Project N°: FP6-015905

Project Acronym: MOBIUS

Project Title: Mobility, Ubiquity and Security

Instrument: Integrated Project

Priority 2: Information Society Technologies

Future and Emerging Technologies

Deliverable D5.2

Evaluation of type based and logic based verification techniques

Due date of deliverable: 2007-03-31 (T0+18)

Actual submission date:

Start date of the project: 1 September 2005 Duration: 48 months

Organisation name of lead contractor for this deliverable: FT

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Contributions

This document was written by Richard Bubel (CTH) Pierre Crégut (FT), Jacek Chrzaszcz (WU), Keqin Li (SAP), Wojciech Mostowski (RU), Mariela Pavlova (TL) Erik Poll (RU), German Puebla (UPM), Aleksy Schubert (WU), and Tadeusz Sznuk (WU).

Site	Contributed to Chapter
FT	2.4
SAP	3.3
TL	1, 2, 3.1, 2.2, 4
RUN	1, 2,2.1, 4
UPM	3.2
CTH	2.3
WU	2.6,2.1

Executive Summary:
Evaluation of type based and logical verification technology

This document describes the evaluation results of the MOBIUS verification techniques performed in tasks T 5.2 and T 5.3 in workpackage WP 5. Here task T5.2 focused on the evaluation of type based techniques developed in workpackage WP2, and task T5.3 on the logic based verification techniques developed in WP 3.

The techniques which have a prototype are evaluated upon a relevant set of case studies defined in task T5.1. Most case studies concern realistic problems from the field of MIDP mobile phone application software, but we also looked at SAP web-applications. The evaluation considers properties worked out in package WP 1 which mainly refer to safe information flow release and resource security policies.

The objective of the evaluation phase is to consider the application of the verification techniques in realistic scenarios. More particularly, we look at how the techniques scale up, what is their language and property coverage, what is the degree of automation, is the time for verification reasonable (including both the annotation if such is necessary and verification process), what is the ratio between precision and complexity.

Contents

1 Introduction 6

2 Evaluation of program verification techniques based on program logic 8
- 2.1 The Quiz Demonstrator Case Study . . . 9
  - 2.1.1 Quiz Game Navigation Graph . . . 9
  - 2.1.2 JML Annotations . . . 11
  - 2.1.3 Verification . . . 12
  - 2.1.4 Conclusions . . . 12
- 2.2 The Telnet client case study . . . 13
  - 2.2.1 The telnet client . . . 13
  - 2.2.2 Specifying the Telnet client in JML . . . 13
  - 2.2.3 Conclusion . . . 15
- 2.3 Ensuring secure information flow . . . 16
  - 2.3.1 The SecureSMS application . . . 16
  - 2.3.2 Secure information-flow properties . . . 16
  - 2.3.3 Verification of secure information-flow properties . . . 16
- 2.4 Improving navigation graphs with tests and logic based verification techniques . . . 18
  - 2.4.1 Navigation graph extraction . . . 18
  - 2.4.2 Refining navigation graphs with testing . . . 18
  - 2.4.3 Proving restrictions with invariant propagation . . . 19
  - 2.4.4 Benchmarks . . . 19
  - 2.4.5 Limitations and further work . . . 21
- 2.5 Input validation for a web application . . . 22
  - 2.5.1 Case Study . . . 22
  - 2.5.2 Experimentation . . . 22
  - 2.5.3 Perspective . . . 24
- 2.6 Case Study with BML tools . . . 24
  - 2.6.1 The Bill Case Study . . . 24
  - 2.6.2 The Mobius Quiz and BML tools . . . 26

3 Evaluation of type systems 27
- 3.1 Prototype for an explicit information flow . . . 27
  - 3.1.1 Limitations . . . 29
  - 3.1.2 Case studies . . . 29
  - 3.1.3 Conclusion . . . 30
- 3.2 Experiments with the COSTA System . . . 31
  - 3.2.1 Upper Bounds for Java Bytecode Programs . . . 31
  - 3.2.2 Peak Heap Consumption for Java Bytecode . . . 32
  - 3.2.3 Termination Analysis for Java bytecode . . . 32
- 3.3 Implicit flows in nonmalicious code . . . 33

3.3.1 Introduction 33
3.3.2 Information-leakage patterns 33
3.3.3 Logging API and encryption API 34
3.3.4 Security analysis for logging 34

4 Conclusion 36

Chapter 1

Introduction

The current document reports on the evaluation phase over the MOBIUS technology. The main objective of this phase is to see how mature the developed MOBIUS techniques are, whether they scale up, whether they provide a sufficient coverage for the relevant security requirements and whether they are usable in a more industrial scenario. The evaluation also serves to identify the directions to be investigated for improving the MOBIUS technology.

MOBIUS has as a main goal to build a PCC infrastructure to address important security issues concerning the execution of untrusted code on a host system. Those problems are concerned with the resource consumption of computational resources, the consumption of external billable resources and information flow, as well as certain functional properties, notably the notion of midlet navigation graph, introduced in deliverables D1.1 and D5.1, as a means to give a high-level specification of (constraints on) the functional behaviour.

The techniques developed for treating these issues can be divided in two major groups: type-based techniques and logic-based verification techniques.

Type systems are program static analysis algorithms dedicated to a particular problem. Several type based technologies have been produced by the MOBIUS academic partners which address important software security issues related to the preservation of information confidentiality, resource consumption and access control on billable resources. An important body of theoretical research concerning type systems for constraint resource consumption as well as information flow has been produced in the scope of the MOBIUS workpackage WP2. A main feature of type systems is that those provide a very good (if not full) level of automation and from end user perspective they can be very interesting. On the other hand, because of the level of automation they are usually not precise. The evaluation of type systems therefore should take into account whether they provide a good compromise between precision and automation. For this, we have chosen several pertinent example applications and have checked them with prototype tools. Chapter 3 is dedicated on the description of those experiments.

Differently from the type system approach, logic-based program verification techniques provide a general framework for expressing and verifying a large spectrum of program properties. The concrete objectives of the logic based verification evaluation are to assess the specification language expressiveness (whether it is sufficiently rich to express the needed security properties) and whether the specification and verification process can be carried out successfully (even by non experts). The evaluation of logic-based verification techniques, described in Chapter 2 includes case studies in annotating source code of midlets with JML to express relevant security properties (midlet navigation graphs, explicit information flow, and usage of billable resources), experiments in generating BML annotations and certificates from this, and case studies tackling more ambitious forms of information flow. The first case studies required the annotation of parts of the J2ME MIDP API.

The project MOBIUS has chosen as case study targets several software technologies for which the security problems studied in the project are of major importance. The first one is the MIDP framework which is the Java Sun technology for mobile phone devices. As this framework is open and at the same time features

both connectivity and management of sensitive data, the detection of undesirable information flow or the abuse of billable resources in the application code becomes of major importance from end user or mobile operator perspective. The other platform used for case study in MOBIUS is the SAP platform which also features security concerns with information flow and SQL injection attacks. Early work of the MOBIUS project in the scope of workpackages WP 1 and WP 5 have defined the application and framework security requirements relevant to these frameworks as well as application software which features real life security problems. These requirements and applications are the basis for the evaluation phase described here which addresses both MOBIUS type and logic based verification techniques.

The results of the performed case studies show that the MOBIUS technology is feasible as various theoretical results ended up as prototype implementations. Those implementations were particularly customized to address the security problems of the studied industrial technologies (e.g. providing particular models of the frameworks' API). These prototypes proved to scale up as they were applied to real life case studies with realistic sizes and common or complex program features. The practical interest of the MOBIUS technology is also obvious as the addressed security problems align with the needs reported by the MOBIUS industrial partners in early deliverables (see D1.1). It is interesting to note that some case studies were carried out successfully by non experts in program verification and formal methods. This shows that the MOBIUS technology replies adequately to industrial reality and constraints where such expertise is rare.

In the following, we proceed to the description of the case studies performed in the project. Chapter 2 is focused on the evaluation of logic based verification techniques. Various of the academic and industrial partners were working on this task. As a result, we have various case studies over different MIDP or SAP applications which were verified against several relevant security policies. Chapter 3 is related to the evaluation of the techniques for type based verification. Here, we provide an account on several type system prototypes tailored to the verification of different security policies related to information flow and resource consumption.

Chapter 2

Evaluation of program verification techniques based on program logic

This chapter decribed the work in Task T5.3 on the evaluation of logic-based verification technniques developed in WP3.

Sections 2.1 and 2.2 describe two case studies in expressing security properties for MIDP midlets using JML. Section 2.1 describes work on the MOBIUS Quiz Game, the MOBIUS demonstrator developed by TLS. This is a smallish midlet, but one coded in a style that comes from industry practice, (i.e. not optimised in any way to make it amenable to verification), except maybe that the code was not deliberately obfuscated. For the MOBIUS Quiz Game conformance to its midlet naviation graph was proved, which includes constraints on some billable events, namely the number of SMS sent. Section 2.2 describes work on a smaller Telnet client, for which again conformance to its midlet naviation graph as proved, but also some explicit information flow property (to constrain the IP address the application connects to).

For these case studies JML specification of parts of the J2ME MIDP API were needed, which were developed jointly by RU, TL, and WU. The main tool support used in developing the JML annotations was ESC/Java2, as this is easiest to use, and gives the quickest feedback, of the verification tools in the MOBIUS Program Verification Environment (PVE). For the MOBIUS Quiz Game we then carried out experiments in generating BML byte code annoations from the JML source code annotations, using the tool support for this in the PVE, as would be needed for actual certificate generation. The focus of the case studies was checking the feasibility of annotating midlets, and the MIDP API, to express interesting security properties, in such a way that they would be useable in a PCC scenario (which imposes additional constraints).

The work on the Quiz Game was carried out by an experienced researcher, but the work on Telnet client was carried out by a complete novice, so this gives some indication that this work can be done by people with prior experience with formal specification. Although for the smaller Telnet client this turned out to be feasible, we doubt if for the larger MOBIUS Quiz Game this would have been.

Section 2.3 considers the verification of a variety of information flow properties for an existing open source SecureSMS application. This tackles more ambitious information flow properties than the simple explicit information flow considered for the Telnet client in Section 2.2: it considers non-interference, and two variants of non-interference extended with some form of declassification, namely delimited information release and conditional information release.

Section 2.4 discusses work on the extraction of navigation graphs from bytecode (published as [9]) and subsequent refining of these navigation graphs by testing.

Section 2.5 discusses a case study using JML to trak input validation in a SAP web application, in the same manner as (explicit) information flow is handled in Section 2.2.

Section 2.6 describes the use of the entire tool chain, from JML to BML to Coq proof obligations and ulitmately certificates, for a small toy example, the Bill class.

2.1 The Quiz Demonstrator Case Study

The Mobius Quiz game was developed by TLS [17] as a midlet demonstrating some of the security concerns in midlets. It also provides a realistic midlet written according to industry conventions: it has not been simplified or optimised for the purposes of formal analysis in any way (except maybe in the fact that the code has not been obfuscated). As such, it provides an interesting measure to see how far our proposed technologies can cope with such code.

