earlier in Section B.3.1: during the symbolic execution of a function, it will open and close the predicates as necessary to satisfy the precondition of the operations it encounters. However, the analysis has a more thorough understanding of the lemmas: it will know in what circumstances they need to be

without abstraction	with abstraction
`Node(p', p) &*& LSeg(p, 0)`	`LSeg(p', p) &*& LSeg(p, 0)`
`Node(p', p1) &*& Node(p1, p) &*& LSeg(p, 0)`	`LSeg(p', p) &*& LSeg(p, 0)`

Figure B.2: Finding a fixed point

applied. A good example of this is the inference of the loop invariant where shape analysis uses the lemmas to abstract the state, which is necessary to prevent the symbolic heap from growing indefinitely while looking for a fixpoint. Consider the following pseudocode:

$$p' := p; \mathbf{while}\ p \neq 0\ \mathbf{do}\ p := p{\rightarrow}\mathrm{next}\ \mathbf{end}$$

Initially, the symbolic heap contains `LSeg(p, 0)`. To enter the loop, `p` needs to be non-null, hence it is a non-empty list and can be opened up to `Node(p', p1) &*& LSeg(p1, 0)`. During the next iteration, `p1` can be null (the loop ends) or non-null (a second node). Thus, every iteration adds the possibility of an extra node. This way, we'll never find a fixed point. Performing abstraction will fold nodes back into `LSeg`s. The difference is shown in Figure B.2. One might wonder why the abstraction doesn't also merge both `LSeg`s into a single `LSeg`. The reason for this is that the local variable `p` points to the start of the second `LSeg`: folding would throw away information deemed important.

For our purposes, the algorithms defined in [17] need to be extended so that apart from the verification results of a piece of code and final program states which determine the postcondition, they also generate the necessary annotations to be added to the verified code. This way, the results can be checked by VeriFast, keeping our trusted core to a minimum size (i.e. we do not need to trust the implementation of the shape analysis tool), and extra annotations can be added later on if we wish to prove properties other than memory safety.

For our example, shape analysis is able to deduce all `open` and `close` annotations, the lemma applications, the loop invariant and the postcondition (in our implementation, we chose to require only the precondition and we manually check that the generated postcondition is as intended). Hence, the number of necessary annotations for Figure B.6 is reduced to 1, namely the precondition.

B.4 Comparison

In order to get a better idea of by how much we managed to decrease the number of annotations, we wrote a number of list manipulation functions. There are four versions of the code:

(A) A version with all annotations present.

(B) An adaptation of (A) where we enabled auto-open and auto-close.

(C) A version where we take (B) and make `NotNull` an autolemma (Section B.3.2).

(D) Finally, a minimal version with only the required annotations to make our shape analysis implementation (Section B.3.3) able to verify the code.

Figure B.3 shows how the annotation line counts relate to each other.

C-code	#code	A	B	C	D	lemma	A	B	C
`length`	10	12	9	9	1	`Distinct`	9	7	7
`sum`	11	11	7	7	1	`NotNull`	7	6	6
`destroy`	9	6	4	4	1	`AppendNode`	19	16	16
`copy`	23	32	15	14	1	`AppendLSeg`	27	19	18
`reverse`	12	9	5	5	1	`AppendNil`	9	7	6
`drop_last`	28	28	13	13	1	`NoCycle`	11	10	9
`prepend`	7	5	3	3	1				
`append`	13	20	11	11	1				

	#code	A	B	C	D
total	113	205	132	128	8

Figure B.3: Annotation line count comparison

B.5 Related work

Smallfoot [9] is a verification tool based on separation logic which given pre- and post-conditions and loop invariants can fully automatically perform shape analysis. It has been extended for greater automation [52], for termination proofs [11, 16], fine-grained concurrency [13] and lock-based concurrency [24].

jStar [18] is an another automatic separation logic based verification tool which targets Java. One only needs to provide the pre- and post-conditions for each method, after which it attempts to verify it without extra help. It is able to infer loop invariants, for which it uses a more generalized version of the approach described by Distefano et al. [17]. This is achieved by allowing the definition of user-defined rules (comparable to our lemmas) which are then used by the tool to perform abstraction on the heap state during the fixed point computation.

A third verification tool based on separation logic is SPACEINVADER [17, 52], which performs shape analysis on C programs. ABDUCTOR, an extension of this tool, uses a generalized form of abduction [12], which gives it the ability not only to infer loop invariants and postconditions, but also preconditions.

Other tools which don't rely on separation logic are for example KeY [1] (dynamic logic [26]), Spec♯ [6], Chalice [34], Dafny [35], and VCC [15], the latter three being based on Boogie2 (verification condition generation [7, 33]). Still other alternatives to separation logic are implicit dynamic frames [50] and matching logic [49], the latter being an approach where specifications are expressed using patterns which are matched against program configurations.

B.6 Conclusion

We can divide verifiers in two categories.

- Fully automatic verifiers which are able to determine whether code satisfies certain conditions without any help of the programmer. Unfortunately, this ease of use comes with a downside: these tools can only check certain properties for certain patterns of code. More ambitious verifications such as ensuring full functional correctness remains out of the scope of these automatic verifiers, since correctness only makes sense with respect to a specification, which needs to be provided by the programmer.
- Non-automatic tools are able to perform more thorough verifications (such as full functional correctness), but these require help from the programmer.

In practice, given a large body of code, it is often sufficient to check only automatically provable properties except for a small section of critical code, where a proof of full functional correctness is necessary. Neither of the above two options is then ideal. Our proposed solution is to combine the best of both worlds by using the following verification framework: at the base lies the non-automatic "core" verifier (in our case VeriFast), which will be responsible for performing the actual verification. To achieve this, it requires code to be fully annotated, but in return, it has the potential of checking for a wide variety of properties. On this base we build an automation layer, consisting of specialized tools able to automatically verify code for specific properties. Instead of just trusting the results of these tools, we require them to produce annotations understood by the core verifier.

A first advantage is that only the core verifier needs to be trusted. Indeed, in the end, all automatically produced annotations are fed back to the core verifier, so that unsoundnesses introduced by buggy automation tools will be caught.

A second advantage is that it allows us to choose which properties are checked for which parts of the code. For example, in order to verify a given program, we would start with unannotated code, on which we would apply an automatic verification tool, such as the shape analysis tool discussed in Section B.3.3. This produces a number of annotations, which are fed to the core verifier. If verification succeeds, we know the application contains no memory-related errors.

Now consider the case where a certain function `foo` appears to be troublesome and shape analysis fails to verify it, which could mean that all other parts of the code which call this function also remain unverified. In order to deal with this problem the programmer can manually add the necessary annotations for `foo`, let the core verifier check them, and then re-apply the shape analysis tool, so that it can proceed with the rest of the code.

After the whole program has been proved memory-safe, one can proceed with the critical parts of the code where a proof of full functional correct is required. Thus, it makes an iterative incremental approach to verification possible where manually added annotations aid the automatic tools at performing their task.

In this paper, we presented preliminary experience gained in our work in progress towards this goal. Future work includes gaining additional experience with larger programs, gaining experience with the usability of an iterative infer-annotate process, and improving the power of the inference algorithm.

