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 D2.4

Report on implementation of type systems

Due date of deliverable: 2008-03-01 (T0+30)

Actual submission date: 2007-04-21

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

Organisation name of lead contractor for this deliverable: **TL**

CREATION

Project co-funded by the European Commission in the Sixth Framework Programme (2002-2006)		
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)	

Revision

Revision Table

Chapter	Difference to Deliverable 2.2
1	Revision of chapter
2	Revision of chapter including presentation of the heap model and analysis in the TL static analysis tool for the Java Card framework
3	Revision of chapter including • description of improvements of the inter method analysis based on method ordering and the Midlet lifecycle • the heap model used in the TL static analysis tool tailored to the Midp framework
4	Revision of chapter
5	Revision of chapter

Executive Summary

Report on implementation of type systems

This document summarizes the work performed in the context of task T2.6 of project FP6-015905 (MOBIUS), co-funded by the European Commission within the Sixth Framework Programme. Full information on this project, including the contents of this deliverable, is available online at `http://mobius.inria.fr`.

The main focus of task 2.6 is to confront the most forward-looking academic research work on type systems and the current constraints of the industrial world. The objective is here double. First, some of the technology developed by academic researchers can be transferred to industry in a fast cycle. Then, some of the feedback provided by the industrial prototype developers can be used by the designers of advanced type systems and algorithms in order to identify interesting and challenging issues that were overlooked at first.

In Chapter 2, we present the static analysis tool developed by TL for Midp programs. In particular, we describe the underlying techniques of abstract interpretation - the abstraction of program values, the model of the heap, etc. We also give an account about the abstract domain of strings as this type of values play an important role in the definition of few security properties. As we shall see, this first version of the Midp analysis tool scales up for large programs on the price of a coarse precision. This imprecision affects methods' input, output as well as the memory heap.

Chapter 3 details about our work in the scope of MOBIUS which aims at improving both the precision and the efficiency of the Midp analysis. The changes are related to the domains of abstract values, the heap model as well as the intermethod analysis. These changes are a necessary step for the integration of the type systems developed in MOBIUS.

Chapter 4 differs from the previous chapters as it is focused on the adaptation of MOBIUS' type systems (described in [4]) in an industrial context. We provide several criteria to measure such integration and show that the type based techniques developed in MOBIUS have a practical interest.

Finally, Chapter 5 concludes with an overview of the results of the task 2.6 and discusses future work.

The document was written by Anthony Ferrari, Mariela Pavlova, and Eric Vétillard from Trusted Labs.

Authors

Site	Contributed to
TL	All document

Contents

Chapter 1

Introduction

This document reports on Task 2.6 (Type System Prototypes) and presents the tool integration of the scientific results from Work Package 2. Task 2.6 of the MOBIUS project consists in developing a prototype implementation of the type systems defined in the other tasks of the work package.

The work in this task differs from other prototyping tasks in the project by the fact that the development is performed by an industrial partner (TL), and is based on an industrial tool. The objective of the task is to explore the practical issues related to the implementation of the type systems, and to use them for the evaluation of actual applications.

In practice, the differences are the following:

- The prototype is based on preexisting TL static analysis technology, which forms the basis of the work in the project.
- The prototype is aimed at the analysis of of a very specific kind of real-life applications (MIDP applications)
- It investigates the practical issues related to type-based analysis.
- The prototype is aimed at the analysis of unmodified applications, without any specific annotations or type certificates, which means that all the typing information needs to be inferred during the analysis.
- The result of this task is a stand-alone result, which will not be integrated further with the other prototypes developed in the project.
- As the development is based on a proprietary industrial product, the results are solely owned by TL.

Chapter 2 describes the technology on which the work is built, in particular the generic library developed by TL for the static analysis of Java programs and the architectures derived from this library. Chapter 3 then describes the results obtained on the implementation of the type analysis tool, focusing on the changes made in the tool in order to support more sophisticated type systems. Finally, Chapter 4 discusses the adaptation of the MOBIUS type systems, as they are described in [4], focusing on the issues which determine their integration in the tool.

The final chapter describes the achievements and future work envisaged on the prototype, as well as the objectives w.r.t; the demonstration of the implemented prototype.

Chapter 2

Static Analysis at Trusted Logic

The challenge of MOBIUS' task 2.6 is to implement a prototype of the type systems to explore the practical issues in the implementation. The prototype is based on Trusted Logic's industrial products and therefore, this chapter presents an overview of Trusted Logic's technologies on static analysis.

2.1 The Generic JVM

Trusted Logic has a long experience in building tools for static program analysis. One of its first major objective was to build a tool that would allow logical reasoning about Java programs by running programs in a formal way. To this end, in parallel to its participation to the SecSafe IST/FET project, Trusted Logic developed a generic Java virtual machine, in which every component of the virtual machine could be replaced by an appropriate formalization [8].

This library supports all features of the Java language and provides all necessary features to implement the dataflow analysis algorithm as described by Xavier Leroy in [2] for bytecode verification. Moreover, this library allows type-level abstract interpretation of a Java virtual machine. The genericity of this virtual machine comes from the fact that it only implements the core logic of the Java Virtual Machine; all other elements need to be provided specifically as an abstraction domain.

The tool is based on the abstractions of the following Java components:

- All values, with different characteristics for integral and reference values.
- The Java stack including all the elements of the frame.
- The Java heap allowing objects to be abstracted in different ways.
- Execution hypotheses, a set of constraints that models the history of a given execution path.
- The behavior of the API methods, which can be entirely simulated in an abstraction.

In addition, the way in which transitions between execution states are handled can be customized in order to meet the specific requirements of different analysis algorithms. In particular, it is possible to implement local analyses as well as global analyses, simply by modifying the execution state transitions.

This ambitious project grinded to a halt as the tool's initially targeted market (formal models required for the high-level security certifications of Java Card applications) did not materialize.

2.2 The Java Card analysis

The generic virtual machine nevertheless represented several years of work. When the need arose to develop an automated static analysis tool for verifying the portability of SIM Toolkit applications with respect to SIM Alliance's requirements [5], it became obvious that the generic virtual machine could be turned into

an abstract machine, suited for running an abstract interpretation of Java Card applications. Although full automation has proven difficult, this static analysis tool remains in use today as part of Trusted Labs' application certification offer for telecom and banking operators.

The Java Card version of the static analysis tool takes advantage of the specificities of current Java Card applications: applications are rather small, they have very limited support for dynamic object allocation, and they are single-threaded. All these characteristics greatly restrict the execution space, and make it possible to obtain precise information that cannot be obtained in other contexts [7]. The analysis algorithm therefore is quite precise:

- The architecture implements the *polyvariant* flow analysis algorithm, *i.e.*, a context-sensitive analysis, where states affected to each program point are determined by the approximation of the control-flow path that lead to this state[1]. The approximation of the control-flow path is precise, and it includes a precise handling of loops, which can unravel small loops (others are approximated).
- The heap approximation is also precise, allocated objects are tracked individually, as they are identified by the control-flow path that led to their allocation. This is of course possible only because Java Card objects are in most cases statically allocated.
- The hypothesis domain, on the other hand, has remained simple, in order to limit the overhead linked to the management of a large set of constraints in a solver. Constraints are only tracked on the most crucial variables, such as the most important bytes of the command that triggered the Java Card application.
- Finally, value abstractions are such that integers are represented by integer intervals, and reference values by the list of objects that the value can represent.