Primary goal in the work on the Mobius Quiz game was to see if and how the game's behaviour described as midlet navigation graph [9] could be faithfully captured in JML specifications, and in such a way that formal verification was possible. An underlying issue here was finding out the size of the API the midlet relies on, and the way the midlet interacts with the platform through this API. In practice, the need for detailed specifications of large APIs may well be a show-stopper in applying formal verification, and hence in using PCC.

A subsequent goal was seeing if such a specification and verification would lend itself to PCC. If specifications become heavily entwined with program code, it becomes hard for the code consumer to check if the policy the code adheres to is the intended policy. Ideally, there should be a clear separation between

(i) specifications expressing the intended policy,

(ii) specifications expressing assumptions about the MIDP platform,

(iii) any additional annotations needed for the purposes of verification (such as invariants, 'internal' pre- and postconditions), i.e. additional annotations that are needed to prove that the midlet conforms to (i) under the assumption the API behaves according to (ii).

Also, ideally specifications of the platform should not be specific to the midlet, but the close interaction between a midlet and the MIDP platform (e.g. through call-backs and listeners) complicates this.

Another goal was seeing if the concurrency patterns in the Quiz Game would be amenable to formal analysis using techniques developed in WP3.3. This last part would be informal, since tool support for these techniques, insofar it exists, is still not capable of coping with anything as complicated as the Quiz Game midlet.

To formally express and check the conformance of the Quiz Game to its the midlet navigation graph the following steps needed to be taken:

- defining a detailed navigation graph for the midlet,
- devising a way of translating the graph into corresponding JML annotations,
- engineering the necessary JML annotations for the J2ME API and the midlet, and
- verifying these specifications.

For the verification we used the ESC/Java2 which is part of the MOBIUS Verification Environment tool suite. The details of our effort are reported in [18], here we give a brief overview of the work.

2.1.1 Quiz Game Navigation Graph

Figure 2.1 presents the detailed navigation graph for our midlet. The graph uses UML notation. The states of the graph represent the possible screens of the midlet, the arrows represent the screen transitions labelled by user action events (button presses) or internal application events (errors). The transitions are also augmented with security-sensitive operations that may possibly take place during a given transition. These operations (HTTP – establishing an HTTP connection, SMS – sending out an SMS) are included in the transition guards (in [] brackets). A star attached to an operation denotes the operation's failure.

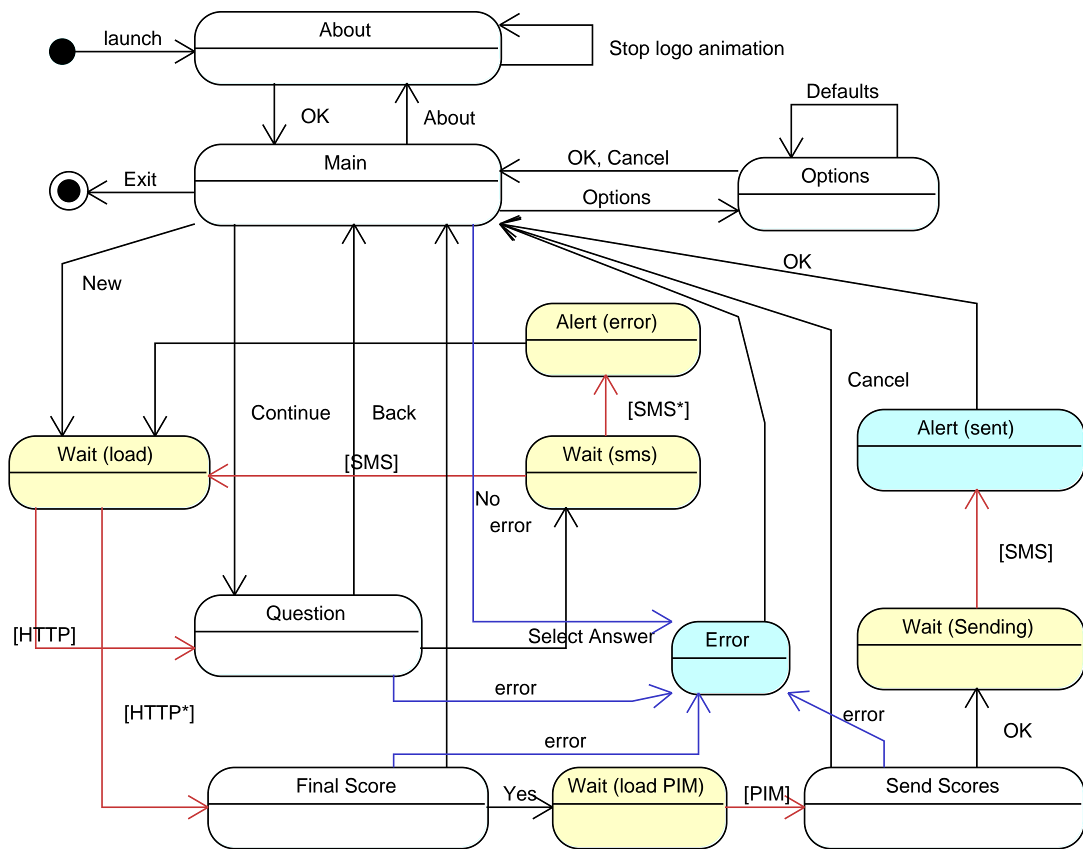

Figure 2.1: The MOBIUS demonstrator quiz game navigation graph

2.1.2 JML Annotations

To annotate and verify the midlet w.r.t. its navigation graph two steps were necessary:

- The first step was to provide as simple as possible formal MIDP API specification. By simplicity here we understand a minimal set of generic (i.e. application independent) annotations that would allow as to specify and reason about different MIDP widgets displayed on the screen, user and system actions, and invocation of security sensitive operations of the MIDP API. All other aspects of midlet functional behavior were not considered and specifications of these were avoided whenever possible. The simplicity of the API specification has a direct consequence for feasibility and performance level of the verification.
- The second step was to provide midlet specific JML annotations to reflect the actual navigation graph.

An example of part of the MIDP API specification is the following:

```
public class Display –

   private /*@ spec˙public @*/ Displayable current;

   //@ public static ghost boolean displayUpdated;

   //@ requires nextDisplayable != null;
   //@ ensures current == nextDisplayable;
   //@ ensures Display.displayUpdated;
   //@ assignable current, preAlert, Display.displayUpdated;
   public void setCurrent(Displayable nextDisplayable);
```

The method setCurrent simply updates the midlet display with a new Displayable object. It also sets a global flag stating that the display is in a stable (just updated) state. We use this flag in our midlet specifications to keep track of when exactly our conditions related to navigation graph should hold, as in these two examples of the midlet code:

```
 public class Options –
   //@ requires cmd == cmd˙ok —— cmd == cmd˙cancel —— cmd == cmd˙defaults;
   //@ requires cmd.displayable == d;
   //@ requires midlet.display.current == d && d == form;
   //@ ensures Display.displayUpdated;
   //@ ensures cmd == cmd˙ok ==¿ midlet.display.current == midlet.mainMenu.list;
   //@ ensures cmd == cmd˙cancel ==¿ midlet.display.current == midlet.mainMenu.list;
   //@ ensures cmd == cmd˙defaults ==¿ midlet.display.current == form;
   //@ ensures Connector.openCount == “old(Connector.openCount);
   //@ ensures MessageConnection.smsSent == “old(MessageConnection.smsSent);
   //@ assignable maxQNum, baseURL;
   //@ assignable midlet.display.current, Display.displayUpdated;
   public synchronized void commandAction(Command cmd, Displayable d) –
   if (cmd == cmd˙cancel) –
     midlet.getDisplay().setCurrent(midlet.getMainMenu().getDisplayable());
    ˝ else if (cmd == cmd˙ok) –
     updateValues();
     midlet.getDisplay().setCurrent(midlet.getMainMenu().getDisplayable());
    ˝ else if (cmd == cmd˙defaults) –
     baseURLField.setString(Consts.WWW˙URL);
     maxQNumField.setString(Consts.DEFAULT˙MAXQ);
```

```
    ″
   ″...
″

 public class QuizMidlet extends MIDlet –

   /*@ invariant Display.displayUpdated ==¿
       (display.current == mainMenu.list ——
        display.current == Options.opts.form —— ...); @*/
   ...
 ″
```

The specification of the commandAction method reflects the graph transitions from the Options screen, as given in Figure 2.1. It specifies how the screen changes after different commands (namely cmd˙ok, cmd˙cancel, and cmd˙defaults), and also that no sensitive methods are invoked during any of the transitions (the static references to openCount and smsSent fields). The invariant in the QuizMidlet class defines the whole set of possible midlet screens. The rest of the midlet is specified in a very similar fashion.

2.1.3 Verification

The annotated midlet was verified with the extended static checker for Java, ESC/Java2, which completely ignores concurrency. The use of concurrency in the midlet is limited and is according to a very strict pattern, where a new thread is started in response to some platform event to handle that event, so that the GUI remains responsive. It would seem that this pattern could be handled by the permission accounting ideas considered in WP3.3 [16] as described in Clément Hurlin's forthcoming PhD thesis [13], where the newly started thread get the permission to change the midlet's state which it relinquishes when it dies.

We did not attempt verification of the generated verification conditions for the Quiz game, as our experience with the much smaller Bill class, described in Section 2.6, showed that Coq support for this (in the from of tactics) is still very much underdeveloped.

Still, we have shown that the tool chain for producing Coq verification conditions from JML-annotated midlets works on realistic (albeit still small) midlets. Also, our Umbra editor could cope with the generated BML.

2.1.4 Conclusions

We have fully specified in JML (modulo concurrency) the conformance of the Mobius quiz game demonstrator w.r.t. its midlet navigation graph. All this took a substantial efforts, in the order of several person-months by an experienced researcher. In retrospect, it would have been better to tackle something smaller first, to work out the basic approach of how to specify the API and typical way the midlet interacts with the API.

The level of API specifications needed was less than we anticipated, and developing the API specifications needed for the conformance to the midlet navigation graphs is definitely doable.

The specifications for the midlet are quite involved, as should be clear from the sample specification of the Options class given earlier. Although large parts of these could be produced automatically from the navigation graph, as was done in [19] for the verification of an SSH midlet. There JML annotations were generated from a finite state machine, which was not a midlet navigation graph, but the same principle could be used for midlet navigation graphs. Still, annotating the midlet with the additional annotations for verification to go through would be a lot of work. Moreover, just providing the navigation graph is already a lot of work – note that for a relatively simple application such as the Mobius Quiz Game the detailed navigation graph given in Figure 2.1 is already quite complex.

A potential loophole when it comes to enforcing the policy, as currently expressed by the JML annotations, by means of PCC for this case study is non-termination: if the midlet does not terminate then it could avoid having to meet any requirements expressed as postconditions. E.g., a diverging method might try to send

an infinite number of SMS text messages, even though the method's postcondition specifies that the number of SMS sent will be unchanged in the postconditions. To avoid this, termination of the midlet's methods would have to be proved. This would not exclude the possibility of the midlet to keep on running forever, but it would require that the action the midlet performs in response to some platform event always ends. Again, the use of concurrency complicates this: all threads started in response to a single platform event would have to terminate.