C. Additional information on packages of the integrated POPS scenario

For each package of the integrated POPS scenario, we give in this appendix some additional details on its content: the number of classes and interfaces defined, the number of methods defined in each class/interface, the visibility of these methods from outside of the package in which they are defined, and finally the number of *jump targets* in their bytecode. Actually, a *jump target* corresponds to a (bytecode) instruction of a method that is directly reachable by two different control flow paths.

C.1 The `newepurse.cap` package

The `newepurse.cap` package contains 2 interfaces and 3 classes:

- `IEPurseServiceCredit` interface inherits `Shareable` and defines 2 methods:

Method	#param.	Visible	Implem.	#jump targets
`charge`	1	✓		
`transaction`	1	✓		

- `IEPurseServiceDebit` interface inherits `Shareable` and defines 1 method:

Method	#param.	Visible	Implem.	#jump targets
`debit`	1	✓		

- `EPurseServiceCredit` class that implements `IEPurseServiceCredit` and the 3 following methods:

Method	#param.	Visible	Implem.	#jump targets
constructor	1		✓	2
`charge`	1	✓	✓	12
`transaction`	1	✓	✓	12

- `EPurseServiceDebit` class that implements `IEPurseServiceDebit` and the 2 following methods:

Method	#param.	Visible	Implem.	#jump targets
constructor	1		✓	2
`debit`	1	✓	✓	12

- `NewEPurseApplet` class that extends `Applet` and implements the 12 following methods:

Method	#param.	Visible	Implem.	#jump targets
constructor	0		✓	3
`install`	3	✓	✓	3
`process`	1	✓	✓	18
`getBalance`	1		✓	7
`credit`	1		✓	10
`debit(APDU)`	1		✓	8
`debit(short)`	1	✓	✓	4
`transaction`	1	✓	✓	4
`charge`	1	✓	✓	4
`setLimit`	1		✓	10
`addClientApplet`	1		✓	26
`searchClientAID`	1	✓	✓	6
`getClientLimit`	1	✓	✓	1
`getShareableInterfaceObject`	2	✓	✓	13

C.2 The `newjticket.cap` package

The `newjticket.cap` package contains 1 class:

- `NewJTicketApplet` class that extends `Applet` and implements the 7 following methods:

Method	#param.	Visible	Implem.	#jump targets
constructor	0	✓	✓	3
`install`	3	✓	✓	3
`process`	1	✓	✓	12
`getCounter`	1		✓	5
`useTicket`	1		✓	6
`buyTickets`	1		✓	12
`setEPurseAID`	1		✓	12

C.3 The `neweidapplet.cap` package

The `neweidapplet.cap` package contains 1 interface and 6 classes:

- `INewEidPoints` interface inherits `Shareable` and defines 1 method:

Method	#param.	Visible	Implem.	#jump targets
`sharePoints`	1	✓		

- `NewEidPoints` class that implements inherits `INewEidPoints` and defines 2 methods:

Method	#param.	Visible	Implem.	#jump targets
constructor	0	✓	✓	2
`sharePoints`	1	✓	✓	1

- `NewEidCard` class that extends `Applet` and implements the 50 following methods:

Method	#param.	Visible	Implem.	#jump targets
install	3	✓	✓	2
initializeFileSystem	0		✓	16
eraseBinary	2		✓	14
updateBinary	2		✓	21
fileAccessAllowed	1		✓	12
getCardData	2		✓	18
readBinary	2		✓	23
activateFile	2		✓	12
clear	0		✓	6
initializeEmptyLargeFiles	0		✓	7
initializeKeyPairs	0		✓	11
selectByFileIdentifier	2		✓	20
selectByPath	2		✓	22
initializePins	2		✓	9
constructor	0	✓	✓	20
select	0	✓	✓	2
deselect	0	✓	✓	2
process	1	✓	✓	45
verifyPin	2		✓	16
checkPin	2		✓	11
changePin	2		✓	13
userChangePin	2		✓	11
administratorChangePin	2		✓	21
isNewPinFormattedCorrectly	2		✓	15
isNewPinCorrectValue	1		✓	10
logOff	2		✓	12
unblock	2		✓	8
prepareForSignature	2		✓	22
generateSignature	2		✓	34
generatePkcs1Md5Signature	2		✓	18
generatePkcs1Sha1Signature	2		✓	18
generatePkcs1Signature	2		✓	18
preparePkcs1ClearText	3		✓	9
generateKeyPair	1		✓	22
getPublicKey	1		✓	53
putPublicKey	2		✓	2
eraseKey	2		✓	6
activateKey	2		✓	4
deactivateKey	2		✓	2
internalAuthenticate	2		✓	28
getResponse	2		✓	13
getChallenge	2		✓	12
selectFile	2		✓	6
setPreviousApduType	1		✓	1
getPreviousApduType	0		✓	1
setSignatureType	1		✓	1
getSignatureType	0		✓	1
deactivateFile	2		✓	12
getShareableInterfaceObject	2	✓	✓	4
askForCharge	0		✓	6

- `File` class that implements the 4 following methods:

Method	#param.	Visible	Implem.	#jump targets
constructor	1	✓	✓	2
getFileID	0	✓	✓	1
setActive	1	✓	✓	1
isActive	0	✓	✓	1

- `ElementaryFile` class that extends `File` and implements the 10 following methods:

Method	#param.	Visible	Implem.	#jump targets
constructor	3	✓	✓	3
constructor	3	✓	✓	3
`getData`	0	✓	✓	4
`getCurrentSize`	0	✓	✓	4
`getMaxSize`	0	✓	✓	1
`eraseData`	1	✓	✓	2
`updateData`	4	✓	✓	2
`getFileID`	0	✓	✓	1
`setActive`	1	✓	✓	2
`isActive`	0	✓	✓	2

- `DedicatedFile` class that extends `File` and implements the 8 following methods:

Method	#param.	Visible	Implem.	#jump targets
constructor	1	✓	✓	2
constructor	2	✓	✓	3
`getParent`	0	✓	✓	1
`addSibling`	1	✓	✓	3
`getSibling`	1	✓	✓	7
`getFileID`	0	✓	✓	1
`setActive`	1	✓	✓	1
`isActive`	0	✓	✓	1

- `MasterFile` class that extends `DedicatedFile` and implements the 7 following methods:

Method	#param.	Visible	Implem.	#jump targets
constructor	0	✓	✓	2
`getParent`	0	✓	✓	1
`addSibling`	1	✓	✓	2
`getSibling`	1	✓	✓	1
`getFileID`	0	✓	✓	1
`setActive`	1	✓	✓	1
`isActive`	0	✓	✓	2

C.4 The `newmypackage.cap` package

The `newmypackage.cap` package contains 1 interface and 6 classes:

- `INewMyAppletPoints` interface inherits `Shareable` and defines 1 method:

Method	#param.	Visible	Implem.	#jump targets
`sharePoints`	1	✓		

- `NewMyAppletPoints` class that implements inherits `INewMyAppletPoints` and defines 2 methods:

Method	#param.	Visible	Implem.	#jump targets
constructor	0	✓	✓	2
`sharePoints`	1	✓	✓	1

- `NewMyApplet` class that extends `Applet` and implements the 14 following methods:

Method	#param.	Visible	Implem.	#jump targets
constructor	5		✓	7
`install`	3	✓	✓	10
`process`	1	✓	✓	17
`processSelectCmd`	1		✓	19
`checkIncomingData`	3		✓	9
`processPutData`	1		✓	10
`getOptionalDataIndex`	1		✓	2
`processAppendRecord`	1		✓	38
`processDeleteRecord`	1		✓	33
`processReadRecord`	1		✓	20
`sh`	1		✓	1
`askForSharingPoints`	0		✓	6
`askForPayment`	0		✓	8
`getShareableInterfaceObject`	2	✓	✓	4

D. Structure of on-device repositories of methods footprints for the implementation of the *global policy model*

The structure of methods footprints repositories managed on-device is given in Figure D.1 and detailed in the remaining part of this section.

```
ondevice_footprint_repositories {
    u1 package_footprints_size;
    ondevice_package_footprint package_footprints[];
}

ondevice_package_footprint {
    u1 package_index;
    u2 class_footprints_size;
    ondevice_class_footprint class_footprints[];
}

ondevice_class_footprint {
    u2 classref;
    u1 class_token;
    u2 method_footprints_size;
    ondevice_method_footprint method_footprints[];
}

ondevice_method_footprint {
    u1 bitfield;
    u1 method_token; /* present according to bitfield */
    u2 method_offset; /* present according to bitfield */
    footprint_t complete_footprint;
}
```

Figure D.1: Data structures of footprints repositories managed on-device for the global policy model.

The `ondevice_footprint_repositories` structure describes a set of binary repositories of methods footprints:

- `package_footprints_size` represents the size in bytes of the `package_footprints` field;
- `package_footprints` contains a `ondevice_package_footprint` entry for each package currently installed on the system;

The `ondevice_package_footprint` structure describes the footprints attached to all methods of all classes (and all interfaces) of an installed package:

- `package_index` contains the internal package index of the current package;

- `class_footprints_size` represents the size in bytes of the `class_footprints` field;
- `class_footprints` maps to each class (or interface) of the current package the footprints of its methods;

The `ondevice_class_footprint` structure is used to describe the methods footprints of an installed class or interface:

- `classref` contains the location (*i.e.*the offset) in the Class Component (*c.f.*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_footprints_size` represents the size in bytes of the `method_footprints` field;
- `method_footprints` maps to each method of the current class (or interface) its footprint;

The `ondevice_method_footprint` structure describes the footprint of a method:

- `bitfield` is the mask of modifiers coming from the GPCC, with the additional modifier indicating which repository this `complete_footprint` belongs to:

Mask	0x01	
Value	0x00	repository of verified footprints (repository R in the model, Section 5.6.3 of the deliverable D6.3)
	0x01	repository of believed footprints (repository R_{tmp} in the model, Section 5.6.3 of the deliverable D6.3)

- `method_token` represents the static/virtual/interface 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 (*c.f.*JCVM specifications) if the method is implemented (*i.e.*not an abstract method or a method definition in an interface), according to `bitfield`;
- `complete_footprint` contains the binary encoded footprint of the current method.

E. Structure of on-device repositories of methods signatures for the implementation of the *non-interference model*

The structure of methods signatures repositories managed on-device is given in Figure E.1 and detailed in the remaining part of this section.

```
ondevice_signature_repositories {
    dataflow_t dataflow;
    u1 package_signatures_size;
    ondevice_package_signature package_signatures[];
}

ondevice_package_signature {
    u1 package_index;
    u2 class_signatures_size;
    ondevice_class_signature class_signatures[];
}

ondevice_class_signature {
    u2 classref;
    u1 class_token;
    u2 method_signatures_size;
    ondevice_method_signature method_signatures[];
}

ondevice_method_signature {
    u1 bitfield;
    u1 method_token; /* present according to bitfield */
    u2 method_offset; /* present according to bitfield */
    u2 signature_size;
    signature_t complete_signature;
}
```

Figure E.1: Data structures of signatures repositories managed on-device for the non-interference model.

The `ondevice_signature_repositories` structure describes a set of binary repositories of signatures:

- `dataflow_t` is a two dimensional bit array where each cell bit contains the secret flow policy from one security domain to another (a bit set to 1 means the secret flow is permitted) (Section 6.3.1 of the deliverable D6.3);
- `package_signatures_size` represents the size in bytes of the `package_signatures` field;
- `package_signatures` contains a `ondevice_package_signature` entry for each package installed on the system.

The `ondevice_package_signature` structure describes the signatures attached to all methods of all classes (and all interfaces) of an installed package:

- `package_index` contains the internal package index of the current package;
- `class_signatures_size` represents the size in bytes of the `class_signatures` field;
- `class_signatures` maps to each class (or interface) of the current package the signatures of its methods.

The `ondevice_class_signature` structure is used to describe the methods signatures of a class or an interface:

- `classref` contains the location (*i.e.*the offset) in the Class Component (*c.f.*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_signatures_size` represents the size in bytes of the `method_signatures` field;
- `method_signatures` maps to each method of the current class (or interface) its signature.

The `ondevice_method_signature` structure describes the signature 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

Mask	0x01	
Value	0x00	repository of verified signatures
	0x01	repository of believed signatures[1]

- `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 (*c.f.*JCVM specifications) if the method is implemented (*i.e.*not an abstract method or a method definition in an interface) according to `bitfield`;
- `signature_size` contains the size in bytes of the `complete_signature` field;
- `complete_signature` contains the byte encoded signature of the current method where each byte corresponds to a flow relation between two abstract entities (Section 6.1.2 of the deliverable D6.3).

F. SxC for the OSGi Platforms

F.1 Introduction

The Open Services Gateway Initiative (OSGi) framework [42, 5] is one of the most flexible solutions for the deployment of pervasive services in home, office, or automobile environments [25, 32, 47]. OSGi-compatible implementations constitute the backbone of many recent proposals for embedded systems [8] or other industry-based services [45]. The OSGi services are also the basic building blocks for service mash-ups extending the classical "smart homes" scenarios to richer settings [39].

In a nutshell, the OSGi framework redefines the modular system of Java by introducing *bundles*: JAR files enhanced with specific metadata. The *services* layer connects bundles in a dynamic way with a publish-find-bind model for Java objects. An OSGi-based system has several advantages over the traditional JAR modules. First, OSGi provides a robust integrated environment where bundles can be published and exported to be used by other bundles. Second, OSGi provides versioning of bundles for every new deployment and maintains the bundle lifecycles. And third, the bundles can be updated dynamically at run time without restarting the system and seamlessly to other bundles.

As a result, an OSGi platform is expected to be highly dynamic. All pervasive and mash-up applications expect that bundles can be installed, updated or removed at any time depending on business needs.

From a security perspective, the possibility of bundle interactions is a threat for bundle owners. Since bundles can contain sensitive data or activate sensitive operations (such as locking doors and windows of somebody's house), it is important to ensure that the security policy of each bundle owner is respected by other bundles. However, such aspects have been only partially investigated.