As a result, this analysis provides a good approximation of the program behavior, and thus, allows for checking many security policies for Java Card applications. However, like for any other application framework, the size of Java Card applications continuously increases. Where the typical size of Java Card applications was 10 kilobytes in 2000, 50 kilobyte applications are becoming common, and their code also grows more complex. This new complexity makes the analysis very long. In addition, the proportion of applications for which the analysis does not terminate in a reasonable time increases significantly.

Because of this increase in complexity, the original Java Card analysis has been modified many times over the years, and it has become necessary to significantly refactor it.

2.3 The MIDP analysis

2.3.1 Initial challenges for the analysis tool

In 2003, with the introduction of MIDP 2.0, mobile applications started having access to sensitive resources. A need therefore emerged for the analysis of mobile Java applications [1, 6]. Trusted Logic then decided to adapt their Java Card static analysis tool to this new market. The main challenges to be faced at this stage were at follows:

- MIDP applications are far larger and more complex than Java Card applications, and it is not possible to perform a deep analysis of the entire execution tree. Instead, the analysis needs to be more local, in order to avoid a potential exponential explosion of execution time.
- Because of multithreading, it is very difficult to rely on the current state of the objects in the heap without incurring the high cost of a difficult analysis. The analysis therefore needs to rely on a less precise model of application data.

[1] Note that a static analysis based on such implementation constrains the bytecode structure; in particular, applications shall not use recursive code.

- Strings play an important role in MIDP, as most sensitive API's use string parameters, whereas Java Card (which does not support Java's String type) heavily relies on discrete integer identifiers. A MIDP static analysis therefore needs to be able to reason on string values.

- The API's used by mobile applications are large and complex, and difficult to model. They are based on the core CLDC and MIDP API's, which are often enriched by other JSR's[2].

For its initial prototype, built in 2003 and 2004, Trusted Logic has relied on simple algorithms, based on a local analysis of each method in a program, and a very simple representation of application data. Most of the effort was put in the development of an abstract domain for strings, and on the refactoring of the Java Card tool, in order to allow its reuse for the MIDP tool. This prototype, still in use today, has been fitted with two distinct user interfaces: an interactive and a web-based interface that verifies a set of rules where the latter generates a report for each submitted application. Both interfaces are based on a fixed set of security rules.

Meanwhile, Trusted Labs was founded in 2004 to handle Trusted Logic's business in security consulting and evaluation. The certification of mobile content slowly emerged as a significant issue for mobile operators, in particular as they were required to open even further their platforms to third-party content, and to grant more privileges to third-party content.

As the research and development challenges were deemed difficult by Trusted Logic, collaboration was sought with major research institutions through European R&D projects. This collaborative effort led to the involvement of Trusted Logic in two projects: MOBIUS and S3MS. In MOBIUS, TL is developing the core algorithms for its tools, taking input from the researchers involved in the project. In the S3MS project, TL focuses on the way in which application security properties can be defined as security contracts, and how such contracts can be used with the Java static analysis tool.

2.3.2 Static analysis with reasonable complexity

The first version of the MIDP analysis tool consists in a simple analyzer with a linear execution time (relatively to the program size). More precisely, the algorithm consists in determining for each point in the program, *i.e.* each program counter of a method, what are the possible values in the top Java frame, including the method's local variables and the Java parameter stack as well as an approximation of the possible types in the heap. The analysis have been simplified in several ways which we discuss briefly below.

Simple abstract domains

Analysis based on domains with a complex structure can be a potential source of slow convergence to the solution of the analysis. Thus, the representation of values is greatly simplified, in order to reduce the complexity of the analysis.

- All primitive values are represented as flat lattices, where the only possible abstract values are singletons and $\top$. Integral values are therefore considered as fully determined or unknown, without further information.

- Return addresses (used for the implementation of the Java subroutines) are only kept for reference, since the mandatory off-device preverification removes all uses of subroutines, and therefore of return addresses.

- Object references are represented by a trivial domain that contains the null value, and keeps track of the value's type. However, a specific domain is used for strings and string buffers, which is described below.

[2] Java Specification Requests, which is the way in which extension libraries get defined in Java, through the Java Community Process.

These domains reflect the basic requirements of the analysis, which is to determine some information about the parameters of important methods. Most interesting parameters are either strings or integral values that are expected to be constants (basic parameters, which are fixed for a given program).

Monovariant analysis

The abstract domains are not the only ones that have been simplified. Without providing an approximation of the program execution the worst case complexity of the analysis algorithm becomes exponential in the number of method calls.

Thus, the analysis is here *context-insensitive* or *monovariant*. The monovariant analysis that we use consists in analysing in arbitrary order the methods in the program until reaching stable solutions for the heap and the initial state and return value of every method.

The level of complexity is therefore comparable to the complexity of the standard Java bytecode verifier. Nevertheless, the static analysis does not replace the Java bytecode verifier, and it only works properly on applications that have passed the CLDC preverifier, and in which all method arguments are initialized with values of the correct type.

Simple abstraction of the heap

Given the complexity of MIDP applications and taking into account every instance in the heap can also turn the complexity of the analysis unmanageable. Moreover, a more precise heap abstraction in the context of multithreaded programs can easily result in considering an exponential number of execution paths as we have to take into account all the possible interleavings between threads. On the other hand, if the abstract model discards completely the heap (e.g. every value written or read from the heap is considered to be anything) then we lose all the information and thus, the analysis becomes too imprecise. A compromise is to consider a simplified model of the heap where all the possible instance references in the heap of a particular Java class are collapsed to a single abstract instance. Thus, every element C in the set of references *AbsRef* stands for the object reference of the class C.

The heap abstraction h maps pairs of abstract references taken from *Ref* and field identifiers taken from the set $Field$ to an abstract value from the set $Value$:

$$h : AbsRef \times Field \to Value$$

The interpretation of the heap h is any concrete heap h' (where a concrete heap is a mapping from references and fields to concrete values) such that for every reference r of class C, $h'(r.f) \in h(C.f)$. The modelisation of heap update uses the operator updateHeap such that it erases the previously stored value for $r.f$ in h when the field f of the reference r is updated with the new value v:

$$\mathsf{updateHeap}(h', r.f, v) = h \oplus (r.f \to v)$$

Here, we take a different approach due to the way in which we abstract references, the aliasing and the presence of multithreading. We define the operator updateHeap to take the following effect on the abstract heap h when the field $C.f$ is updated with v:

$$\mathsf{updateHeap}(h, C.f, v) = h \oplus (C.f \to h(C.f) \sqcup v)$$

Such overapproximation is definitely much simpler than tracing separately every instance in the heap and is useful when objects of the same class are treated in a similar way, e.g. to check that a particular value is never assigned to some particular field $C.f$ in the heap for any instance of class C. Consider the example in Listing 2.1.

Listing 2.1: Illustration of heap approximation

```
  {h}
m(A a) {
   a.f = 2
  {h ⊕ (A.f → h(A.f) ⊔ 2)}
   a.f = 3
  {h ⊕ (A.f → h(A.f) ⊔ {2,3})}
}
```

The analysis[3] collects all possible values that any instance of class A may contain in the field f. Abstracting from the exceptional cases, this provides a sound and simple approximation for the heap when concrete instances are ignored and also is an information which can be useful when analysing programs executed in parallel with the method m. Consider that we have two methods m and n which are invoked in parallel in method main denoted as m || n

Listing 2.2: Example of multithreaded program

```
m(A a) {
   a.f = 2;
   a.f = 3;
}

n(A a, B b) {
   b.g = a.f;
}

main(A a, B b) {
  m(a) || n(a,b);
}
```

We ignore what value will be assigned to the field B.g, because of the possible interleaving between m and n. However, we can build a sound approximation of the value assigned to B.g if we collect all the updates done in the heap by the threads in the program.