2.2 The Telnet client case study

The telnet client is a small midlet application which provides an interface for connecting via telnet to the URL specified by the user. The telnet client features connectivity which is one of the most vulnerable resources in midlets as it can cost money to the user or can leak user personal information. We have therefore selected this applet as it provides a pertinent security issue. The objective of our study here is aiming to test the logic-based verification approach, which relies on JML as specification language, and to reply to the following questions

- is JML sufficiently expressive to express the relevant security policy?
- is the logic-based technology in MOBIUS efficient in terms of effort and precision? The idea was to see whether the specification and verification process are reasonably complex. If the technology has to be used as part of the development process then it should respect the time and effort constraints which are important in an industrial context.

In the following, we briefly describe the telnet client, then we discuss on the security policy, its encoding in JML and verification with the logic based verification tool, for which we used ESC/Java2 in the MOBIUS PVE. Finally, we provide remarks on the use of JML as part of a PCC certificate.

2.2.1 The telnet client

The behaviour of the telnet client is described by its navigation graph in Fig. 2.2. As we can see from the figure the user first enters a URL name in a text box, then he confirms that he wants to connect to the URL by hitting a button (producing a command event). The application then launches in a separate thread the telnet connection with the selected server. The user can then exchange data with the server.

The security concern here is that the application must establish connections only to the URLs specified by the user. In other words, we have to get sure that any connection that the applet establishes is specified by the user or is not for instance coming from the program code or even from another network connection.

2.2.2 Specifying the Telnet client in JML

For specifying the security property in JML we actually had a preliminary phase of specification mainly to express that the program is safe w.r.t. the Java semantics - typically that there are no null pointer dereferences or that arrays are accessed in their scope. This part of the specification is useful as it allows to verify that the code is free of such ordinary program bugs. The following listing gives an example of such a specification

```
//@ requires resp!= null && resp.length ¿=0;
public byte[] detokenize(byte[] resp) –
  ...
  //@loop˙invariant 0 ¡= i && i ¡= (resp.length + 2);
  for (int i = 0; i ¡ resp.length; i++) –
    ...
  ˝
˝
```

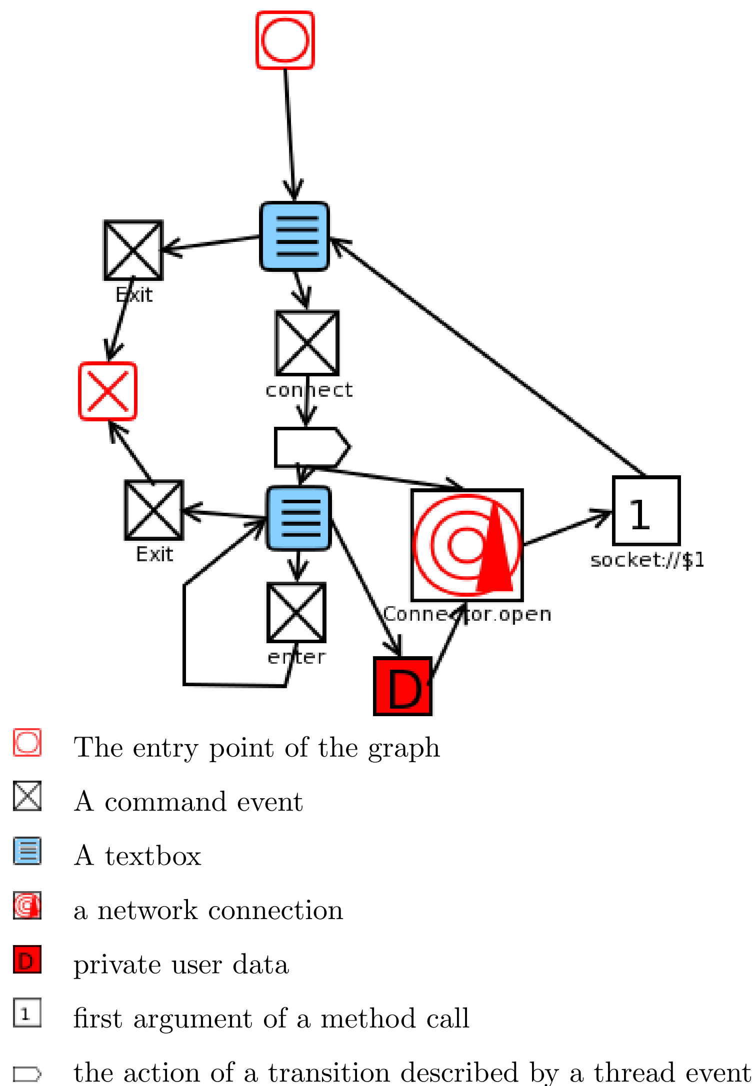

Figure 2.2: KTelnet navigation graph

To express the security property of interest as a JML specification we needed first to customize the specification of several API classes. For this, we used a special kind of JML constructs called ghost variables - variables which are invisible to the program semantics but which express an extra-functional fact. We had to distinguish string values entered by the user. For this we added to the java.lang.String API class the ghost field enteredByUser as follows:

```
public final class String–
  //@ public ghost boolean enteredByUser = false;
  ...
”
```

The enteredByUser ghost field is by default false and is set to true only for Strings which are entered by the user, i.e. which are coming from a TextBox displayed on the screen. To express this property as a JML specification we provide the following specification for the TextBox.getString() method which returns the String entered by the user:

```
public class TextBox extends Screen –
  ...
  /*@
    @ensures “result.enteredByUser == true;
    @*/
  public String getString();
...
  ”
```

Network connections in midlets are done via the API class method java.io.Connector.open(String url). For our particular purpose, it is necessary to specify that connections are made only to urls coming from the user input, in other words the parameter of the open method must come from the user input and therefore we specify the following precondition:

```
//@requires url.enteredByUser == true;
 public static Connection Connector.open(String url) –...”
```

The next listing shows the specification to the method connect in the application code which is responsible to store the server name to which the telnet connection will be established. The method therefore requires that the server name is specified by the user:

```
//@requires DIAGRAM˙STATE == INIT;
//@requires server.enteredByUser == true;
//@ensures serverAddr == server;
//@ensures DIAGRAM˙STATE == SERVERADDR˙SPECIFIED;
public boolean connect(String server)–
  serverAddr = server;
  //@ set DIAGRAM˙STATE = SERVERADDR˙SPECIFIED;
  return true;
”
```

2.2.3 Conclusion

The telnet case study was successfully validated against its respective security policy. This is a promising result as it shows that JML tool can handle realistic security policies.

A potential loophole in the use of this specification for the security policy in a PCC scenario is the use of ghost variables as sketched in the Subsection 2.2.2. This is because ghost variables can be modified in the specification by using the special construct set. There is nothing to prevent the value of such ghost variables being modified by some using the special set construct in some annotation anywhere in the midlet code. A malicious code producer could use this to subvert any security guarantee the JML specifications is meant to provide. For instance, we can set the ghost field enteredByUser of any string value to true even of those Strings which are definitely not entered by the user as in the following example:

```
String url = ”www.pirate.com”;
//@set url.enteredByUser = true;
```

While in the context of software development, the use of ghost variable is completely legal as they express the intention of the developer in the context of untrusted code they should be used with great care. In particular, any ghost variable appearing in a PCC certificate must have a clear semantics in the trusted computing based on the host system. Presumably we would need a trusted checker to ensure the absence of malicious set statements in untrusted code.

2.3 Ensuring secure information flow

We report on a small case study on verification of secure information-flow properties using logic-based methods developed in MOBIUS. The case study is centered around a Java midlet application called SecureSMS [1] which is publicly available under terms of the GPLv2.

2.3.1 The SecureSMS application

Short messages SMS are commonly used by mobile phone owners to communicate with friends, banking, buying tickets, participating at games and more. SecureSMS promises increased privacy and security by encrypting the send messages. In its current stage it supports only symmetric encryption with one key per contact stored in the phones address book.

We extended SecureSMS slightly by adding a basic password protection to its address book which provides us a good example to verify a secure information policy based on conditional information release.

2.3.2 Secure information-flow properties

As examples we selected the following three secure information-flow properties to be treated using logic-based verification:

- non-interference is the strictest information-flow policy prohibiting any confidential information to be leaked to a public observer. Non-interference serves us as information-policy when checking the message display logic: a received message is displayed together with a header composed from information queried from the address book. Using the abstraction based method described in [7] developed in MOBIUS Task 3.5 to ensure that the encryption key is not accidentally leaked and e.g. shown as part of the header.
- delimited information release is a relaxation of non-interference as this policy allows to specify the amount of information that may be released to the public. We prove delimited information release for the password checker, which may only leak if a given password is valid or not.
- conditional declassification allows leakage of information if a certain property is satisfied e.g. a valid password has been provided or similar. In our case study we use conditional declassification as secure information-flow policy for the method getKey of the address book class. The method is only allowed to return the encryption key for the specified contact if the provided password is valid.

Most logic-based formalisation of non-interference properties build on self-composition of programs as described in [10, 6] or build on the insight that information-flow can be modelled in terms of variable dependencies which can be expressed using alternating quantifiers [11, 10]. We used formalisation build on top of the latter approach using quantifier-shift expressions and in addition on an abstraction-based approach [7] which keeps explicitly book of variable dependencies.

2.3.3 Verification of secure information-flow properties

In this section we summarise our experiences when verifying the secure information-flow policies for selected methods of the SecureSMS application.

Delimited information release and conditional declassification

Delimited information release allows to leak a well specified amount of information like for instance the average income of an employee in a company, but not the exact income of a concrete individual employee. This information-flow policy is aligned on the what dimension of the classification given in [20].

[1] http://www.midlet.org/

The logic formalisation is a slight variant of the general non-interference formalisation:

$$\forall\overline{lv}\forall\overline{eh}\exists\overline{rv}\forall\overline{hv}.(\overline{eh} \doteq \overline{escapeHatches} \rightarrow \{\bar{l} \doteq \overline{lv} \wedge \bar{h} \doteq \overline{hv}\}p(l,h);\{\overline{rv} = \overline{lv}\})$$

which declassifies the escape hatch by basically assigning it to a public variable in advance. In our scenario, we ensured that the method checking if the entered password is valid, does only leak one bit of information, namely password is correct resp. incorrect.

Access control is also a typical scenario where a conditional declassification policy is required. In our scenario, the contact specific key needs to be addressed if a message has to be encrypted resp. decrypted. Access to the key requires the knowledge of a passcode granting the access.

Standard non-interference is obviously too strict for this usage scenario as the secret is actually leaked if a valid passcode has been provided. Such a policy is classified as conditional information release, and belongs to the when dimension.

The formalisation is similar to the one given above, but restricts the equation declassifying the escape hatch expression only if certain conditions are satisfied in the initial states. In the tested example this worked again automatically, but for the general case the remarks concerning automation from the previous section hold.