How do we make sure that one's services are invoked by one's authorized siblings? A simple solution is to rely on service-to-service authentication to identify the services and then interleave functional and security logic into bundles, for example, by using aspect-oriented programming [47]. However, this decreases the benefits of using a common platform for service deployment and significantly hinders evolution and dynamicity: any change to the security policy would require redeployment of the bundle (even if its functionalities are unchanged). Vice versa, any changes in the bundle's code would require redeployment of security as well.

Our solution is to use the Security-by-Contract methodology (SxC) [21, 20] for load time security verification in order to separate security and the business logic while achieving a sufficient protection of applications among themselves.

F.2 The SxC Architecture

We assume at least high-level understanding of the main notions of the OSGi platform such as *bundles* (the OSGi components made by developers), *services* (plain old Java objects connecting bundles in a dynamic fashion by the means of publish-find-bind model), *lifecycle* API (the API to install, start, stop, update, and uninstall bundles) and *modules* – the layer that defines how a bundle can import and export code.

The SxC framework consists of two main components: the ClaimExtractor and the PolicyChecker. The verification workflow is described on Figure F.1.

Informally, the SxC process starts when a new bundle B is loaded. The ClaimExtractor component then accesses the manifest file, retrieves the information about imported and exported packages and obtains the bundle contract. Then the ClaimExtractor reads the permissions.perm file, which contains local bundle permissions, extracts permissions requested by the bundle B and related to services retrieval, packages importing, requirements of bundles, etc, and combines this information into the overall "security claims and needs" of the bundle. Then the PolicyChecker component receives the result from the ClaimExtractor and matches it with the security policy of the platform, that aggregates the security policies of all the installed bundles, and with the functional state of the platform (installed bundles, running services, etc). If the PolicyChecker failed on either of the checks, the bundle is removed from the platform. Otherwise, it is installed and the security policy of the platform is updated by including the security requirements of B.

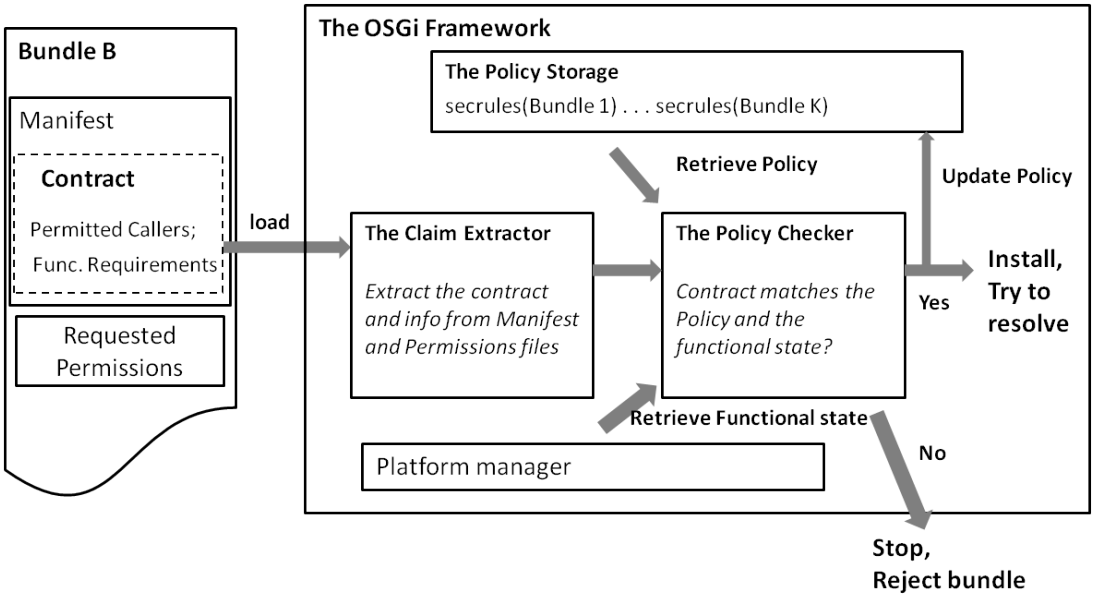

Figure F.1: SxC Workflow

When a bundle is removed we do not run the SxC checks, as only the functional requirements can fail in this case and a bundle is free to define its actions in case its functional requirements will fail in the future. It will receive synchronously a notification about the removal of a necessary service or an arrival of a rival bundle, and it's up to the bundle provider to decide what to do.

The SxC checks will be run in case of bundle code update or bundle policy update. These checks, however, are variations of the installation scenario. Thus in the sequel we will focus on only the installation scenario as the most representative one.

In terms of technical realization, the SxC steps can be easily integrated with the OSGi framework. The key requirement is getting the correct and up-to date information about the state of the platform and being able to access the received bundle *before* it is deployed on the

Provider	Requirement
securechange.eu	*Access to 'eu.securechange.homes.feed.gateway.FeedBundleService' is allowed only for bundles signed by securechange.eu provider*
example.com	*Access to 'com.example.FeedServiceBundle' is allowed only for bundles signed by example.com provider*
example.com	*Only bundles signed by example.com can import the package containing 'com.example.FeedService'*
example.com	*Access to 'com.example.FeedHappyFarmerService' service can be granted only for bundles signed by facebook.com or by example.com*
facebook.com	-

Table F.1: The running example policies

platform. The SxC framework itself can be a bundle, provided it can access the service registry, the framework policy file, the lifecycle layer, etc) and the manifest file and permissions.perm file of bundles being loaded on the platform.

F.2.1 The Running Example

We consider as a case study an OSGi platform deployed as a service gateway in a smart home. Let us consider Alice, the smart home resident, that can install untrusted bundles on the OSGi platform. The owner of the platform is a telecom provider Telefonica, that wants to avoid security problems, as otherwise Alice will call the provider hotline and might consider it liable for any problem.

Alice can download bundles for entertainment (news RSS feeds, media bundles from TV providers) or even bundles with traditional Internet content (like Facebook or Twitter), as nowadays new TV sets can be used for all these purposes. The interested reader can refer to [28] for more details on the news feed scenario which we also consider in the current report. In the running scenario we have used fictional names, but they give an idea of realistic bundle interactions and possible policies regarding these interactions.

Alice wants to be kept up to date on financial news. She downloads and installs the bundle 'eu.securechange.homes.feed.gateway.impl', which can provide her an interface with a series of news feeds, including some premium subscriptions. This bundle includes a service 'eu.securechange.homes.feed.gateway.FeedBundleService' that retrieves updates from the news feeds. However, Alice later adds another news feed provider and installs 'com.exampe.FeedServiceBundle', that also provides the service for feed retrieval 'com.example.FeedService' and the service 'com.example.FeedHappyFarmerService ' that allows Alice to receive regular updates of game events from her Happy Farm account on facebook.com.

The bundle providers want to ensure that their security policies related to bundles and services usage are enforced on the Alice's platform. Their requirements are listed in Table F.1. We see that for news feed services usage the providers securechange.eu and example.com want to deploy a classic same origin policy, as they do not want to share the sensitive news feed information. Instead, for its Happy Farm service the example.com provider wants to share also with bundles coming from the facebook.com provider.