Thus, the approximation of the heap is calculated by first collecting all possible write accesses on the heap done by the program threads. To do this, we should set initially the heap to the bottom heap function $\bot_h$ defined as follows

$$\bot_h(C.f) = defVal(C.f), \forall \text{ field } C.f$$

where $defVal$ returns the default value for $C.f$ which depends on the type of f (0 for integers and null for reference types). This is a sound approximation as the field of any newly created reference is initialised with this value.

However, before knowing all the information for the heap updates performed by every thread in the program, we neither know what is the current state of the heap - i.e. any read access is considered to return any value. But this implies that we should initialise the heap to the top heap function $\top_h$ defined as follows

$$\top_h(C.f) = \top_{\mathsf{Ty}}, \forall \text{ field } C.f \text{ is of type } \mathsf{Ty}$$

where $\top_{\mathsf{Ty}}$ is the top value of the type Ty (for any reference type Ty the top $\top_{\mathsf{Ty}}$ coincides with the unique value of this type $T \in AbsRef$). To reconcile these two opposite initial conditions on the heap, the heap model is composed of two components - h_w for write access and h_r for read access. Thus, the h_w and h_r are set initially respectively to $\bot_h$ and $\top_h$. Once all updates in the heap done by the program threads

[3]For the sake of simplicity and without losing generality, here abstract states do not mention local variables

are collected in h_w those updates are set accessible in a read mode by setting the read heap h_r to h_w. The iterations continue until h_r and h_w become the same. Finally, note that the heap is global, i.e. the heap is not approximated in every execution state separately, but rather one approximation is kept for the whole program. The heap approximation is thus computed by the following algorithm

Listing 2.3: Heap Analysis

```
Initially
  h_r := ⊤_h , h_w := ⊥_h
  program := thread_0 || ... || thread_n //analysed program
Repeat {
  For (0 ≤ i ≤ n) {
    h_w := analysis( thread_i , h_r , h_w )
  }
  h_r := h_w
  h_w := ⊥_h
}
Until h_r = h_w

h := h_r
```

Without further details, the function analysis analyses the code in $thread_i$ by a monovariant analysis over the methods that are executed in it.

For illustration, we show the steps of the analysis for the example from Listing 2.2

During the first pass, the read and write access heap are initialised to $\top_h$ and $\bot_h$, respectively following the specification of the algorithm. The algorithm iterates three times before reaching a fix point for the read and write components of the heap. From this we can infer that the value of the field g of any instance of class B may be 0, 2 or 3.

Here, we have not given much attention to the analysis in the case of exceptional termination. Actually, we treat it by systematically considering both exceptional and normal termination of execution. This of course, results in considering execution paths never taken. In the context of our current heap model, it would be possible to use a NullPointer analysis that can identify whether references of some class type may be null and to discard execution paths that are never taken.

Note finally that this approximation is sound but coarse, basically due to the fact that it does not distinguish between instances of the same class. The current analysis will infer for the example in Listing 2.4 that the possible values of the field B.g are 0, 1, 2. A more precise abstraction of the reference domain and the heap must be able to infer that the value of the field $B.g$ does not depend on the value of the instance reference assigned to the field A.a1.

Listing 2.4: Another example

```
public class A{
  int f;
  public A(int _f) {
    f = _f;
  }
}

public class C {
  A a1;
  A a2;
  B b;
```

<table>
<tr><th>Pass 1</th><th>Pass 2</th></tr>
<tr><td>$h_r := \top_h$
$h_w := \bot_h$

$\{h_r, h_w\}$
m(A a) {
a.f = 2;
$\{h_r, h_w \oplus (A.f \to \{0,2\})\}$
a.f = 3;
$\{h_r, h_w \oplus (A.f \to \{0,2,3\})\}$
}

$\{h_r, h_w \oplus (A.f \to \{0,2,3\})\}$
n(A a, B b) {
b.g = a.f;
$\{h_r, h_w \oplus (A.f \to \{0,2,3\}) \oplus (B.g \to \top_{\mathsf{int}})\}$
}
// $h_r \neq h_w$, so continue</td><td>$h_r := \bot_h \oplus (A.f \to \{0,2,3\}) \oplus (B.g \to \top_{\mathsf{int}})$
$h_w := \bot_h$

$\{h_r, h_w\}$
m(A a) {
a.f = 2;
$\{h_r, h_w \oplus (A.f \to \{0,2\})\}$
a.f = 3;
$\{h_r, h_w \oplus (A.f \to \{0,2,3\})\}$
}

$\{h_r, h_w \oplus (A.f \to \{0,2,3\})\}$
n(A a, B b) {
b.g = a.f;
$\{h_r, h_w \oplus (A.f \to \{0,2,3\}) \oplus (B.g \to \{0,2,3\})\}$
}

// $h_r \neq h_w$, so continue</td></tr>
<tr><th>Pass 3</th><th>Result of the analysis</th></tr>
<tr><td>$h_r := \bot_h \oplus (A.f \to \{0,2,3\}) \oplus (B.g \to \{0,2,3\})$
$h_w := \bot_h$

$\{h_r, h_w\}$
m(A a) {
a.f = 2;
$\{h_r, h_w \oplus (A.f \to \{0,2\})\}$
a.f = 3;
$\{h_r, h_w \oplus (A.f \to \{0,2,3\})\}$
}

$\{h_r, h_w \oplus (A.f \to \{0,2,3\})\}$
n(A a, B b) {
b.g = a.f;
$\{h_r, h_w \oplus (A.f \to \{0,2,3\}) \oplus (B.g \to \{0,2,3\})\}$
}

// $h_r = h_w$ and the analysis terminates</td><td>$\bot_h \oplus (A.f \to \{0,2,3\}) \oplus (B.g \to \{0,2,3\})$</td></tr>
</table>

Table 2.1: Analysis for the program in Listing 2.2

```
  public C() {
    a1 = new A(1);
    a2 = new A(1);
    b  = new B();
  }

  m(A a) {
    ...
    a.f = a.f + 1;
  }
  n(A a, B b) {
    b.g = a.f;
  }
  main(A a, B b) {
    m(a1) || n(a2,b);
  }
}
```

Modeling Java Strings

In Java, strings are immutable. Although they are references, they can also be considered as primitive values. Our first approximation of strings consists in considering a string as following pair:

- A determined concrete string
- A flag indicating whether or not the concrete string is followed by an undetermined number of characters.

Definition 2.3.1 *String domain* Σ
The abstract domain $(\Sigma, \sqsubseteq_\sigma, \sqcup_\sigma)$ *for non-null strings is defined as follows:*

* Σ is the set of couples $u = (s, c)$, where s is a concrete determined string and $c \in \{\epsilon, *\}$, indicating whether the string s is followed by undetermined characters $(*)$.
* $\top_\sigma = (\text{""}, *)$. It represents the empty string followed by undetermined characters.
* $(s_1, \epsilon) \sqsubseteq_\sigma (s_2, \epsilon)$ iff $s_1 = s_2$, for any $s_1, s_2 \in \mathit{String}$.
* $(s_1, c) \sqsubseteq_\sigma (s_2, *)$ iff s_2 is a prefix of s_1, for any $s_1, s_2 \in \mathit{String}$ and $c \in \{\epsilon, *\}$.