The verification worked automatically for the given problems as the required instantiation of the existential quantifier was relatively easy to find. In slightly more complicated cases, human interaction would be required. To eliminate human interaction as far as possible two proof systems using different techniques have been developed on a theoretical level as part of Task 3.5. In the following, we focus on the approach using explicit book keeping of dependencies and describe our as experiences in Sect. 2.3.3.

Explicit Dependency Tracking and Abstraction

We explored also the method introduced in [7] developed as part of Task 3.5 of MOBIUS. The approach tracks information-flow explicitly by assigning each location a 'ghost' location keeping track of variable dependencies, i.e., which other locations have contributed to the current value of the variable. Together with an abstraction based proving procedure, it is possible to automatically verify secure information-flow policies.

To realise the explicit book-keeping of dependencies within current tools we decided to realise the ghost locations as actual JML ghost fields and to encode locations and location sets as integers resp. bit-sets.

We explain now the encoding idea in a bit more detail: Given program variables l and h, we assigned each of the variables a location id-number 1 (only bit 0 is set) resp. 2 (only bit 1 is set). Further, we declared for each variable a ghost field l˙dep resp. h˙dep of integer type. These ghost fields encode the associated location dependency sets as bit vectors. For instance, if bit number 1 is set, i.e., that the current value of the associated location depends on the initial value of the location with id $2(= 2^{\mathbf{1}})$[2]

When assigning a constant to h its h˙dep is set to 0 erasing all further dependencies, while an assignment h=l; causes the value of h˙dep to be updated to h˙dep = h˙dep — l˙dep Testing if the value of h depends on the initial value can then be done by testing for the associated bit, e.g., dependsOnH = (l˙dep & 1) != 0

For the encoding we used the class DependencyTracker to assign any location a unique id and to keep book about the location dependency for given locations.

We used our simple encoding to successfully verify that the method displaying a received message on the screen and decorating it with additional information from the senders contact details in the address book does not accidentally leak the encryption key.

[2]A bit vector support is not yet sufficiently supported by the used tool, we used modulo and division operations instead. The basic principle remains unchanged.

2.4 Improving navigation graphs with tests and logic based verification techniques

In [9], we presented a crude algorithm to extract a compliant navigation graph from the compiled bytecode of a midlet. The algorithm is based on a points-to analysis and although its implementation can be improved with finer points-to algorithms (context-sensitive or object-sensitive versions), it suffers from intrinsic limitations arising from the fact that it only analyzes the data structures directly involved in the definition of the user interface and not the other elements that may influence the global control flow of the midlet.

Logic based verification techniques are usually more flexible and can describe any kind of invariant verified at any point of the program. On the other hand, this flexibility comes at the expense of much less automation, the context of the proof is also usually more local and it is usually up to the evaluator to describe the expected properties.

Another source of information on the behaviour of the midlet is "exhaustive" testing. This is the kind of testing performed by test houses that try to assess the usability and also whether the midlet is dangerous before it is put on an application shop. Extracted navigation graphs can be used as a guide for this kind of testing and give a good estimation of the coverage of the test campaign.

We propose to use static analysis to build a first approximation of the navigation graph, then to use testing to check which transitions are in fact available and then to use automatic proof techniques to prove that the other transitions are in fact unreachable. A last step (not performed here) would be to modify the code to definitely forbid some of the potentially dangerous transitions that have not been reached by testing but that have not been proved unreachable.

2.4.1 Navigation graph extraction

We used an implementation of the algorithm presented in [9]. It is organized around two applications:

- The first one performs the points-to analysis (using Soot [21]) and the local analysis of paths within event listeners callbacks where tests and relevant events (setting the screen, performing an operation involving security relevant methods) are isolated.
- The second one builds the graph from the results accumulated by the first set of analysis. It also interprets tests to prune the graph.

2.4.2 Refining navigation graphs with testing

The graph obtained from the previous phase can already be used to guide testers. Tests are an essential part of validation process but their target is more the usability of the midlet (a lot of problems arise from font sizes, item layout, color problems, bad image rendering) rather than security. Although they cannot find hidden (potentially dangerous) transitions that only appear under specific conditions, such test campaign try to exhaustively cover the set of transitions of the midlet. The extracted navigation graph provides a good conservative approximation of this set.

The midlet is instrumented so that every call to a method modifying the user interface is logged:

- We have used the AspectJ bytecode weaver to instrument calls to MIDP APIs (creating new graphical objects, changing the display, associating a listener or a command to a displayable).
- We have used a custom instrumenter written on top of BCEL to instrument the beginning of callback methods (there is no direct way to designate the entry point of a method as a pointcut). Weaving in Aspect oriented language is usually done on the caller side (which is not accessible here), not on the callee. It also registers the bytecode address of the instrumented MIDP methods (again, AspectJ was not designed to extract bytecode offsets).

For each method logged, we send its bytecode offset and a unique representation of its parameters. Fortunately, m.toString() usually gives back the name and the address of the object in memory. There is a (very) low probability of address conflicts with garbage collection.

Communication with the phone is done using the serial connection protocol:

- Serial connection over USB with real devices [3]
- Using socat to establish a fake serial line on the PC with a midlet emulator.

The application protocol itself is line oriented so that it is easy to debug. Enough information is provided to follow the execution path in the navigation graph of the midlet [4]

As we carry on the test campaign, it may appear that some transitions visible in the navigation graph are not reachable by testing. Some of these transitions may cause some security problems if they could be taken (for example because they trigger the sending of SMS messages).

2.4.3 Proving restrictions with invariant propagation

Usually those transitions are in fact not active because an invariant is enforced on the states corresponding to the source node that prevents the midlet from taking that path. This invariant may involve data structures that are not directly related to the description of the GUI of the midlet and has been missed by the static analysis phase.

The principle of the proof is the following. For each such transition, we instrument the call in the listener callback that changes the display with the BML assertion False meaning that this state cannot be reached. Then we use the Weakest precondition calculus on the Java bytecode to propagate back this assertion back to the entry point of the callback where we obtain a set of proof obligations. The shape of those obligations is:

$$H_1 \Rightarrow \ldots \Rightarrow H_n \Rightarrow \mathbf{False}$$

We strengthen them by removing the hypothesis involving the arguments of the callback (the previous Displayable object and the Command for a regular CommandListener callback) as it is difficult to get a pointer to the actual objects and the information available on them has already been used by the static analysis phase.

Now we need to prove those invariants for the paths that establish the source state. We add those invariants as assertions at the end of the paths that changed the current displayable to be the source display of the transition we try to eliminate (they are requirements for the callback method when those paths are taken).

Again we use the weakest precondition calculus and we try to conclude on all the proof obligations with an automatic theorem prover. The implementation uses the BtoJ library that is part of the implementation of the Jack tool [14]. Our implementation is only able to solve “trivial” goals where some hypothesis are obviously contradictory. It is possible to use an SMT solver (Simplify for example) on the formulas generated by Jack.

2.4.4 Benchmarks

We have experimented our approach on a very simple midlet made of a single class that act as both the midlet and the command listener that can display three displayable objects (screen contents) d1, d2 and d3. It also defines a field used as a global state such that its value is i if and only if the current screen is di.

For the purpose of the example there is a single command but the command listener only use its internal state.

[3] Another more complex alternative is to use bluetooth and the RFCOMM protocol.

[4] The navigation graph is built on the instrumented midlet and not on the original one, so that bytecode offset of method calls are the same for the graph and the midlet executed.

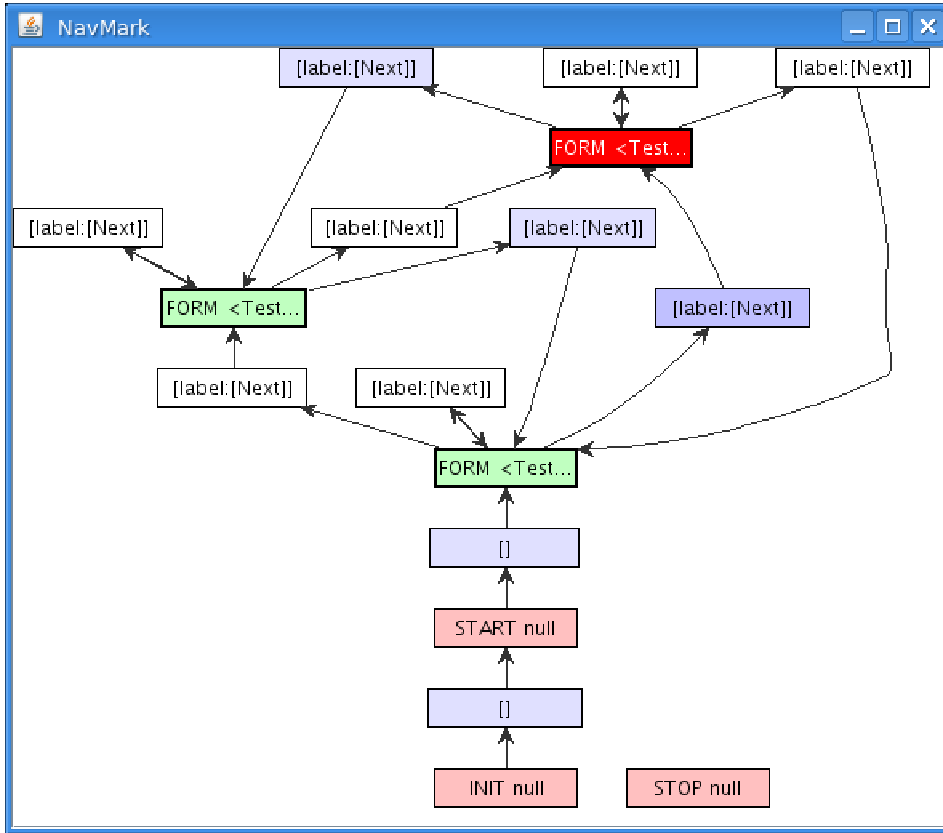

Figure 2.3: Navigation graph of the test midlet with reachable transitions

```
void commandAction(Command c, Displayable d) –
  if (state == 1) –
    state = 2;
display.setCurrent(d2); // (1)
  ” else if (state == 2) –
    state = 3;
    display.setCurrent(d3); // (2)
  ” else if (state == 3) –
    state = 1;
    display.setCurrent(d1); // (3)
  ”
”
```

As expected, the static analysis cannot decide which paths are valid and builds a fully connected graph with three nodes but among those transitions only three are reachable through testing (figure 2.3).