In the sequel we will refer to the securechange.eu provider as $SC.eu$ provider, to the 'eu.securechange.homes.feed.gateway.impl' as bundle A and to the service 'eu.securechange.homes.-feed.gateway.FeedBundleService' as service S_A. Correspondingly, we will denote example.com provider as $Ex.com$ provider, its bundle 'com.example.FeedServiceBundle' we will denote as bundle B, service 'com.example.FeedService' we will denote as service S_B and service

'com.example.FeedHappyFarmerService' we will denote as S_{bf}. Finally, facebook.com we will refer to as $FB.com$ provider, Facebook bundle will be referred to as F bundle, package 'com.facebook.FarmerPackage' as P_F and service 'com.facebook.FarmerService' as S_F. In the examples we assume the default version of bundles and packages 0.0.0, unless specified otherwise.

The OSGi platform at Alice's smart home has to ensure that the requirements of each provider are respected. We will discuss in Section F.3.1 how the OSGi platform itself can enforce the requirements of the providers $SC.eu$ and $Ex.com$ and why this approach is not satisfactory. We will also demonstrate that there can exist similar requirements of bundle providers that cannot be enforced by the OSGi platform at all. However, prior to this discussion we need to introduce more details about the OSGi technology itself.

F.3 An OSGi Technology Overview

The current section is dedicated to the OSGi platform (v. 4.3 [42]) details relevant to the running scenario. An OSGi bundle is a JAR file enhanced by specific meta-data. A bundle includes the manifest.mf file (*manifest file* in the sequel) containing the necessary OSGi meta-data: the symbolic name of the bundle, its version, the dependencies and the provided resources. Some packages of a bundle can be *exported* (accessible for other bundles on the platform). A bundle also typically includes an *activator* (used for bootstrapping when the bundle is started) and a file with security permissions requested by the developer).

The OSGi specification defines *manifest headers* used by developers to describe a bundle, some of them are:

- **Export-Package** contains a list of exported packages.
- **Import-Package** declares the imported packages.
- **Provided-Capability** specifies a set of provided capabilities.
- **Require-Bundle** specifies that all exported packages from another bundle must be imported.

The OSGi system maintains the evolving lifecycles of bundles. A bundle must first be installed. It can be resolved when all its dependencies are resolved. In the Resolved state, the bundle activator can be launched in order to configure the bundle and possibly launch some services. Figure F.2 presents the lifecycle of a bundle, that is managed by the OSGi platform in a centralized fashion.

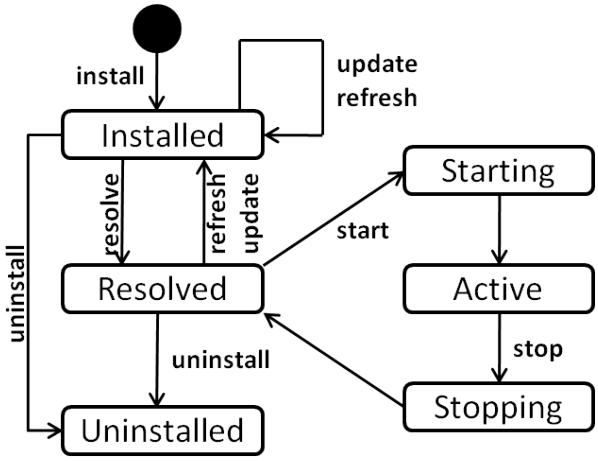

Figure F.2: Lifecycle of a bundle

Bundles can depend on external entities (they can require other bundles, an execution envi-

ronment, a specific library, etc). Once a bundle is started, it assumes that those dependencies were resolved. Bundles can express their dependencies as *requirements* on *capabilities* that are provided by the runtime environment or other bundles.

Capabilities are attribute sets in a specific namespace and requirements are filter expressions that assert the attributes of the capabilities. A requirement is satisfied when there is at least one capability that matches the filter. The requirements are transitive: bundles can only provide their capabilities when their own requirements are satisfied. The ***Require-Capability*** and ***Provided-Capability*** headers are manifest headers that declare generic requirements and capabilities. The OSGi framework can match the requirements to capabilities of other bundles in the resolving phase.

Example 1 *Let us consider the bundle A from Section F.2.1. We can assume that this bundle exports the package 'eu.securechange.homes.feed.gateway.FeedBundlePackage' and requires the OSGi execution environment with a version greater or equal 4.0. This bundle then has the following headers in its manifest file:*

- ***Bundle-SymbolicName****: eu.securechange.homes.feed.gateway.impl;*
- ***Bundle-Version****: 1.0;*
- ***Export-Package****: eu.securechange.homes.feed.gateway.FeedBundlePackage; version = 0.0.0;*
- ***Require-Capability****: osgi.ee; filter="(version $\geq$ 4.0)".*

The require/provide capabilities mechanism is complex enough, but it has its drawbacks. One of them is that the bundles themselves declare what they are going to provide to the platform. In fact, it may happen that a bundle declares certain functionality (for example, provision of a certain API), but does not provide it in reality. Consequently, some bundles can be resolved at a moment when their needs are not satisfied, potentially crashing the framework.

Bundles can interact through two complementary mechanisms: the export/import of packages and the service registration/lookup facility. A *service* is a normal Java object that is registered under a Java interface with the service registry. Bundles can register services and search for them, or receive notifications when their registration state changes.

A *service interface* is the specification of the service's public methods. A developer creates a service by implementing its service interface and registering it with the framework service registry. When requesting a service from the framework, a bundle specifies the name of the service interface and, optionally, a filter to narrow the search. In response, the framework first sends ServiceReference objects of the services that satisfy the search filter. The actual service object can then be acquired by passing the specific ServiceReference to the platform, providing the caller has the ServicePermission[ServiceReference, GET] permission.

Security of the OSGi platform is based on the Java 2 security architecture. Each bundle is associated with a set of permissions, that are queried at runtime. The OSGi platform can authenticate code by download location or by signer (digital signature). The Conditional Permission Admin service manages the permissions based on a comprehensive conditional model.

A bundle has *local permissions* defined by the developer in the file permissions.perm (*permissions file* in the sequel). These are the actual detailed permissions needed by this bundle to operate. A framework also provides an administrative service to associate a set of *system permissions* with a bundle. The bundle's *effective permissions* are the intersection of the local permissions and the system permissions. That is, a bundle cannot get more permissions than its local permissions set. Thus a bundle developer can limit the possible permissions of a bundle, but she cannot require a minimum set of necessary permissions to be granted and she cannot directly influence the set of system permissions granted to the bundle.

The OSGi specification defines ServicePermission, BundlePermission and PackagePermission. These are used to provide access to getting or registering a service, importing or exporting

bundles and packages respectively.

Example 2 *Bundle A from example 1 needs the following permissions (among others):*
• ServicePermission[eu.securechange.homes.feed.gateway.FeedBundleService, REGISTER] *to be able to register the service;*
• PackagePermission[eu.securechange.homes.feed.gateway.FeedBundlePackage, EXPORTONLY] *to be able to export the package.*

F.3.1 Security Challenges

We assume the framework can host multiple third-parties' bundles, and these bundles can freely register services. The goal of the telecom provider (Telefonica) running the platform is to make sure that there are no undesired security or functionality problems among different bundles installed by the end user (who most likely does not even know what is a bundle and just sees the web interfaces of the services). Thus a threat scenario under investigation is a case when a bundle gains unauthorized access to the sensitive data of another bundle (security threat), or a bundle is malfunctioning due to unavailability of a certain service (functionality threat). We now discuss these threats separately in the light of the running example from Section F.2.1.