Chapter 3

Work in the context of MOBIUS

3.1 Objectives

The technology described in Chapter 2 is powerful enough to verify simple security requirements (as defined in [3]), and in particular the requirements related to API usage and some of the API usage restrictions patterns. Some early experiments conducted with a large set of publicly available applications have shown that these algorithms are able to identify a large proportion (between 75% and 90%) of the uses made of the target APIs. However, since an application can use these APIs several times, this proportion reduces significantly when applied to entire applications.

The objective for TL in MOBIUS is to improve these results in two directions: by enhancing the precision of the existing analysis domains, and by adding new domains, in order to address new security requirements. Before results from MOBIUS research can be included into the prototype, there is a lot of preparatory work to be performed. Therefore, the first half of the work on the prototype has been mostly devoted to preparation work:

- Enhancing precision of existing domains, section 3.2.
  In the context of Java mobile applications, the ability to determine as much as possible the elements of a connection URL is critical. In particular, this section describes the enhancement of the abstract domain for the strings.

- Multiple domains management, section 3.3.
  In order to integrate new domains, and in particular those developed in the MOBIUS project, the type-level abstract interpretation engine needs to be enhanced to manage additional domains in parallel to the basic abstract value domain.

- Information flow domain, section 3.4.
  This section describes the experiments w.r.t. the management of an additional domain for information flow properties.

- Optimisations in the monovariant analysis, section 3.5.
  We describe here a few improvements of the analysis consisting in a more elaborated order of method traversal.

- New model of the memory heap, section 3.6
  For being able to trace more fine grained properties, the MIDP analysis presented in the previous section should be upgraded with a more precise model of the memory heap. Here, we present such model which basically takes into account object references.

- Capitalization of the Java Card experience, section 3.7.
  The last section of this chapter describes future experimentations issued from the Trusted Logic experience with the Java Card framework.

3.2 Enhancing the precision of existing domains

In the beginning of the MOBIUS project, TL started some exchanges with other partners in order to enrich the basic prototype with state-of-the-art research on typing. We focused in particular on the string domain described in chapter 2, because it is at the heart of many security requirements. In MIDP, the information about established connections is very important, because this is how all the communication with the outside is initialized, and this also represents the main origin of billing from a mobile application. All connections are made through Java's Generic Connection Framework (GCF). In fact, all attempts to open a connection are made through the `Connector.open` static method, regardless of the connection type. This method takes a URI parameter, with the following structure:

scheme `:` *hierarchical part* [`?` *query*] [`#` *fragment*]

Many security requirements can only be enforced after determining at least the scheme part, which identifies the type of the connection. For example, it is very important to differentiate a low-level connection (as socket connection, datagram connection, *etc.*) to a high-level connection (as HTTP connection, HTTPS connection, *etc.*). One way to do so is to check the way in which the resulting `Connection` object is used, and in particular how it is cast after its creation; however, this technique has some limits, because this is not a requirement, and some applications may cast connections into very generic types, which brings little information. The other alternative is to determine the value of the scheme part of the URI. Other parts of the URI are also quite important in other contexts, for instance the hierarchical part when analyzing SMS target numbers, and the query when analyzing Web services vulnerabilities.

On the initial implementation Σ of the string domain, an abstract string is represented by a unique value, which is the longest prefix common to all actual values represented by the abstract value. In many simple cases, this value is $\top_\sigma$, the undetermined value for the domain. Typically, a URL can be initialized with a few different values, with little in common, and the analysis loses all the information in such cases. This happens for instance in the following code example:

```
String v;                          (1)
if (a_test)                        (2)
  v=''sms://85531'';               (3)
else                               (4)
  v=''sms://30553'';               (5)
Connector.open(v);                 (6)
```

On this case, the static analyser will produce two states for the string `v` :

- One with an abstract value set to (‘‘`sms://85531`’’, ϵ)
- One with an abstract value set to (‘‘`sms://30553`’’, ϵ)

As a result, when reaching the actual connection opening at line (6), the two possible values are merged, and the result of this merge is

(‘‘`sms://`’’, ϵ)

, since the two phone numbers have nothing in common. In such a case, no information is inferred about the number.

This situation is quite common in applications. When that occurs, the new domain extends the initial domain with a set of string values instead of a unique String value. This extended domain increases the precision for the static analysis of strings, but it is more complex. Indeed, all operations on strings have to be redefined. For example, let's consider the `String.concat` operation (+). With the initial implementation, it is very easy to simulate this by:

- $(s_1; *) +_\sigma (s_2; c) = (s_1; *), \forall s_1; s_2 \in \mathit{String}$ *and* $c \in \{\epsilon, *\}$
- $(s_1; \epsilon) +_\sigma (s_2; c) = (s_1 s_2; c) \forall s_1, s_2 \in \mathit{String}$ *and* $c \in \{\epsilon, *\}$, where $s_1 s_2$ is the string concatenation.

But with the extended domain, this simple operation becomes more complex. Indeed, now a concatenation is a cartesian product of all values in the two sets. We can consider an example:

$$\begin{aligned} & \{(s_{11}, \epsilon), (s_{12}, *), (s_{13}, \epsilon)\} + \{(s_{21}, *), (s_{22}, \epsilon)\} \\ = \; & \{(s_{11}s_{21}, *), (s_{11}s_{22}, \epsilon), (s_{12}, *), (s_{12}, *), (s_{13}s_{21}, *), (s_{13}s_{22}, \epsilon)\} \\ = \; & \{(s_{11}s_{21}, *), (s_{11}s_{22}, \epsilon), (s_{12}, *), (s_{13}s_{21}, *), (s_{13}s_{22}, \epsilon)\} \end{aligned}$$

With this extended domain, it is possible to verify more properties relying on string values.

3.3 Multiple domains management

The initial implementation of the type analysis tools, as explained previously, is based on Trusted Logic's GJVM (Generic Java Virtual Machine). This GJVM allows type-level abstract interpretation of a Java virtual machine. For CLDC (and MIDP) application, the static analyser algorithm is monovariant and context-insensitive and works on a single and simple domain, strictly based on an approximation of the concrete values.

However, there are many cases in which these basic values can be complemented by additional data. One example is information flow data, which abstract the origin/sensitivity of information rather than its actual value. This kind of information is not directly related to the value domains, and is even orthogonal to these domains; such information is interesting for all value domains, but their analysis does not interact with the analysis of values.

Information flow analysis is interesting, but there are more potentially interesting analyses, as the work done in MOBIUS shows. We have therefore chosen to introduce in the tool the management of multiple domains in the GJVM. The objective is to be able to combine different analyses, and also to facilitate the future integration of new algorithms. The choices made in this implementation are as follows:

- Each abstract value can be associated to an unlimited number of domains.
- Domains do not need to be orthogonal (the result of some operations may depend on two distinct domains).
- One domain (here, the basic value approximation domain) will be considered as the main domain, with the other ones being considered as attributes of this domain.

The definition of a new domain is a lot of work. The minimal investment consists in defining all the abstract operations on the domain, and also to model the various APIs for the new domain. In order to make this task lighter, some refactoring has started and will continue, in particular in order to model the APIs through the basic abstract operations, making it easier to develop new domains. The process nevertheless remains costly, and the only application of multiple domain management is currently the implementation of the information flow domain.

3.4 Information flow domain

As defined in the security requirements, information flow is useful in MIDP, in a fairly simple way. The objective is here to control the use of sensitive data, which is not necessarily entirely confidential. As a first approximation, we have here chosen to consider only explicit information flow, and to reason on the origin of data rather than on the confidentiality of the information. Since there are several possible origins for an information (code, user entry, network, *etc.*), this domain therefore is a composition of several domains.