The formula obtained from the verification conditions corresponding to the assert(FALSE) have the following shape (example taken at point (1)):

(! local(0) = NULL) ==¿ 1 = test.Test.state(local(0))
  ==¿ typeof(local(0)) ¡: Ltest/Test; ==¿ FALSE

When injected as assertions at the points where the origin screen for the transitions we try to remove is established (here points (1) and (2)), now the generated VCs contain as hypothesis (for a VC generated at point (2)):

1 = [test.Test.state (local(0)) ¡+ (WITH local(0) ¡- 3)]

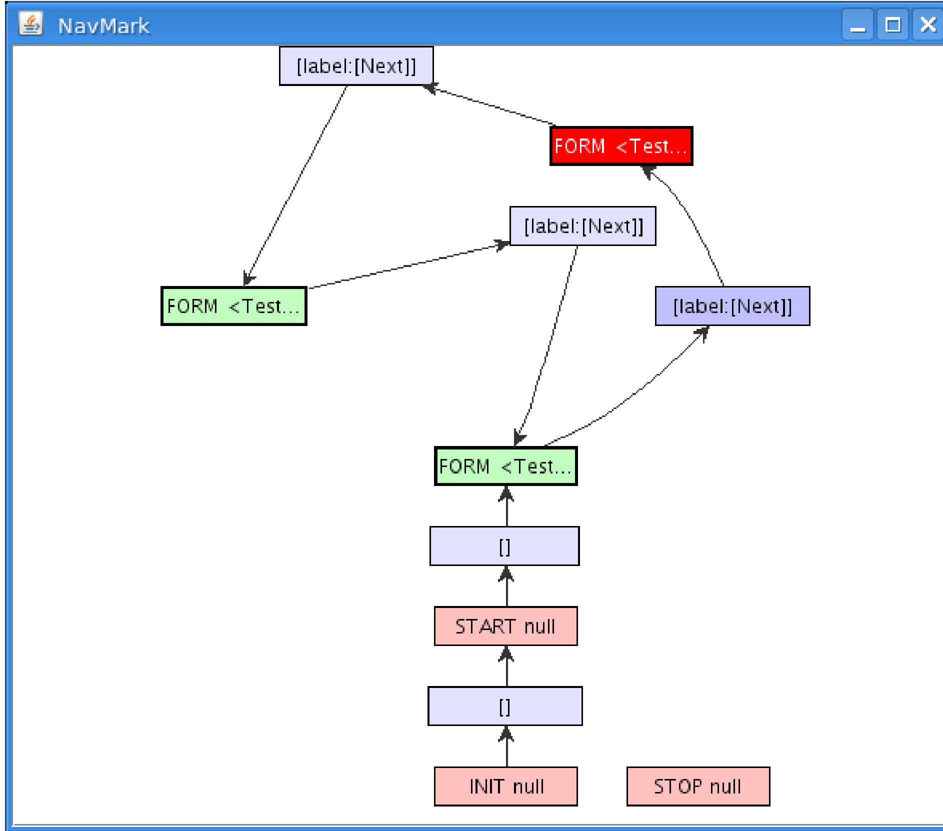

Figure 2.4: Navigation graph after path pruning

which means that $1 = 3$ after substitution. The VCs obtained can then be automatically simplified and proved.

2.4.5 Limitations and further work

Due to lack of time, the implementation of the last phase is very crude and we could probably reduce the number of proof obligations generated (a lot of them are in fact duplicates).

The process presented here is good at pruning transitions from an initial navigation graph. Unfortunately there are cases where we first need to replicate part of the navigation graph before we can apply pruning effectively.

The current version is only valid if nothing occurs on the state between twp calls to callback listeners (for example, threads could modify this state). The invariant that the part of the state used in the proof is not modified by other threads or callbacks that could be triggered between events should be formally checked.

The initial graph may be too coarse and sometimes it is necessary to replicate some nodes before pruning the transitions. When we have a source of non determinim in the navigation graph (a transition associated to a unique source node N and a unique event e with more than one potential target node (for example N_1 and N_2) and two executions p_1 and p_2 taking transitions with target N_1 and N_2 respectively, we can duplicate the nodes (N in particular) and transitions belonging to the longest common subsequence of p_1 and p_2 before the occurrences of N_1 and N_2. In that case it is necessary to propagate the proof obligations on the transitions (that are invariants for the states abstracted by the nodes) among all the duplicated nodes until we find a proof.

2.5 Input validation for a web application

This case study investigated the use of JML for tracking of explicit information flow in a web application, in the style that was also used in Section 2.2. The Mobius PVE was used in developing and checking the JML annotations. We did not attempt generation of BML annotations or certificates for this application, as it is larger than any of the other case studies considered here. So only use was made of ESC/Java2 to develop and check the JML annotations. Still, the case study shows that the PVE is usable by a newcomer to the field of JML specification.

2.5.1 Case Study

In this case study, we look into security requirements of a Defense PLM (Plant Maintenance) web application. This application can be used for sending information via e-mail about certain military equipment, identified by material ID. Since that kind of data can be confidential, e-mail address of the recipient has to be checked inside the database. However, in the relation web application-database, database becomes vulnerable for various kinds of attacks, primarily SQL injection. This is why input validation has to be imposed. Further security requirements are server-side validation and role-based access since not everyone is eligible to send confidential data.

Input validation for input field "Email Address" must be executed prior to making a database call. It includes checking whether any input has been entered and if yes, it is checked for containing character '@'. This second step can consist of applying any filter method on the input and it is placed inside one class - Input. If input is validated, connection to the database can be made when is additionally checked whether the e-mail address of the receiver exists in the database. (This step is placed inside other separate class.) Only in that case, navigation can be continued or form can be successfully saved.

In this case study, code was annotated to track data flows and verify that input validation has been applied in all the right places.

2.5.2 Experimentation

In this case study, it was necessary to perform input validation by using a ghost field that would give information whether the input has been validated regardless of the correctness of the filter method itself. Therefore, the ghost field and validateData() method were implemented as members of class Input as depicted in figures 2.5 and 2.6.

```
//@ ghost private boolean valid;
//@ initially valid == false;
```

Figure 2.5: Ghost Field

```
/*@ ensures valid == true;
  @ signals (InvalidException) true;
  @*/
public void validateData() throws InvalidException {
    if (data.length() == 0)
        throw new InvalidException("Missing input!");
    else if (data.indexOf("@") == -1) {
        throw new InvalidException("Wrong format!");
    }
    //@ set valid = true;
}
```

Figure 2.6: Method validateData()

This specification indicates that after application of validateData() method Input object is considered valid. To ensure this, ghost field valid has to be set to true on every possible normal (i.e. exception free) exit point of the method. Otherwise, valid has value false since it is its initial value. Now we can restrict non-validating DataBaseConnection class to accept only valid input as depicted in Figure 2.7.

If you try to make a database call now with provided input, but without performing input validation first, the PVE will issue warning as depicted in Figure 2.8.

```
/*@ requires in != null; @*/
/*@ requires in.valid; @*/
public void checkUser (Input in) {
 ..
}
```

Figure 2.7: Method checkerUser()

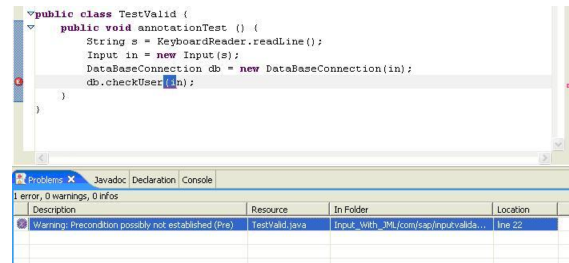

Figure 2.8: Warning

In addition, it should also be ensured that all mutating methods of Input objects reset valid to false as well as that all Input objects are properly encapsulated (cannot be mutated through aliases that bypass the API). In this example it means that ghost field valid must not be set to true outside validateData() method of class Input. Solution is supposed to be found in declaring ghost field private. However, static checks cannot detect violation of this access modifier. Therefore, if Input class is instantiated, ghost field valid can be accessed and manipulated so there is no guarantee that validateData() method has been applied, as depicted in Figure 2.9.

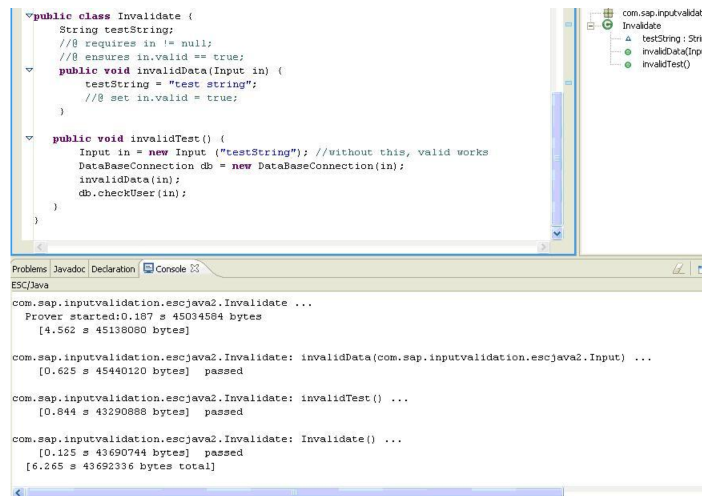

Figure 2.9: Invalidation of ghost field passing checker

Furthermore, value of ghost field can be set only if set annotation is inside the method or constructor which ensures that value. In other words, set and ensures annotation have to be used together in order to change the value of the ghost field.

In the end, input validation is verified by statically checking /Java2 the example project that contains described three classes, KeyboardReader, Input and DataBaseConnection. Besides them, there is class InputCheck that contains main method so Java code can actually be run and perform functionalities that web application has at runtime.

2.5.3 Perspective

If all one is interested in is tracking explicit information flow, then using heavy technology of JML and a checker for JML is a bit of an overkill. It would probably make more sense to use a more lightweight approach, namely using Java annotations to mark security levels of data (in place of the static ghost field) and then using a dedicated checker (for instance based on the JSR308 Checker framework, or based on some Trusted Logic's SAT tool discussed in Section 3.1) to check these, instead of a general purpose program verifier. The formal infrastructure built in Mobius (in particular the 'Bicolano' formal semantics of Java byte code) could be used for a provably correct dedicated checker.

In SAP, there are some programming guidelines to improve the quality of software with respect to security. Performing input validation is among these guidelines. It would be helpful if some mechanism is included in development environment to enforce or remind the developer to follow the guidelines. From this case study, we can see that annotating code using JML as an enforcement mechanism is feasible. Trying to minimize the annotation effort is the next step.

2.6 Case Study with BML tools

This case study investigated the use of BML tools. The Mobius BML tools were used for checking the annotations and finally a class file with a PCC certificate based upon Coq proofs was generated. We were able to verify within our time resources only one class in Coq, namely the Bill class that was produced as a running example for [15]. However, we generated proof obligations for the annotations in the Mobius Quiz developed by TLS [17]. The case study shows that it is possible to generate PCC certificates using BML tools for small examples and generate proof obligations for a smallish midlets.

2.6.1 The Bill Case Study

The Bill class is a small Java class that was developed as a running example for [15]. It was was used as a first small toy example to try out the entire tool chain from annotated source code to certificates.

The experiment involved all the different steps in the tool chain, including the tools for bytecode and BML [8]:

- The JML2BML compiler [12] was used to translate the Java source code with JML specifications to Java bytecode with BML annotations;
- The BMLVCGen was used to produce Coq verification conditions from the BML-annotated bytecode.
- The Coq formulae express the code and its properties in terms of the Bicolano semantics, and were verified with Coq.
- Finally, we successfully packed the proof into a certificate carried within a class file. This certificate in the class file can be unpacked and checked off-device using our certificate handling toolset.