A confidentiality attack can be realized by the bundle A of provider $SC.eu$ getting access to the premium news feeds in service S_B of provider $Ex.com$. This might happen if A imports the package containing the service S_B definition, requires the bundle B (thus importing all its exported packages), or tries to get a reference to this service from the Service Registry and then get access to the object referenced.

We now discuss how the current OSGi security management can address this security threat. Import of a package or a require-bundle action can be granted if the requiring bundle has corresponding permissions. Simple reviewing of the manifest file and permissions file of the bundle A can report about a (potential) attempt to interact with the bundle B. However, there is no convenient and simple way for the owner of the bundle B, the $Ex.com$ provider, to declare which other bundles are allowed to import its packages.

Package importing can be guarded by the permissions mechanism, as we discussed before. Currently only the platform owner (the telecom provider) can define and manage policies in the Conditional Permission Admin policy file. The $Ex.com$ provider might contact the telecom provider to ask him to set the required permissions, or its bundle B, being granted the necessary permissions, can add new permissions to the Conditional Permission Admin policy file. These approaches are organizationally cumbersome and costly, as they require the operator to push the changes to its customers before any downloads of $ex.com$ bundles, even the customers have no intention of using them. The SxC paradigm, on the other hand, enables the bundle providers with a way to specify in the bundle contracts the necessary authorizations. The framework can then collect these authorizations from the bundles and incorporate them in the SxC policy on demand.

Service usage is more tricky though. Again, the necessary authorizations for the service usage (more precisely, GET permissions for service retrieval) can be delivered within bundle contracts and incorporated into the policy file of the system. But the invocations of the methods within a service, once the necessary reference is obtained, are not guarded by the permission check, and usually the security checks are placed directly within the service code, thus mixing the security logic with the execution logic.

Another solution, that is traditional for mobile Java-based component systems, could be to ask Alice each time a specific permission is needed. But Alice is not the owner of the bundles to make such decisions, nor is she interested to do so.

Let us consider a more complex scenario now.

Example 3 *Alice wants to install the Sims add-on from the thesims.ea.com provider. This add-on is packaged into the bundle 'com.ea.thesims.FarmerSimsBundle' and it will provide an integration of the Happy Farm account with her the Sims account. The functional requirement of the thesims.ea.com provider follows: "The bundle 'com.ea.thesims.FarmerSimsBundle' can be installed if and only if the Facebook bundle is available on the platform and provides the service 'com.facebook.FarmerService' ".*

The requirement in Example 3 means that 'com.ea.thesims.FarmerSimsBundle' can be installed only if the service 'com.facebook.FarmerService' is already provided on the platform. This requirement prevents the denial of service by the Sims bundle. The bundles are running on top of a single JVM, thus the denial of service attack can cause a restart of the whole system [22].

This functional requirement is, in fact, unsupported by the current OSGi specification. Requirements/capabilities model cannot provide guarantees on the provided services (except that their definition exists on the platform). We will further demonstrate how this kind of policy can be enforced by the Security-by-Contract mechanisms, and how the requirements from the running example in Section F.2.1 can be enforced in a flexible and efficient manner.

Further we will refer to thesims.ea.com provider as $EA.com$ and 'com.ea.thesims.FarmerSimsBundle' as C bundle.

F.4 The SxC solution

F.4.1 The OSGi Platform Formal Model

We start with the formal model of the OSGi platform. The entities on this platform are bundles and services, but the model also takes into account the lifecycle of bundles, as it is an explicit part of the OSGi platform.

Let $\Delta_{\mathcal{B}}$ be a domain of symbolic bundle names, $\Delta_{\mathcal{S}}$ be a domain of symbolic service names and $\Delta_{\mathcal{P}}$ be a symbolic domain of package names. Let also $\Delta_{\mathcal{V}}$ be a standard domain of bundle and package versions, $\Delta_{\mathcal{L}}$ be a domain of location strings for the bundles and Δ_{Sign} be a domain of bundle signers names. We also define a set of local permissions requested by each bundle in its permissions file by $\mathsf{Permissions}_{\mathsf{bundle}}$, where each single permission perm is a pair $\langle Target, Action\rangle$. Packages are identified in the model by their fully-qualified symbolic names and (optionally) versions, a package $P \in \Delta_{\mathcal{P}} \times \Delta_{\mathcal{V}}$. If a package version is omitted, a default value is expected.

Definition F.4.1 (Bundle) *A bundle B is a tuple* $\langle \mathsf{name_B}, \mathsf{state(B)}, \mathsf{exports_B}, \mathsf{imports_B}, \mathsf{version_B}, \mathsf{Permissions_B}, \mathsf{Location_B}, \mathsf{Signer_B}\rangle$*, where* $\mathsf{name_B} \in \Delta_{\mathcal{B}}$ *is the symbolic name of the bundle,* $state(B) \in \{\mathsf{Installed}, \mathsf{Resolved}, \mathsf{Starting}, \mathsf{Active}, \mathsf{Stopping}\}$ *is the current state in which the bundle resides at the moment,* $\mathsf{exports_B} \subseteq \Delta_{\mathcal{P}} \times \Delta_{\mathcal{V}}$ *and* $\mathsf{imports_B} \subseteq \Delta_{\mathcal{P}} \times \Delta_{\mathcal{V}}$ *are the sets of packages exported and imported by B correspondingly,* $\mathsf{version_B} \subseteq \Delta_{\mathcal{V}}$ *is the bundle version,* $\mathsf{Permissions_B}$ *is the set of local permissions of bundle B,* $\mathsf{Location_B}$ *is the download location and* $\mathsf{Signer_B}$ *is the signer (the provider) of the bundle B.*

Uninstalled state is not considered, as the bundle is not functioning in this state and cannot be transferred to other states, so this state is equivalent to deletion of a bundle.

We define an *"is defined in"* relation $\diamond$ for packages, bundles and service interfaces. Let B be a bundle, S be a service interface and P be a package. We will denote by $P \diamond B$ the fact that P is a package defined in the bundle B and by $S \diamond P$ the fact that the service interface S is defined in the package P. For defining locations of bundles we define a $\vdash$ relation (*"comes*

from"), we will denote as $L \vdash B$ the fact that bundle B comes from location $L \in \Delta_{\mathcal{L}}$. Also for locations we can define a notion of location inclusion, we will denote as $L_1 \subseteq L_2$ if the string location L_1 includes the string location L_2 as a prefix (without wildcards).