For each possible origin, the abstract domain is simply an abstraction of a simple binary value (0 or 1), and we will use the following simple values and abstraction $0_\phi = \alpha(\{0\})$, $1_\phi = \alpha(\{1\})$, $*_\phi = \alpha(\{0, 1\})$, and $\emptyset_\phi = \alpha(\{\})$. The basic abstract operations are the direct mappings of the set operations.

We can then define the abstract domain $(\Phi, \sqsubseteq_\phi, \sqcup_\phi)$ on the data flow control as:

- Φ is a set of n-tuples (n is the granularity of the control flow), with each tuple is composed by n values representing the abstraction of a given origin.
- $\top_\phi = \langle *_\phi, \ldots, *_\phi \rangle$ is a special element of Φ representing the unknown data flow control (generalest value). This value appears if the information flow is lost or if no hypothesis can be inferred on the data flow.
- $\bot_\phi = \langle \emptyset_\phi, \ldots, \emptyset_\phi \rangle$ is a special element of Φ representing the undefined data flow control. This abstract value may have two meanings: either that its holder has not been initialised with a concrete value, or that this branch of analysis is in fact unreachable code.
- Partial order:
- $\forall x \in \Phi, x \sqsubseteq_\phi \top_\phi$
- $\forall x \in \Phi, \bot_\phi \sqsubseteq_\phi x$
- $\forall x = \langle x_1, \ldots, x_n \rangle, y = \langle y_1, \ldots, y_n \rangle \in \Phi$,
  $x \sqsubseteq_\phi y$ if and only if $\forall i, x_i \subseteq y_i$
- Join:
  $\forall x = \langle x_1, \ldots, x_n \rangle, y = \langle y_1, \ldots, y_n \rangle \in \Phi$,
  $x \sqcup_\phi y = \langle x_1 \cup y_1, \ldots, x_n \cup y_n \rangle$

In the rest of the section, we provide two examples of the use of this domain. Each example starts with the Java code to analyze, followed by the different stages of the analysis, and some examples of properties that can be verified with the proposed analysis.

3.4.1 Simple example without multiple origins

For this example, we define the Φ domain as defined last with three possible origins:

- `CODE`, this origin is for all values that are created in the source code of the application (for example with the bytecode LDC).
- `NET`, represents each values that are provided by a network interface (like a HTTP connection).
- `OTHER`, represents all the other dataflow values.

Each variable is associated with a triplet $\langle$`CODE`, `NET`, `OTHER`$\rangle$ with the convention `X` for a "true" value and $\neg$`X` for a "false" value (`X` $\in$ {`CODE`,`NET`,`OTHER`}).

We can note:

- α the abstraction function for this domain.
- $\overbrace{\texttt{getAppProperty}}(\langle x_1,\ x_2,\ x_3 \rangle) = \langle x_1,\ x_2, \texttt{OTHER} \rangle$ the abstract function of `getAppProperty`.
- $\overbrace{\texttt{Connector.open}}(\langle x_1,\ x_2,\ x_3 \rangle) = \langle x_1,\ \texttt{NET},\ x_3 \rangle$ the abstract function of `Connector.open`.

Java code The example is voluntarily extremely simplified, just like the abstraction is:

```
String address = ''sms://+358401234567:6578'';
MessageConnection smsconn  = Connector.open(address);
...
```

Analysis

- We define: $\alpha(\texttt{"sms://+358401234567:6578"}) = < \texttt{CODE}, \neg\texttt{NET}, \neg\texttt{OTHER} >$
- So $\alpha(\texttt{address}) = < \texttt{CODE}, \neg\texttt{NET}, \neg\texttt{OTHER} >$
- Then $\overbrace{\texttt{Connector.open}}(\alpha(\texttt{"sms://+358401234567:6578"})) = < \texttt{CODE}, \texttt{NET}, \neg\texttt{OTHER} >$
- Thus $\alpha(\texttt{smsconn}) = < \texttt{CODE}, \texttt{NET}, \neg\texttt{OTHER} >$

Few examples of rules The following rules can be checked in that case:

- The origin of the connection String parameter is from the code. To conclude this, the analysis should approximate the parameter of the `Connector.open` method with the abstract value CODE.
- The origin of connection String parameter is coming from the net. In this case, the parameter of the `Connector.open` method is approximated with NET.

3.4.2 More complex example (with a join)

For this section, we define the same Φ domain as defined last with three possible origin. We consider the following code fragment

Java code

```
if (prop == null)
  address = ''sms://+358401234567:6578'';
 else
  address = prop.getAppProperty(''property'');

Connector.open(address);
...
```

Analysis result By applying the analysis on the code fragment, we get:

- If `prop` is `null`:
  - We define
    $\alpha(\texttt{"sms://+358401234567:6578"}) = \langle \texttt{CODE}, \neg\texttt{NET}, \neg\texttt{OTHER} \rangle$
  - Thus $\alpha(\texttt{address}) = \langle \texttt{CODE}, \neg\texttt{NET}, \neg\texttt{OTHER} \rangle$
- If `prop` is not `null`:
  - We define:
    $\alpha(\texttt{''property''}) = \langle \neg\texttt{CODE}, \neg\texttt{NET}, \texttt{OTHER} \rangle$
  - So $\overbrace{\texttt{getAppProperty}}(\texttt{''property''}) = \langle \neg\texttt{CODE}, \neg\texttt{NET}, \texttt{OTHER} \rangle$
  - Thus the approximation of `address` after the if statement is $\alpha(\texttt{address}) = \langle \neg\texttt{CODE}, \neg\texttt{NET}, \texttt{OTHER} \rangle$

With a final join operation on $\alpha(\texttt{address})$ we obtain:
$\alpha(\texttt{address}) = \langle \texttt{CODE}, \neg\texttt{NET}, * \rangle$

3.4.3 Examples of rules which this domain can handle

This kind of information flow domains is useful for the rules that apply on the origin of data. A set of examples of these types of rules (in [3]) is:

G-3 : The resources are used according to their type.
This rule ensures that there is no hidden content in files attached as resource to the application.

G-73 : The application does not publish PIM information.
This allow to certify that an application doesn't reveal confidential information even if a connection is open.

G-75 : The application does not modify existing records.
Guarantees that at least nothing will be destroyed in the personal diary (but it may be polluted).

G-79 : The application only creates players with local resources.
Avoid network usage abuse with unexpectedly big resources.

3.4.4 Present limitations

Our first attempt to define an information flow domain was quite imprecise mainly for these reasons:

Lack of precision. The analysis is context-insensitive as well as the heap representation which is quite simplified. This leads to many unknown values in all domains, including the information flow domain. While the performance issues impede us from adopting a context sensitive analysis, we have investigated a more sophisticated memory model (see section 3.6) which provides much more precision than the model that we presented in the previous chapter.

Implicit information flows. The present analysis does not take into consideration the implicit information flows. However, the GJVM includes tools that would allow us to take them into consideration, as it is in particular able to associate a hypothesis with each state, and to manage this hypothesis in the abstract computations. The current implementation also includes a control flow analysis, which is used to build a call graph and to detect loops, which could be used for the analysis of implicit information flows.

3.5 Improvements of the monovariant analysis

As discussed in the preliminary presentation of our tool in Subsection 2.3, we use a monovariant analysis which traverses all methods separately in arbitrary order. This technique scales up for realistic programs but still there are ways to make it even more efficient. Here, we discuss two such possibilities.