We also tested to Umbra bytecode editor and tried it out to view and edit the generated bytecode and BML specifications. Figure 2.10 presents the BML specifications for a method in the Bill class which were generated by the JML2BML compiler.

The most time-consuming phase in the whole certificate generation process was the construction of Coq proofs for the verification conditions. A lot of additional effort will be needed to make this workable. In the course of the case study, we have made a small step towards this easier by constructing a tactic which transforms Bicolano formulae to arithmetical formulae. A big obstacle in the course of the proof construction is the problem that the formulae the user sees are not comprehensive. In addition, the unfolding of many definitions causes a huge blow up of the formulae. This calls for a development of a set of intelligent tactics which facilitate a reasonable management of the proof development. (Of course, one could go back to using

```
/*@
  @ requires 0 <= n && n < 1000000 && sum < 1000000
  @ modifies \everything
  @ ensures sum <= \old(sum) + n * (n + 1) / 2
  @ signals (java/lang/Exception) true
  @ signals_only \nothing
  @*/
public boolean produce_bill(int n)
0: iconst_0
1: istore_2
2: iconst_1
3: istore_2
4: goto  #24
7: aload_0
8: dup
9: getfield  Bill.sum I (23)
12: aload_0
13: iload_2
14: invokevirtual Bill.round_cost (I)I (25)
17: iadd
18: putfield  Bill.sum I (23)
21: iinc  %2 1
24: iload_2
25: iload_1
/*@
  @ loop_specification
  @ loop_inv i <= n + 1 && sum <= \old(sum) + (i + 1) * i / 2
  @ decreases 1
  @*/
26: if_icmple  #7
29: iconst_1
30: ireturn
31: astore_3
32: iconst_0
33: ireturn
```

Figure 2.10: Bytecodes and BML specifications for the method produce_bill in the Bill class

an automated SMT solver to solves some proof obligations, rather than an essentially interactive theorem prover such as Coq.

Moreover, the verification conditions that were generated contained not enough information to complete the necessary proofs. Therefore, a considerable review of the process of the verification condition generation is necessary to make it complete to the degree which allows to prove at least cases which occur in practice. Although, one must say that the verification condition generation is in all cases sound and constitutes a difficult task in which it is not easy to see some small design and development flaws as the relevant software is of big size.

The whole experiment is presented in detail at http://zls.mimuw.edu.pl/~alx/umbra/casestudy/.

2.6.2 The Mobius Quiz and BML tools

The Mobius Quiz was also used as a basis to check the applicability of the BML bytecode tools [8]. The tools were used on an early version of the annotated Quiz midlet, with only the annotations required by ESC/Java2 not to raise warnings about possible program errors. We were able to generate the verification conditions in Coq for the class files resulting from the translation of the source code JML specifications to BML using the JML2BML compiler [12]. However, the support for proving of the verification conditions in Coq is still much undeveloped, so we did not attempt this for the verification conditions of the Quiz midlet. Still, it proves the tool chain for producing Coq verification conditions from JML-annotated midlets works.

Chapter 3

Evaluation of type systems

This chapter describes the work in Task T5.2 on the evaluation of type-based verification techniques.

Section 3.1 describes Trusted Logic's SAT (Static Analysis Tool), which has mainly been extended and used to study information flow properties of MIDP applications.

Section 3.2 describes case studies using UPM's COSTA system for obtaining upper bounds on resource usage, including analysis peak heap memory consumption and execution time (more in particular, termination analysis).

Section 3.3 explores the middle ground between ignoring implicit flow and taking all implicit flows into account for information flow, using SAP web-application software to evaluate the effectiveness.

3.1 Prototype for an explicit information flow

TL SAT (short for static analysis tool) is a tool for the static checking of numerous security policies varying from structural properties, specification properties, security properties, access control etc. Its main use has been in the scope of the security evaluation of mobile Midlet applications prior to making them available on mobile operator web sites. Information flow policies which so far have not been taken into account in the TL SAT are an integral part of the overall security especially in the context for the mobile applicative layer. Therefore, TL has been particularly interested in the integration of the information flow techniques developed in MOBIUS as part of TL SAT. We have therefore produced an information flow prototype with selected features of the MOBIUS information flow algorithms.

Making a technique scale up in an industrial context requires several conditions to be met. The algorithms should have a full coverage of the target language, i.e. it should be possible to analyse any program in the target language. It is also important that the algorithm has a reasonable complexity in terms of computation resources and time. Interaction with the user should be reduced at a minimum and the internal technical details of the algorithm should be transparent to him. On the other hand, the algorithm must provide interesting information, for instance a too strong analysis which rejects most of the programs will be of less help.

In the scope of MOBIUS, we extended the tool with an explicit fully automatic information flow analysis to trace information flows coming from sensitive origins. This analysis is an extension of the original semantical analysis which calculates overapproximations of the program behavior in every point of the program code.

Properties which this new analysis can check address information flow related to the use of critical framework APIs. In summary, the analysis consists in inferring overapproximations for the data origin of the values manipulated by the analysed application and particularly the values passed as parameters to APIs which may allow an attacker to access sensitive information manipulated by the application. Examples for the type of data origins that we trace currently are as follows - data which is coming from the personal information store, data which is coming from the network, data which is coming from the device system properties, data which is coming from the jad file etc.

Apart from the extensions in the analysis itself we extended the tool with several new rules to trace the the information flows. More particularly, those new properties are concerned with the application connectivity and communication features:

- sms messaging and more particularly how their destination is specified. It is important to detect whether the applet sends messages which are potentially out of the user knowledge. This can be for instance the case when the destination phone numbers are completely specified in the code or coming from the network.
- the access to the system information and more particularly who accesses it. Tracing malicious access to this information can be done via tracing the origins of the names of the accessed system properties. The same applies for the dynamic instantiation of classes via the reflection API where we can trace the origin of the dynamically instantiated classes.
- network connections and who specifies them. It is important to trace who performs network connections. For this, we will be interested in the origin of the urls to which network connections are realised. For instance, it is acceptable if a connection url is specified by the user but the connection can be potentially malicious if the connection url is coming entirely from the network.
- network communication and more particularly what information the applet outputs and to what site. It is important to understand what is the kind of information that an application leaks, to whom it leaks the information, what is the nature of the leaked information i.e. whether it is a sensitive data like the system settings, user personal data etc.

To illustrate how we infer information about the origin of the values manipulated by a program, consider the following example program which opens the user contact list, reads names and phones and sends them over the network.

Listing 3.1: Example for an implicit flow over the network

```
1   ContactList contacts = null;
2    //open the user contact list
3   contacts = (ContactList) PIM.openPIMList(PIM.CONTACT_LIST);
4   //@  contacts → {PIM}
5   Enumeration<Contact> enPimItem = pimList.items();
6   //@  contacts → {PIM}, enPimItem → {PIM}
7    //open an http connection
8   connBad = (HttpConnection) Connector.open("http://www.malicious.com");
9   OutputStream ous = connBad.openOutputStream();
10   //get the name and phone from which Contact item
11  while (enPimItem.hasMoreElements()) {
12     Contact pimItem = (Contact) enPimItem.nextElement();
13     //@ pimItem → {PIM}
14     String contactName = contact.get(Contact.NAME);
15     //@ contactName → {PIM}
16     String contactTel = contact.get(Contact.TEL);
17     //@ contactName → {PIM}, contactTel → {PIM}
18      //send the contact information to "http://www.malicious.com"
19     ous.writeUTF("name= "+contactName);
20     ous.writeUTF"phone= " +(contactTel);
21  }
```

Here, the call to the method OutputStream.writeUTF(String) can potentially communicate with a malicious network site and thus leak a personal information to somebody which could abuse of it, e.g. use it for telephone spamming. The analysis assumes security types for the framework APIs which are then propagated over the program values which are affected or are returned by these APIs. For instance, for the method PIM.openPIMList we assume that the returned value is coming from the PIM. The information is

then propagated to the flow of values via an abstract execution. The approximations of the poststate of each statement is in comments introduced by //@ after the respective statement.

The current example is simple as the sensitive information is accessed and then sent to the network in the same method. The prototype handles more complex cases where information about data origins is propagated through method calls and takes correctly into account the presence of the memory heap and multithreading.

3.1.1 Limitations

The implemented analysis can detect any explicit information flow. This means that situations like in Fig. 3.1 where the flow leaks sensitive information in a variable accessible by an attacker will be detected. However, the prototype does not take currently into account implicit information flows, typically flows caused by a conditional statement where the condition depends on a sensible information will not be detected.

The implicit information flow has been for the moment ignored as it relies on preliminary bytecode analysis to determine the control dependency regions in the code. Such analysis implies a non trivial algorithm which has at least a quadratic complexity.

An origin of imprecision for the prototype is the model of container data structures. The model of containers currently ignore the difference between the elements stored in it by abstracting them by a single abstract element. Containers like vectors or hash tables are widely used in Java code and in particular in Midlets (the MIDP framework relies on a garbage collector so applications does not have to follow a constraint memory use policy). As a consequence, once an object is entered in a container we lose its individual information flow approximation. This can some times result in imprecisions as containers like vectors are widely used in MIDP applications and future extensions of the framework should provide a more precise model for those.

3.1.2 Case studies

We have chosen two midlet applications described in [?] and one taken from the web, each of which features network connections and actions which may potentially compromise the user personal data.

Table 3.1 shows statistics concerning the analysis performed on the examples that we have selected. The first column indicates the size of the applications in terms of the number of classes in the application jar file. The second shows the numbers of application method analysed. Note that the design of the tool is such that API methods are rather modeled than analysed so the tool analyses only application code. The third column shows the analysis time. The last two columns indicate the total number of errors and warnings concerning all groups of properties checked by the tool. The analysis reports an error when the over - approximations allow to conclude that a problem is possible (there is a possible execution which may produce this error) and it reports a warning when the over - approximation does not allow to decide whether a problem may occur.

Midlet Application	number of classes	analysed methods	time(min)	errors	warnings
Google Maps	78	554	10	644	116
Google Local	59	420	5	137	33
Mobile Emule		200	3	30	7

Table 3.1: Characteristics of analysed midlets and analysis result

We can see that the applications have a realistic size in terms of number of application classes, the number of analysed methods also confirms this. The analysis time remains definitely reasonable compared to the time which can be spent for manual inspection of the same properties that the tool supports.

The following tables show statistics concerning the results of the information flow analysis over the case study applications. The columns in the table determine the origin of the data sent potentially to the network and for each application there are two rows - one for the data which was partially approximated(e.g.

a string data is partially approximated when we know part of the prefix and) or undetermined (e.g. a string data is undetermined when the analysis can not guess any prefix). Note that all of the application feature connectivity and the main objective of the evaluation was to trace whether the information leaked via these connections is sensitive.

Table 3.2 shows what is the origin of the data leaked through the internet connections, e.g. the midlet Google Maps sends to the network at 7 places in the code information which may be hardcoded, at 3 places information which is coming from the jad file etc. Note that for Google Maps none of the data sent to the network could be even partially determined.