Example 4 *Let us consider the bundle A discussed in Examples 1–2. This bundle can be represented as a tuple $\langle'eu.securechange.homes.gateway.impl', state(A), \mathsf{exports_A}, \mathsf{imports_A}, \mathsf{version_A}, \mathsf{Permissions_A}, \mathsf{Location_A}, \mathsf{Signer_A}\rangle$, where $state(A)$ depends on the current state, for example, we can assume that this bundle was resolved successfully, as it does not include any external dependencies, and currently $state(A)$ =* Active*.*
$\mathsf{exports_A}$ *=* $\{('eu.securechange.homes.gateway.FeedBundlePackage', 0.0.0)\}$*,*
$\mathsf{imports_A}$ *=* $\emptyset$*,*
$\mathsf{version_A}$ *= 1.0,*
$\mathsf{Location_A}$ *='https://securechange.eu/bundle',*
$\mathsf{Signer_A}$ *='securechange.eu'. Relevant permissions are listed in Example 2, so we do not repeat them.*

Definition F.4.2 (OSGi Platform) *The platform Θ is a tuple $\langle \mathcal{B}, \mathcal{S}, \mathcal{R}\rangle$ where $\mathcal{B}$ is a set of bundles on the platform, $\mathcal{S} \subseteq \Delta_{\mathcal{S}}$ is a set of services on the platform, and $\mathcal{R} \subseteq \mathcal{B} \times \mathcal{S}$ is a service provision relation such that for each service $S \in \mathcal{S}$ there exists at least one bundle $B_1 \in \mathcal{B}$ such that the pair $(B_1, S) \in \mathcal{R}$, $state(B_1) \in \{\mathsf{Active}, \mathsf{Starting}, \mathsf{Stopping}\}$, and there exists bundle $B_2 \in \mathcal{B}$ such that $state(B_2) \in \{\mathsf{Active}, \mathsf{Starting}, \mathsf{Stopping}\}$ and $S \diamond P$ such that $P \diamond B_2$ and $P \in \mathsf{exports_{B_2}}$, and $B_1 = B_2$ or $P \in \mathsf{imports_{B_1}}$.*

Thus, in the model we consider that a service can be provided by a bundle in an appropriate state, and there should exist a bundle (possibly the same one) in an appropriate state that exports a package containing the definition of this service. We will denote the fact that bundle B_1 can provide service S as $S@B_1$.

We want to ensure that bundles interact on the platform in compliance with the pre-defined security policies set by the bundle owners. Thus we start with the definition of bundle interaction. Informally, two bundles interact if one of them imports an exported package from another, or consumes a service provided by the other bundle.

Definition F.4.3 (Bundle Interaction) *Let $B_1, B_2 \in \mathcal{B}$. We say that B_1 interacts with B_2, denoted $B_1 \bowtie B_2$ if at least one of the following conditions is satisfied:*
• *Exists $P \in \mathsf{exports_{B_1}} \cap \mathsf{imports_{B_2}}$ such that $P \diamond B_1$ – there exists a package exported by B_1 and imported by B_2;*
• *Exists a local permission* perm= $\langle GET, S\rangle \in \mathsf{Permissions_{B_2}}$ *where S is a service of bundle B_1 ($S@B_1$) – there exists a local permission of bundle B_2 to get a service reference from bundle B_1.*

This definition captures *a potential direct control flow among bundles*.

Functionality guarantees on the OSGi platform can vary. An interesting scenario was discussed in Example 3, where a bundle wants to have some functional requirements fulfilled prior to be loaded. The capabilities approach currently explored on the OSGi platforms is purely static and declarative. We want to enhance it with the dynamics of evolving platforms and with the guarantees given by the framework itself rather than by (potentially untrusted) bundles.

In Example 3, the C bundle wants to be installed on the platform only if a specific service is already provided there. While the presence of the service interface definition can be (limitedly) ensured by the capabilities approach, only the Framework (the platform in our model) itself can assure that the service is indeed provided, or a certain bundle is in a desired state, or that a

competitor's bundle is not installed at all. If later, when the bundle will already be installed, the platform will evolve such that the desired service will be unregistered or an undesired bundle appear, the bundle can be notified about it through the event system and take the actions it needs to protect itself (be removed, stop, notify the provider, etc).

There is also another interesting problem that can be considered. The primary purpose of the requirement-capabilities model is to provide an explicit assertion about the environment before a bundle becomes active and its code starts to run. This prevents bundles that cannot run because they are not suitable for a given environment from becoming active, or even installed, when this header is used by a management system. The SxC framework can become the management system that will assure bundles they will never enter even the Installed state on a platform that is not suitable for them.

F.4.2 Contracts and Policy

The claim of a bundle (sufficient to cover the security and functionality issues discussed above) can be easily extracted from the bundle's manifest file and permissions file. Thus the ClaimExtractor component duties will be to extract this information. The policy of a bundle is a new component specified in the contract that requires a permission notation and a notation for functional requirements.

For contracts notation we will use the wildcard symbol $\star$.

Definition F.4.4 *Let B be a bundle. The* Contract_B *is a tuple* $\langle \text{sec.rules}_B, \text{func.rules}_B \rangle$*, such that:*
– sec.rules_B *is a set of permissions of the form* $\langle Action, Target, Authorized\ entity \rangle$*, that specifies the security policy on the usage of* B*'s packages and services, where*
- $Action \in \{IMPORT, GET\}$;
- $Target \in \bigcup_{P \diamond B} P \cup \bigcup_{S@B} S$;
- $Authorized\ entity \in (\Delta_{\mathcal{B}} \times (\Delta_{\mathcal{V}} \cup \{\star\})) \bigcup \Delta_{\mathcal{L}} \bigcup \Delta_{Sign}$;

– func.rules_B *is a set of functional requirements of* B *of the form* $\langle Desired\ state, Flag, Target, State \rangle$*, that specifies the requests of the bundle for functionality available on the platform, where*
- $Desired\ state \in \{Present, Not\ present\}$;
- $Flag \in \{Bundle, Package, Service\}$;
- $Target$ *and* $State$ *differ in the following fashion:*
  - $Flag = Bundle$*, then* $Target \in \Delta_{\mathcal{B}} \times (\Delta_{\mathcal{V}} \cup \{\star\})$ *and* $State \in \{$Installed, Resolved, Starting, Active, Stopping$\}$;
  - $Flag = Package$*, then* $Target \in \Delta_{\mathcal{P}} \times (\Delta_{\mathcal{V}} \cup \{\star\})$ *and* $State \in \{Present, Exported\}$;
  - $Flag = Service$*, then* $Target \in \Delta_{\mathcal{S}}$ *and* $State \in \{Present, Provided\}$;

Using the above notations bundles can express various security and functional requirements on other bundles on the platform. Bundles can be installed on the platform if and only if all their security and functional requirements are satisfied and their behavior is compliant with the policies of all other bundles on the platform.

We propose the bundle contract to be delivered within its manifest file by using the possibility to define new manifest file headers in the common header syntax. The newly defined headers processed by the SxC manifest file parser (the ClaimExtractor), are **sxc-secrules** and **sxc-funcrules**, they are described in Table F.2.

The security policy of the platform is defined as follows.