Method analysis based on method ordering. Traversing methods in arbitrary order scales up well but still the algorithm can be faster if we use a more fine grained order. Instead of picking blindly a method, we can take into account the order of invocations. Such an approach allows to analyse a method only when information about its invocation input is available (note that the analysis does not inline invoked methods but inspects methods which are known to be invoked). This is not possible in general as Midlet applications interact with the execution environment via a callback mechanism. As an example, consider the example in Listing 3.1 where the class C implements the Midlet interface, i.e. provides implementation of the methods startApp, pauseApp etc. Those methods are the entry points of the Midlet and can be called by the application system manager (ASM) in any order. However, the order in which other methods are invoked by such entry point methods can be discovered from the program text statically. We can follow the call graph starting at every such entry point method and analyse methods in the proper order of their invocations (in the sequential case). For the example in

Listing 3.1, starting for the call back method startApp, we analyse in the order of invocation methods m and p. Note that because we follow the order of invocation, on the first pass through the methods m and p, we will have already a first approximation of the values passed as arguments. In the case of arbitrary traversal, the method m can be potentially inspected at the first iteration with the undefined abstract value which is useless if the returned value depends on the methods' arguments. Note also that this can prevent us from inspecting and identifying dead code. Finally, the same technique is used for event based callback methods, i.e. methods of the Midlet which are called by the ASM when some event (e.g. clicking a button) occurs and for any thread launched by the application.

Using the Midlet life cycle. On the other hand, we can also gain precision if we take into account the order of method execution determined by the Midlet life cycle. In particular, when a midlet application is installed on a MIDP platform, the platform first creates an instance of the class which implements the interface Midlet and initialises it by invoking the class' constructor. Once initialised, the midlet is ready to interact with the ASM via the entry point methods of the Midlet interface startApp, pauseApp etc. By preliminary analysis of the code in the midlet constructor, we can analyse the rest of the code starting with more precise approximation of the heap. For instance, for the code in Listing 3.1, we would know that the heap is already initialised with the value 1 whenever one of the callback methods is invoked.

Listing 3.1: Midlet example

```
public class C implements Midlet {
  int f;
  public class C() {
    this.f = 1;
  }
  public void startApp() {
    ...
    int s = 3;
    m(s);
  }
  public void pauseApp() {
    ...
  }
  private void m(int k) {
    p(k+2);
    ...
  }

  private void p(int k) {
    ...
  }
  ...

}
```

3.6 Memory heap

Although the heap abstraction presented in Section 2.3 provides useful information about the memory heap, it is still quite imprecise, since it gives a minimal support for reference values where all references of one class C are identified into one abstract reference. To improve this, we should chose a suitable level of abstraction

so that not to compromise the performance of the analysis, i.e. the speed of convergence to the fix point solution. We define the set of abstract references Ref such that every element element r^m_{pc} stands for the concrete references which point to objects allocated in method m at program index pc. Next, we introduce the domain $(AbsRef, \subseteq, \cup)$ of abstract reference values whose elements are either subsets of the union of the set Ref and the singleton containing the special reference null or the undetermined value $\top_{AbsRef}$

$$AbsRef = \{r \mid r = \{r_1 \dots r_k\} \text{ where } \{r_1 \dots r_k\} \subset Ref \cup \{\mathsf{null}\} \vee r = \top_{AbsRef}\}$$

The order relation $\subseteq$ is the subset relation where $\top_{AbsRef}$ is greater than any other element in $AbsRef$. The joint operator $\cup$ is the set union where the joint of $\top_{AbsRef}$ and any other element in $AbsRef$ results in $\top_{AbsRef}$. The reference $\top_{AbsRef}$ stands for the undetermined reference value, basically an abstract value is approximated to $\top_{AbsRef}$ if the analysis discovers that the abstract reference does not converge (i.e. it may represent too many references).

In the new setting, the heap abstraction h has a domain $Dom(h)$ which represents the set of references pointing to allocated objects in h and such that $Dom(h) \subseteq Ref$. Moreover, h is a mapping from pairs of references and fields to abstract values (taken from the set $Value$)

$$h : (Dom(h) \cup \top_{AbsRef}) \times Field \to Value$$

Intuitively, the heap h is the abstraction of any physical heap h' such that for every reference r allocated at index pc in method m, $h'(r.f) \in h(r^m_{pc}.f)$.

By definition any field f of the top reference $\top_{AbsRef}$ is mapped to the top abstract value (of the respective type of the field) i.e. $h(\top_{AbsRef}.f) = \top$. This means that if we can not infer anything specific about a reference value then we neither know much about its fields.

The mapping from an abstract reference $\{r_1 \dots r_k\}$ to abstract field values is defined as the joint of the abstract field values :

$$h(\{r_1 \dots r_k\}.f) = \sqcup_{i=1}^{k} h(r_i.f)$$

The operator $\mathsf{new}(h, A, pc, m)$ models the allocation of a new instance of class A in the heap h at instruction at index pc in method m. This parametrisation of references with the index of the program counter and the method name[1] will limit the size of the heap and allow for convergence to the solution after a finite number of iterations. If the domain $Dom(h)$ contains already the reference r^m_{pc}, then the operator returns the same heap and the reference r^m_{pc}. Otherwise, the result is the heap $h \cup r^m_{pc}$ and the reference r^m_{pc}. The domain of the resulting heap $Dom(h \cup r^m_{pc})$ is the same as $Dom(h)$ but extended with the fresh reference r^m_{pc} where every field f of r^m_{pc} is mapped to the default value $defVal(f)$

$$\mathsf{new}(h, A, pc, m) = \begin{cases} (h, r^m_{pc}) & \text{if } r^m_{pc} \in Dom(h) \\ (h \cup r^m_{pc}, r^m_{pc}) & \text{else} \end{cases}$$

where $Dom(h \cup r^m_{pc}) = Dom(h) \cup r^m_{pc} \wedge$
$\forall r : Ref \;\; f : Field, r \neq r^m_{pc}, h(r.f) = h \cup r^m_{pc}(r.f) \wedge$
$\forall f : Field, h \cup r^m_{pc}(r^m_{pc}.f) = defVal(f)$

The update operator follows the same idea as described in Section 2.3, i.e. if a field reference is updated then the new value is the joint of the previous approximation and the new value

$$\mathsf{updateHeap}(h, r.f, v) = h \oplus (r.f \to h(r.f) \sqcup v)$$

We abuse of the notation and define the update operator for an abstract reference $\{r_1 \dots r_k\}$ by applying the update for every concrete reference $r_i, i = 1 \dots k$

[1] We assume that method names contain the class where they are declared and thus, a method name uniquely identifies a method

$$\mathsf{updateHeap}(h, \{r_1 \dots r_k\}, v) = h \oplus (r_i.f \to h(r_i.f) \sqcup v)_{i=1}^{k} \text{ where } \{r_1 \dots r_k\} \subset \mathit{Ref}$$

Abstract states are defined as usual as a pair of a mapping lv from local variables to their respective abstract values and a heap h i.e. $\{lv, h\}$. Finally, the heap is global for the whole program as in our previous model presented in Section 2.3

Example for normal execution For this first example, we shall ignore here the possible exceptional termination of programs and in particular, we consider that references are always initialised before being accessed and so, we shall ignore the null value. Let us have a class A with an instance field f which is initialised by the constructor to the value passed as argument to the constructor

```
class A{
  int f;
  public A(int t) {
    f = t;
  }
}
```