Midlet	Errors							Warnings
Google Maps		CODE	DEV	JAD	USER	NET	SYSTEM PROP	1
	undetermined	7	3	2	2	4	4	
	partially determined	0	0	0	0	0	0	
Google Local		CODE	DEV	USER	NET	SYSTEM PROP		1
	undetermined	0	0	0	0	0		
	partially determined	1	1	1	1	1		
Mobile Emule		CODE						1
	undetermined	2						
	partially determined	5						

Table 3.2: Report on the data sent over the network

Table 3.3 shows the results of the analysis concerning the origin of urls to which the applications connect. Each row describes the url origins for each of the three applications in our test. For each application we give the detected origin of urls. For instance, in Google Maps there are 18 urls which are partially determined and which are partly coming from the code; in the same application there are another 11 urls which are also partly coming from the code but the string analysis could not infer any part of their prefix.

Midlet	Errors						Warnings
Google Maps		CODE	JAD	USER	NET		0
	undetermined	11	7	16	11		
	partially determined	18	5	5	5		
Google Local		CODE	DEV	JAD	USER	NET	0
	undetermined	1	1	0	1	1	
	partially determined	3	0	1	1	0	
Mobile Emule		CODE	USER				0
	undetermined	0	0				
	partially determined	1	1				

Table 3.3: Origin of urls

3.1.3 Conclusion

As we saw in the previous subsection the prototype for explicit information flow developed at Trusted Labs works over realistic examples and gives interesting results which help our security evaluators for tracing the information flow in the analysed Midlet applications. This results are quite promising and we will continue working on the improvement of the analysis and more particularly on providing support for implicit information flow.

3.2 Experiments with the COSTA System

COSTA [1] is a static analysis system which allows obtaining safe symbolic upper bounds on the resource usage of Java bytecode. COSTA follows the classical approach to static resource analysis proposed in Wegbreit's seminal work [22] and which consists of two phases. First, given a program and a cost model, the analysis produces cost relations (CRs for short). Second, the systems tries to obtain closed-form upper-bounds for them. The results are symbolic in the sense that they do not refer to concrete, platform dependent, resources such as execution time, but rather they provide platform-independent information. This has the advantage that the results are applicable to any implementation of the Java Virtual Machine (JVM) on any particular hardware. It also has the disadvantage that the information cannot refer to platform specific resources such as run-time. The fact that the analysis handles Java bytecode represents that, at least in principle, it can deal with general-purpose programs written in a mainstream programming language such as Java and potentially other languages compiled to Java bytecode. The upper bounds computed by COSTA can then be compared against user-provided resource usage specifications. This allows automatically rejecting code not guaranteed to execute within the specified resources.

We refer the reader to MOBIUS deliverable 2.7 for further details on the underlying techniques used by COSTA. In this deliverable we focus on assessing the practicality of our approach by presenting a number of experimental results obtained using COSTA. In Section 3.2.1 below we present an experimental evaluation of the systems when computing upper bounds of a relatively large range of Java bytecode programs. Then, in Section 3.2.2 we discuss the experimental results obtained when extending the COSTA approach for estimating peak heap usage. Finally, in Section 3.2.3 we present an experimental evaluation of the system which evaluates the gains introduced by handling numeric fields in termination analysis of a subset of the standard Java libraries.

3.2.1 Upper Bounds for Java Bytecode Programs

Though the efficiency and robustness of the system can be considerably improved, COSTA can already deal with a relatively large class of Java bytecode programs, and gives reasonable results in terms of precision and efficiency. We plan to distribute the system as free software soon. It can be tried out through a web interface available from the COSTA web site: http://costa.ls.fi.upm.es. To the best of our knowledge, this system is the first to provide evidence that cost analysis can be applied to programs written in a realistic object-oriented programming language, in bytecode form. COSTA allows choosing between several options, by specifying, for instance,

1. whether the code of external libraries (i.e., methods in the Java API which do not belong to the benchmark itself, but are called from the methods in the benchmark) should be also analyzed;

2. whether auxiliary analyses (sign, nullity, slicing, constant propagation) should be included, thus possibly improving both precision and performance;

3. if exceptions, either explicitly thrown in the code or resulting from semantic violations, have to be taken into account;

4. which cost model has to be considered.

The system can deal with most features of Java bytecode. Non-sequential code, dynamic code generation and reflection are not supported. Currently, COSTA handles bytecode programs for Java SE 1.4.2_13. However, there is no fundamental reason for not supporting more recent Java versions and we plan to extend COSTA to also handle Java 5 and 6 soon. As for native code, i.e., methods not implemented in Java, calls to native methods are shown in upper bounds as symbolic constants, since the code for those methods is not written in Java and COSTA cannot analyze them. This could be further improved by providing

[1] More information on COSTA can be found at the COSTA web site: http://costa.ls.fi.upm.es.

assertions which describe the cost of the native method for the different cost models and (optionally) a safe approximation of their input-output behavior, but do not support it yet.

In the paper [4], reported in Deliverable 2.7, we show different tables which show the practicability and efficiency of COSTA. Both the Java source and the bytecode for all programs used as benchmarks is available at the COSTA web interface. We consider two sets of benchmarks which range from constant to exponential complexity.

As regards the options for the analysis previously described, in our experiments we (1) analyze Java API, (2) activate all auxiliary analyses (3) consider all exceptions. It is important to mention that the current implementation of several components of the system is not optimized for efficiency, since the main aim for the time being has been to see whether the approach allows obtaining useful results. Now that the applicability of the approach is proved, we are working on more efficient and robust implementations.

We argue that the computed upper bounds are useful since they are both reasonably accurate and simple. As can be seen in the table, the upper bounds obtained can be constant, logarithmic, linear, n-log-n, quadratic, polynomial and exponential. It is worth mentioning that COSTA has obtained precise upper bounds in all cases.

3.2.2 Peak Heap Consumption for Java Bytecode

As a second group of experiments related to the amount of heap consumed by a program, the paper [5] reported in Deliverable 2.7, describes two tables containing results on the memory consumption for the standardized set of benchmarks in the JOlden suite. This benchmark suite was first used in the context of memory usage verification for a different purpose, namely for checking memory adequacy w.r.t. given specifications, but there is no inference of upper bounds as our analysis does. It has been also used for our same purpose, i.e., the inference of peak consumption. However, since previous work does not deal with memory-consuming recursive methods, the process is not fully automatic in their case and they have to provide manual annotations. Also, they require invariants which sometimes have to be manually provided. In contrast, our tool is able to infer accurate live heap upper bounds in a fully automatic way, including logarithmic and exponential complexities.

3.2.3 Termination Analysis for Java bytecode

The final set of experiments we have performed consists in trying to automatically infer termination of Java bytecode methods. We have observed that the termination behavior of large number of Java methods depend on the value of numeric fields, i.e., instance variables of integer type. Though there is a large body of work in static analysis of numeric values, numeric fields are stored in the heap and most of existing value analyses cannot capture them.

Therefore, in [3], we have devised new techniques for performing value analysis of programs involving numeric fields and we have applied them to the termination analysis of Java programs. In more detail, in order to assess its practicality on realistic programs, we have tried to infer termination of all the loops which contain numeric field accesses in their guards for all classes in the sub-packages of “java” of the Sun’s implementation of J2SE 1.4.2. In total, we have found 133 methods which contain loops of this form, which we have taken as entries. COSTA has an application extraction algorithm (or class analysis) which pulls methods transitively used from each entry. COSTA failed to analyze 11 methods because when analyzing context-independently, it is required to analyze more methods than it can handle.

Before applying our technique, COSTA could prove termination of only 4 of the 176 loops. In those 4 loops it is possible to prove termination using a field-insensitive analyzer because, for example, termination is guaranteed by reaching exceptional states. When we apply our approach to field-sensitive analysis, we prove termination of 165 of the 176 loops. It is also worth mentioning that only in 3 loops we fail to prove termination because the numeric field in the guard is not trackable (in particular, the reference is not constant). In the other 8 loops, though the fields are trackable, we failed due to limitations of the underlying termination techniques used in COSTA, and which are not related to our approach. In most

cases, the problem is that the termination condition does not depend on the size of the data structure, but rather on the particular value stored at some location within the data structure, and also to the use of linear arithmetic operations. On average, the overhead introduced by field-sensitive analysis is 2.51. We argue that our results are quite positive since the overhead introduced is reasonable in return for the quite significant accuracy gains obtained.

3.3 Implicit flows in nonmalicious code

3.3.1 Introduction

Information-flow technology is a promising approach for ensuring security by design and construction. However, recent experiments indicate that information-flow technology has run into a practical obstacle: false alarms. These alarms result from implicit flows, i.e., flows through control flow when computation branches on secret data and performs publicly observed side effects depending on which branch is taken.

The large body of literature exercises two extreme views on implicit flows: either track them (striving to show that there are no leaks, and often running into the problem of false alarms), or not track them (which reduces false alarms, but all bets are often off in terms of security guarantees).

This work explores a middle ground between the two extremes. We observe that implicit flows are often harmless in nonmalicious code: they cannot be exploited to efficiently leak secrets. To this end, we are able to guarantee strong information-flow properties with a combination of an explicit-flow and a graph-pattern analyses.

To evaluate the effectiveness of our approach, we have performed two case studies with industrial applications at SAP written in Java: secure logging and data sanitization in the context of cross-site scripting vulnerability prevention. Due to space limitation, the work about cross-site scripting is not described here.

3.3.2 Information-leakage patterns

The purpose is to check whether the following two information-leakage patterns are common or not: a loop on expressions that involve high variables or a loop whose body contains branching on such (high) expressions.

We do this examination in the following steps:

- Security levels are assigned to variables.
- Control-flow graphs are generated for methods using Control Flow Graph Factory plug-in from Dr. Garbage [1].
- For each loop found in the control-flow graphs, whether the guard is high is checked. If the security level is high, the first information-leakage pattern is found.
- If the security level is low, whether inside the loop there exists branching on a high expression is checked. If yes, the second information-leakage pattern is found.

In our case study, there is only one information-leakage pattern is found. The case is that a user could try to log in into the system for at most three times. If the user is not able to provide valid credential during the three times, his or her account will be locked. At first sight, this appears as a loop with branch on variable with high security level. But since the time of trying is fixed to at most three times, this loop can be unfolded into a sequence of at most three conditionals. Therefore, this pattern is not dangerous (and the information leak is bounded by the number of conditionals in the unfolding: at most three bits).

3.3.3 Logging API and encryption API

Logging and tracing are important elements for securing application server systems. Logs are important for monitoring the security of the system and to track events if problems occur, as well as for auditing the correct usage of the system.

The SAP Logging API is provided with all functionality for both tracing and events logging. The following methods are provided to write messages with different severity. They have intuitive names with severity level indicated, such as FATAL, ERROR, WARNING, INFO, PATH, and DEBUG.

```
fatalT(string the_message) ;
errorT(string the_message) ;
warningT(string the_message) ;
infoT(string the_message) ;
pathT(string the_message) ;
debugT(string the_message) ;
```

For more information about SAP Logging API, please refer to [2].