Definition F.4.5 (Security Policy of the Platform) *For a platform* Θ *its security policy* $\text{Policy}_\Theta = \bigcup_{B \in \mathcal{B}} \text{sec.rules}_B$

Header	Contents
sxc-secrules	Contains the bundle's sec.rules separated by comma sign (elements of permissions are separated by semi sign)
sxc-funcrules	Contains the bundle's func.rules separated by comma sign (elements of requirements are separated by semi sign)

Table F.2: The SxC headers of the manifest file

Case	Check
B imports a package P of A	$\langle IMPORT, P, B\rangle \in \text{sec.rules}_A$
B gets a service S registered by A	$\langle GET, S, B\rangle \in \text{sec.rules}_A$
B requires a bundle A to be present on the platform in Active state: $\langle Present, Bundle, A, \text{Active}\rangle \in \text{func.rules}_B$	exists bundle $A \in \mathcal{B}$ such that $state(A)$=Active
B requires a package P to be exported on the platform: $\langle Present, Package, P, Exported\rangle \in \text{func.rules}_B$	exists bundle $A \in \mathcal{B}$ such that $P \diamond A$ and $P \in \text{exports}_A$
B requires a service S not to be provided on the platform: $\langle Not\ present, Service, S, Provided\rangle$	For all bundles $A \in \mathcal{B}$ there is no $(A, S) \in \mathcal{R}$

Table F.3: Checks to be executed by the PolicyChecker

Example 5 *Let us consider the running example from Section F.2.1. The bundles A, B and F are installed on the platform. The security policy* Policy_Θ *of the Alice's platform equals to* $\{\text{sec.rules}_A, \text{sec.rules}_B, \text{sec.rules}_F\}$.

The contracts of the bundles can be derived from the requirements listed in Table F.1. Thus, $\text{sec.rules}_F = \{\emptyset\}$. $\text{sec.rules}_A = \{\langle GET,' eu.securechange.homes.gateway.FeedBundleService',$ $'securechange.eu'\rangle\}$.

$\text{sec.rules}_B = \{\langle GET,' com.example.FeedService',' example.com'\rangle, \langle IMPORT,' com.exampleFeed-ServicePackage',' example.com'\rangle, \langle GET,' com.example.FeedHappyFarmerService',' example.com'\rangle,$ $\langle GET,' com.example.FeedHappyFarmerService',' facebook.com'\rangle\}$

For a platform Θ its *functional state* is at any given moment of time defined by the platform itself: the installed bundles and provided services.

F.4.3 The SxC Checks

Definition F.4.6 *Let Θ be an OSGi platform and B is a loaded bundle. We say Θ can host B securely iff the following conditions are satisfied:*

- ***Stable Security**. For all bundles $A \in \mathcal{B}$ if $A \bowtie B$ then a corresponding permission for B or* Location_B *or* Signer_B *exists in* sec.rules_A*, and if $B \bowtie A$ then a corresponding permission for A or* Location_A *or* Signer_A *exists in* sec.rules_B*.*
- ***Stable Functionality**. All functional requirements described in* func.rules_B *are satisfied by Θ.*

We note that the stable functionality property may not hold for some other bundle A immediately after the installation of bundle B on the platform. However, as we have discussed above, A will be notified about the situation and can take appropriate actions.

Table F.3 lists some of the checks that are to be executed by the PolicyChecker in each case.

Case	Source of Information
B imports package P of A	The manifest file of B, the permissions file of B
B gets service S registered by A	The permissions file of B
B requires bundle A to be present on the platform in Active state	The platform maintains the bundles and their states
B requires package P to be exported on the platform	The platform identifies the existing packages
B requires service S not to be provided on the platform	The service registry maintains the provided services

Table F.4: Sources of information for the SxC Framework

It can be demonstrated (proof by cases) that if the necessary checks for a new bundle B are performed by the SxC framework, then the platform can host B securely.

Table F.4 contains the sources of information for the components. We can note here that the permissions file could be a weak source of information, as it may only require AllPermission, letting the system to define the upper bound of permissions for the bundle. This problem can be solved by awareness of the developers that their bundles will be rejected if the required permissions will be too demanding.

Conflicts among Bundles

Bundle interactions and functional requirements can give rise to conflict situations.

Example 6 *Let us consider again the running example. Let the C (The Sims) bundle from $EA.com$ provider be installed and running. Then Alice might decide that she wastes too much time playing Happy Farm. Alice wants to remove the Facebook bundle from the platform. But the C bundle has specified the service from this bundle running on the platform as its functional requirement. If the $EA.com$ provider was careless, the removal of the Facebook bundle can cause the C bundle to malfunction or even the framework to restart.*

The choice in this situation is the following: either to remove the Facebook bundle and expect that C bundle can crash, or to forbid the removal of the Facebook bundle. In both cases the consequences are unpredictable. The situation could be improved if the Conflict Resolution component would be provided within the SxC framework. The Conflict Resolution component can take decisions in complex situations depending on the contents of the framework policy file (some bundles may have more permissions than other do), and the system policy specified for such situations.

F.4.4 Threats to Validity

Let us now discuss under which assumptions the SxC model works.

Authentication threats and bundle impersonalization are not treated in the model, as we rely on the correct platform implementation with reliable authentication mechanisms. Thus we identify entities in the model by their fully-qualified symbolic names, considering the names of providers, bundles, packages and services to be unique.

Several features of the latest OSGi specification are not discussed in the current paper. These are, for instance, the remote service provision and service factories, fragment or extension bundles and library bundles (ones that cannot be activated). Some of these features can be introduced easily in the model (like library bundles), and some of them are not related directly to

the running example scenarios. We also did not consider delegation of permissions, assuming each bundle provider acts on her own behalf.

The OSGi platform, besides the bundle lifecycles, maintains also the framework lifecycle, which is not a part of our platform model. We do not treat the framework restarts in the model, as they are not related to the considered attacks.

Conclusions and Possible Extensions

In this chapter we have presented the Security-by-Contract proposal for the OSGi platforms that are the main target of the HOMES case study. We gave an overview of the OSGi Framework and then described how the security and functionality checks already present on the OSGi platforms can be benefited by the SxC approach.

The chapter contains the following contributions:

- A formal model of an OSGi platform;
- Discussion on possible security and functionality threats;
- A contracts notation for bundles, namely, the notations for security and functional requirements of the bundles;
- A sketch of the security and functionality checks that need to be carried out by the SxC framework.

The main benefits that the SxC approach in the current form can bring to the OSGi platforms are:

• **Security** The bundle providers can specify in an easy fashion the authorizations for access to their bundles, packages and services. The policies can be updated easily and the update does not require an interaction from the platform owner or a set of permissions to access the Framework policy file.

• **Functionality** The bundle providers have more powerful tool for expressing their functional requirements than just requirement/capability model. The contracts can express requirements on the current state of the platform (including requirements on the states of the present bundles or certain services provision, or absence of the competitor's resources).

There are, of course, interesting research problems that can be investigated further. First of all, the current definition of the bundle interactions is very restricted. The notion of bundle interactions can be extended to include actual service method invocations or transitive service calls and actual information flow specification. To the best of our knowledge, there are currently no known investigations of information flow among bundles, probably because of the complicated service provision model.

For the functionality issues, it would be interesting to extend the current functional requirements notation to include more interesting and meaningful functional dependencies descriptions.

Finally, the Conflict Resolution component can be proposed, that can incorporate more complex logic for solving conflicts among bundles. This logic can include analysis of consequences of the decision made (if a bundle will crash after its functional dependency is gone or the developers have considered this possibility and there is a safety net), policy on when and how to prompt the user, etc.