Then for the following program fragment in one iteration the analysis will infer the following information about the intermediate abstract states starting with a heap h

```
1  public void m(A a) {
2
3      {lv, h}
4    if (b) {
5      a = new A(1);
6      {lv ⊕ (a → a_5^m), h ∪ a_5^m(⊕(a_5^m.f → {0,1})}
7    } else {
8      a = new A(3);
9      {lv ⊕ (a → a_8^m), h ∪ a_5^m ∪ a_8^m ⊕ (a_5^m.f → {0,1}) ⊕ (a_8^m.f → {0,3})}
10   }
11
12     {lv ⊕ (a → {a_5^m, a_8^m}), h ∪ a_5^m ∪ a_8^m ⊕ (a_5^m.f → {0,1}) ⊕ (a_8^m.f → {0,3})}
13   k = a.f;
14     {lv ⊕ (a → {a_5^m, a_8^m}) ⊕ (k → {0,1,3}), h ∪ a_5^m ∪ a_8^m ⊕ (a_5^m.f → {0,1}) ⊕ (a_8^m.f → {0,3})}
15 }
```

The first observation that we can make is that the abstract value of the field $a_5^m.f$ (line 6) is approximated by 0 and 1. This is because we have collected the default value 0 assigned to the field just after the creation of the instance and the value 1 assigned to the field in the constructor. Indeed, assuming that the whole heap may be modified by all the threads, from the point of view of a thread running in parallel to the method m, the field f may have value 0 or 1, depending on the thread interleavings. We can also note that the abstract execution infers an overapproximation for the value stored in the variable k (line 14) is either 0, 1 or 3.

Notice that for the example in Listing 2.4 from Section 2.3, the present analysis provides more precision and reports that the possible values for the field b.g are 0 or 1.

Exceptions Now, we return to the more realistic scenario with exceptions and null values. To illustrate how exceptional termination is treated let us consider a variation of the previous program. This time we consider that we just initialise the local variable to a new instance of type A (line 3) and then read its value to assign it to the local variable k

```
1  public void m(A a) {
2        {lv, h}
3     a = new A(3);
4        {lv ⊕ (a → {a_3^m, null}), h ∪ a_3^m ⊕ (a_3^m.f → {0,3})}
5     int k = a.f;
6        //state in the normal case
7        {lv ⊕ (a → {a_3^m}) ⊕ (k → {0,3}), h ∪ a_3^m ⊕ (a_3^m.f → {0,3})}
8
9        //state in the exceptional case
10       {lv ⊕ (a → {null}), h ∪ a_8^m ∪ n_5^m ⊕ (a_3^m.f → {0,3}), exc = n_5^m}
11 }
```

As in the previous case, because of the multithreading context even after the assignement at line 5, the approximation of the variable a says that it still might be null. This is a sound approximation. However, it may lead to considering spurious normal and exception execution paths some of them never taken especially in the case when objects do not escape the scope of the current thread (consider that we know that the reference contained in a is accessed only by the current thread, then we know for sure that interleavings with other threads do not influence its value) or when a reference is never null. Therefore, instead we consider the two possible outcomes of the execution. For this, we use a simple optimisation and split the values of the parameter variable a into null and non null values for the two possible terminations of the execution. This optimisation is possible for parameters and local variable, but note however that for heap elements (instance or static fields) this is not possible as we assume a multithreading context in which they can be accessed concurrently. For the normal execution we continue with a state where the variable a is mapped to the new instance and for the exceptional case, we consider a state where the variable a is mapped to null. We can also remark that the heap in the case of the exceptional termination is augmented with a new NullPointer exception reference which follows the semantics of the Java Virtual Machine. Note that we also use the special variable *exc* internal to our analysis which stores the new NullPointer exception instance n_5^m thrown at line 5.

3.7 Leverage of Java Card experimentations

The current MIDP analysis remains much simpler than the initial Java Card analysis, mostly for performance reasons. However, the performance of the current tool remains quite good, and there is some leeway to make the analysis more complex, and get better results. Several directions are being investigated, which could become interesting.

Heap abstraction. The new heap abstraction provides much more precision than the previous abstraction which completely discarded references.

As we introduce a fixpoint algorithm, it is tempting to introduce simultaneously some level of inter-procedural analysis, by maintaining a list of the call patterns for all the methods in the program. Some initial experiments with such an enhanced analysis has shown that the precision of the results can be significantly enhanced, while the complexity increase remains acceptable. Also, supporting escape analysis can also improve the precision of the analysis by approximating better references that never go beyond the scope of the thread that created them.

Undoubtfully, a NullPointer exception analysis can also improve efficiency as the result that a reference is not null can exempt us from inspecting dead exception paths.

In addition, the memory abstraction may also be extended to cover the persistent stores (at least the private ones), in order to be able to keep some information about the data stored persistently. In particular, information flow data can be kept fairly easily. This is quite important, because persistent stores are heavily used in MIDP in order to store user preferences, and some knowledge about these preferences is often required for the analysis.

Hypothesis analysis. Another interesting direction is to make a better use of conditional branches, in particular by considering the result of a test that led to a conditional branch during the analysis of that branch. There are several applications of this analysis:

- The information can be used to increase the precision of the analysis, for instance by taking into consideration tests that are performed as a prefix to a sensitive operation.
- The hypothesis information can also be used as a way to better support implicit information flows.

Adaptation to Java Card. Finally, another area of work is to adapt these algorithms back to the analysis of Java Card programs. There are several reasons for this adaptation:

- Java Card applications are becoming quite large, and the detailed analysis will soon be too expensive. It is therefore necessary to improve the efficiency of the analysis of Java Card applications, and the execution model used for MIDP applications is a good candidate.
- As mobile security solutions are being defined, it becomes obvious that programmable secure tokens often are at the heart of the solution. Java Card applications are therefore often co-designed with mobile applications, and they also need to be evaluated.

Another advantage of retrofitting this work to Java Card applications would be that it would increase the maintainability of the code base, by reducing the differences between the various analyses.

Chapter 4

Adaptation of MOBIUS type systems

4.1 Introduction

The objective of Task 2.6 reported here is to evaluate the integration of the type systems elaborated in the scope of workpackage 2 (see [4]) in an industrial environment. For this, several major factors appear to be crucial:

- The primary target for the type systems is the MOBIUS prototype, which will integrate all advanced research work, and it focuses on the possible verification of certificates as part of a PCC scheme.
- The MOBIUS type systems have been designed for a very wide range of devices, including very small devices. Some type systems, in particular those related to resources, are not crucial for MIDP applications, but they are crucial for programs that run on smaller systems, such as those based on Java Card technology.
- The static analysis tool aims at working with unmodified standard applications. It therefore needs to infer the typing of programs, which is much more difficult than simply verifying a type certificate.

The objective of the present chapter is to identify how the TL tool tailored to MIDP can benefit from the type systems as well as the constraints that the type systems should take into account in order to facilitate integration in the tool. For this, we use the following criteria:

Architectural compatibility. It is not possible in the timeframe of the MOBIUS project to design and implement a new framework. Therefore, a crucial criteria is the assessment of the compatibility of a type system with the architecture of our current framework.

Coverage of features. In order to be efficient in an industrial setting, a type system must cover all important features of the Java ME framework, including in particular multithreading and exception management.

Modeling features. In an industrial tool, it is required to model all the APIs, so there must be efficient ways to do so. In particular, type systems for which APIs are difficult to model will be difficult to adapt.

Usable on standard MIDP applications. An industrial product needs to be usable on real-life applications, using all features of the underlying framework in the "standard" way.

Ease of inference. The industrial product is intended to be used on unmodified applications, for which the typing must be inferred, so the type system must allow this.