One security consideration is that before sensitive information is logged, it must be encrypted in order to prevent information leakage.

In SAP NetWeaver Platform, the following interfaces and classes derived from them are available for implementing digital signatures and encryption in the applications.

The interface ISsfData is the central interface used for the cryptographic functions. Its underlying classes specify the data format used, for example, SsfDataPKCS7, SsfDataSMIME and SsfDataXML. The available methods are sign, verify, encrypt, decrypt, and writeTo.

For more information about SAP interfaces and classes for using digital signatures and encryption, please refer to [2].

3.3.4 Security analysis for logging

As said before, one security consideration in SAP system is that before sensitive information is logged, it must be encrypted in order to prevent information leakage. From the point of view of information-flow analysis, the variables containing sensitive information are with high security levels, and the log file is with low security level. The only allows way for high security level information flowing to the log file is by declassification.

Based on the observation of information-leakage pattern, we use the explicit flow analysis to check whether the secure logging consideration has been applied. The main items are discussed as follows.

The first step is to assign security level, i.e., $\Gamma(v)$, to each relevant variable v. This needs domain knowledge of the developer, and is done manually. Variables containing sensitive information, e.g., password and salary, are assigned high security level. The variable associated with the log file is assigned a low security level.

In one file of our case study, a piece of control-flow graph as depicted in Figure 3.1 is found.

In the figure, line 320 is the declassification, and line 326 is log writing.

In addition to that, the branch, i.e., line 317, is on the variable with high security level. According to the previous discussion, at most one bit of the sensitive information may be leaked.

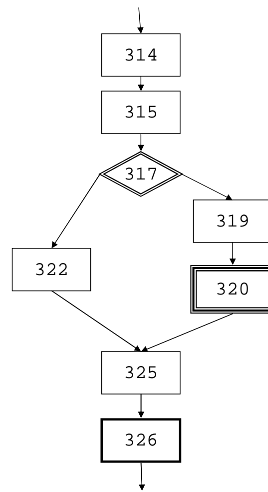

Figure 3.1: A Control-Flow Graph of Secure Logging

Chapter 4

Conclusion

Below we draw some conclusion from evaluation phase of the MOBIUS project as described in the preceding chapters of this document.

- The PCC proof chain
  The entire tool chain from JML-annotated code to Coq proofs to certificates is working and can cope with real Java code with JML annotations. In more detail, the tool chain allows

  - translation of JML to BML using the JML2BML compiler,
  - possibly editing of the generated BML using the Umbra editor,
  - translation BML to verification conditions in Coq using the Direct VCGen BMLVCGen,
  - proving these verification conditions in Coq, and finally
  - wrapping up the Coq proof to a certificate using the CTT tool.

  Verification with Coq is still very time-consuming, and has only been used on tiny toy examples (e.g. the Bill class). Clearly, improved Coq tactics will have to be developed in order to cope with bigger examples.

  Still, the tool chain from JML to BML to verification conditions in Coq does work for realistic sized midlets. In particular, it has been shown to work for the Mobius Quiz Game made as a demonstrator by TLS.

- API and midlet annotations
  Producing the necessary JML specifications for the MIDP API in order to do the midlet case studies proved to be doable – it did not take up the huge effort that we initially feared. Indeed, for small examples as the one described in Section 2.2, we could successfully perform all the phases of the verification chain, i.e. both the specification of the security policy in JML and the verification of the application against the latter. Another important point is that the verification was carried out by a non expert which shows that for security policies which are not very complex the verification task is accessible without an important knowledge in formal methods.

  However, annotating non-trivial midlets to express for instance conformance to a navigations graphs, as we did in Section 2.1, does take a considerable effort. Moreover, it is not so easy to separate annotations expressing the policy from additional annotations needed to carry out the verification, or to exclude the possibility that some additional annotations (notably set-statement altering the values of ghost fields) trivially make the verification go through. Both of these aspects would have to be dealt with to move from 'just' having program verification to having PCC.

- Feasibility
  Many of the algorithms developed in the project have been implemented or applied to case studies. For instance, a variation of the MOBIUS information flow analysis which treats explicit information

leakage has been implemented in the static analysis tool for MIDP applications TL SAT. It was also demonstrated how simple information flow properties can be specified and verified with the help of logical verification implemented in the MOBIUS PVE.

Interesting practical approaches have been developed to treat both implicit and explicit information flows in order both to reduce the number of false alarms (inherent for implicit flow analysis) and keep an important level of security guarantees (difficult to achieve only by considering explicit information flow). This technique has been applied successfully to to the SAP framework and technology as described in Section 3.3. Other approaches relax information flow analysis by providing mechanisms for declassification (leaks of sensitive data are allowed under certain conditions e.g. when the user has entered a correct password) as described in Section 2.3.

The techniques related to the analysis of resource consumption on devices with limited resources have been implemented in the COSTA tool and applied to different case studies depending on the selected cost model. Concretely, COSTA incorporates techniques for computing precise upper bounds on the number of executed instructions, amount of memory allocated in the heap, and calls to a concrete method, for programs which make use of Java libraries. The experimental results performed to-date are very promising and they suggest that resource usage and termination analysis can be applied to a realistic object-oriented, bytecode programming language. More platform specific techniques have been also developed in the scope of MOBIUS. For instance, the verification technology called "navigation graph analysis" for checking the secure behavior of MIDP applications has been implemented.

- Practical interest

  These techniques address real life problems of the mobile and embedded software industry. For instance, the developed information flow techniques identify information flow leakage which is an important issue particularly in the context of connected and open personal devices. One purpose of the security validation performed over MIDP applications is to identify such malicious behavior prior to making the application accessible online. It is however true that automatic support for tracing information flows was not available before the project Mobius in the tool suite of TL which also provides services for the security evaluation of Midlet applications.

  Constraint resource consumption is of a great importance for limited devices like smart cards. Providing correct estimations for the resource consumption for an application can improve significantly the development and deployment process. In this respect, the estimations provided by the tool COSTA can be useful in the development of applications tailored to such limited devices. The specific technique for MIDP applications "navigation graph analysis" which tracks critical actions performed by applets in between two user interactions with the phone device, aligns also perfectly to concrete industrial needs for security of mobile code.

- Scale up

  The implemented techniques have been applied to a bunch of applications varying in complexity and size. The case studies therefore show that the developed techniques cover different programming language features and can manage applications with realistic complexity. It is however true, especially for the logical verification techniques that the human effort required for specification and verification can be important. On the other side, using these techniques in general is not a trivial task and inherently requires important expertise in formal reasoning and understanding of the program semantics.

The results provided by the evaluation phase are promising and we can identify several directions for future work. Some of the most challenging techniques like verification of multithreaded programs addressed in the MOBIUS project are still in a phase of active theoretical research. This is not surprising as the problems to be resolved are complex and need more long time research. On the other hand, multithreading is a real life feature in many software fields and nowadays with advances in hardware technology it is even present on applications for small devices like mobile phones. Therefore future work in the lines of MOBIUS should continue concentrating on this problem.

Another important point is how the techniques which are the subject of this document inscribe in the context of PCC. In particular, the logical verification technology implemented and integrated in the MOBIUS PVE should be carefully considered in order to identify the features which can be used in PCC certificates.

Concerning the information flow analysis implementations they are currently restricted to a subset of the information analysis developed in the MOBIUS project. In order to provide a fully - fledged technology for certifying safe information flow, the technology must address the whole set of information leakage problems. This is a challenging problem as it is difficult to find a good compromise between precision and practical complexity.

Bibliography

[1] Dr. garbage. http://www.drgarbage.com.

[2] Sap netweaver 7.0 knowledge center. http://help.sap.com/content/documentation/netweaver/docu˙nw˙70˙design.htm.

[3] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Field-sensitive value analysis by field-insensitive analysis. 2009. Submitted for publication.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. Submitted to ACM Transactions on Programming Languages and Systems.

[5] E. Albert, S. Genaim, and M. Gmez-Zamalloa. Live Heap Space Analysis for Languages with Garbage Collection. In ISMM'09: Proceedings of the 8th international symposium on Memory management, New York, NY, USA, June 2009. ACM Press.

[6] G. Barthe, P. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In R. Foccardi, editor, Computer Security Foundations Workshop, pages 100–114. IEEE Press, 2004.

[7] Richard Bubel, Reiner Hähnle, and Benjamin Weiß. Abstract interpretation of symbolic execution with explicit state updates. In Springer-Verlag, editor, Formal Methods for Components and Objects, Lecture Notes in Computer Science, 2009. Accepted to FMCO'08 post proceedings (to appear).

[8] J. Chrzszcz, M. Huisman, A. Schubert, J. Kiniry, M. Pavlova, and E. Poll. BML Reference Manual, December 2008. In Progress. Available from http://bml.mimuw.edu.pl.

[9] Pierre Crégut. Extracting control from data: User interfaces of MIDP applications. In Trustworthy Global Computing: Revised Selected Papers from the Third Symposium TGC 2007, number 4912 in Lecture Notes in Computer Science, pages 41–56. Springer-Verlag, 2008.

[10] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In R. Gorrieri, editor, Workshop on Issues in the Theory of Security. IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS, 2003.

[11] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, Security in Pervasive Computing, volume 3450 of Lecture Notes in Computer Science, pages 193–209. Springer-Verlag, 2005.

[12] J. Fulara, K. Jakubczyk, and A. Schubert. Supplementing java bytecode with specifications. In T. Hruška, L. Madeyski, and M. Ochodek, editors, Software Engineering Techniques in Progress, pages 215–228. Oficyna Wydawnicza Politechniki Wroclawskiej, 2008.

[13] C. Hurlin. Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic. PhD thesis, Université Nice Sophia Antipolis, 2009. To appear.

[14] Jack website. www. See http://www-sop.inria.fr/everest/soft/Jack/jack.html.

[15] MOBIUS Consortium. Deliverable 3.1: Bytecode specification language and program logic, 2006. Available online from http://mobius.inria.fr.

[16] MOBIUS Consortium. Deliverable 3.3: Thread-modular verification, 2007. Available online from http://mobius.inria.fr.

[17] MOBIUS Consortium. Deliverable 5.1: Selection of case studies, 2007. Available online from http://mobius.inria.fr.

[18] Wojciech Mostowski and Erik Poll. Midlet Navigation Graphs in JML. Technical Report ICIS–R09004, Radboud University Nijmegen, August 2009. Available at https://pms.cs.ru.nl/iris-diglib/src/getContent.php?id=2009-Mostowski-MidletGraphs.

[19] E. Poll and A. Schubert. Verifying an implementation of SSH. In Workshop on Issues in the Theory of Security, pages 164–177. IFIP WG1.7, 2007.

[20] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. Journal of Computer Security, 2007.

[21] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot – a java bytecode optimization framework. In Proc. of Conf. on the Centre of Advanced Studies for Collaborative Research, pages 125–135, 1999.

[22] B. Wegbreit. Mechanical program analysis. Communications of the ACM, 18(9), 1975.