Performance level. The performance of the type system must be sufficient to motivate its integration in the product. In particular, the type system must provide interesting and usable information.

Coverage of requirements. As the tool is expected to be used for the verification of operator-provided security policies, a type system must cover some of the framework-specific security requirements defined in Chapter 3 of Deliverable D1.2.

With the help of this assessment, it should be possible to derive future work items for both type system designers and prototype implementers, which are developed in the final section.

4.2 Information Flow Security

4.2.1 Description

The information flow type systems are focused on noninterference properties. Usually, a program is said to be noninterfering if it does not leak information of a secret data to an external observer. Such property is formalised as the fact that the secret data can not influence in any way the non secret data which is externally observable and thus, may allow the observer to guess the secret (or deduce some information about it). For this, program variables are assigned with a security level - high for variables that contain secret data and low otherwise. The basic assumption there is that an attacker may observe only the input and output of low program variables. The type systems that have been developed in Mobius address the preservation of noninterference in two scenarios

Sequential programs The type systems deal with both explicit and implicit noninterference. Explicit flow occurs when values of high variables can be assigned to low variables while implicit typically happens when under a condidtional test on the value of a high variable, a low variable is updated and thus may reveal the high value (i.e. secure information) to an external observer.

Multithreading In this case, information flow from high to low level variables may be caused by a particular feature of the scheduler algorithm. The solution consists of a type system for multithreaded programs and constraints on the algorithm for scheduling threads. The type system now assigns security levels not only to program variables but also to threads. The constraints on the thread scheduler require basically that it should be sensitive to threads' security level. The type system guarantees that no timing leaks are possible under the assumption that the scheduler supported by the execution environment respects the constraints.

In both of the scenarios, the type system requires extra security annotations, but it is possible to generate most of these annotations through an appropriate static analysis (note that there is always a minimal human effort to decide which variable contains a secret and which not). In the context of the TL tool, the typing judgment can be implemented as an extension of the virtual machine which will handle security environments for each program point: this could be represented by a security-level abstract interpretation.

4.2.2 Suitability

We here review the information flow type systems in Mobius with respect to the criteria defined above:

Architectural compatibility. There is no major issue at this level, except for the inference of the control dependence regions information, which requires an adaptation of our existing control-flow analysis.

Coverage of features. The type system covers a large part of the sequential features of the Java Virtual Machine - object manipulation and creation, the exception mechanism, method invocation, operand stack manipulation etc. Note that the type system addressing multithreading is not directly applicable for MIDP applications as it works under a particular design of the execution environment - namely, a scheduler which is aware of the security level of threads. In this sense, this algorithm can not be applied directly to our tool as the assumptions for the scheduler will not hold.

Modeling features. Modeling may be an issue, as the implicit flows will need to be modeled. This may lead to a large amount of work or to oversimplification, leading to poor precision.

Usable on standard MIDP applications. The type system is presented as a way to verify an application with respect to a table of method signatures, but it should be possible to only provide as objectives the signatures of the API methods.

Ease of inference. No specific issue.

Performance level. Initial results seem to indicate that the proposed type system is too restrictive, and leads to the rejection of all real-life applications.

Coverage of requirements. The type system is suitable for many information flow requirements, but it needs to be adapted in order to be less restrictive.

The proposed type system is interesting because of the way in which it models implicit information flow. The main issue is here that the type system has been defined too strictly. If we consider for instance the first example considered in section 2.2.4 of D2.1, the fact that the value of a confidential reference may be null is considered like a potential issue. In our analysis, it is extremely difficult to discard the null value for a given field. This single example could lead to the rejection to a large proportion of programs.

4.2.3 Interesting improvements

As we remarked already, the type systems are quite restrictive in the sense that they may reject programs which may contain information leaks but which we consider harmless. In this sense, it will be useful to study techniques which make more liberal the typing algorithms, e.g. declassification which could bring interesting results.

4.3 Basic Resource Policies

The first release of resource type systems is concentrated to the analysis of:

- Heap space consumption
- Access permissions
- Accounting external resources
- Estimating execution time

A separate type system has been defined for each analysis, and each one is described in a section below.

4.3.1 Heap space consumption

The heap space consumption type system ensures a constant bound on the heap consumption of bytecode programs.

Architectural compatibility. No major issue.

Coverage of features. Currently supports only a subset of the Java bytecode.

Modeling features. Not an issue.

Usable on standard MIDP applications. This is the major issue, since real-life MIDP applications tend to allocate a potentially infinite amount of memory, as they heavily rely on garbage collection.

Ease of inference. No major issue.

Performance level. Because of the garbage collection issue, the performance is expected to be very bad on real-life mobile applications.

Coverage of requirements. The requirements about heap consumption are not explicit, as they are more related to portability than security.

Obviously, the major issue is here that this particular analysis is not applicable to real-life MIDP applications, which usually allocate memory in a potentially unbounded way, and rely on the garbage collector to manage their memory consumption. In order to be practical, this analysis must be combined with a reachability analysis, which determines whether or not objects remain reachable after their initial use.

4.3.2 Access permissions

INRIA has proposed an enhanced security model, which improves on the current MIDP architecture, in which applications can request multiple permissions in advance (instead of requesting a global and unlimited permission), and the model can be statically enforced. They also have proposed a static analysis for it.

Architectural compatibility. The analysis is based on an inter-procedural analysis, which could be an issue with our current tool.

Coverage of features. Needs to be adapted to full Java. In particular, loops are not covered.

Modeling features. Not a major issue, as resource usage is well-known.

Usable on standard MIDP applications. This is an issue, since the analysis is based on an extension of the MIDP security model, which is not assumed by actual applications.

Ease of inference. No inference seems possible, since the model is an extension that assumes that the developer declares more precise properties.

Performance level. No information yet.

Coverage of requirements. This analysis could be very interesting, because there are many security properties that are based on resource usage.

Once again, the major issue is practical, since the analysis is not applicable to real-life MIDP applications. The fact that the model is an extension is not necessarily a major issue, since the permissions could be granted at the beginning of the program, based on the proof requirements. On the other hand, the analysis as it is defined today is not precise enough; since resource usage is very diverse in MIDP, a simple control flow analysis cannot be expected to identify all possible issues, and a more precise analysis, in particular of loops, will be required.

4.3.3 Accounting external resources

This type system is an alternative to the system proposed by INRIA, in which a resource manager mechanism is used to verify dynamically that resources have been properly allocated. The system includes static checks that make it unnecessary to actually use the dynamic mechanism.

Architectural compatibility. The analysis requires a logic support that our framework does not offer.

Coverage of features. Needs to be adapted to full Java.

Modeling features. Not a major issue, as resource usage is well-known.

Usable on standard MIDP applications. This is an issue, since the analysis is based on an extension of the MIDP security model, which is not assumed by actual applications.

Ease of inference. Inference requires logic reasoning, which is not possible with our current tool.

Performance level. No information yet.

Coverage of requirements. This analysis could be very interesting, because there are many security properties that are based on resource usage.

In this analysis, there are two main issues. The first one is practical, and is related to the fact that it is not applicable to real-life MIDP applications. However, some work is planned on type inference, which could lift this restriction by making it possible to automatically generate the non-standard code.

The other issue is that the current framework offered by our type analysis tool is not sufficient for this analysis, as it does not include the required logic tools. In particular, it is not able to handle constraints on the domains required.

4.3.4 Estimating execution time

The objective of this type system is to estimate the execution time of an application. It computes an upper bound on the number of basic operations that are executed by a program, and then combines these numbers with an estimation of the time required for each basic operation.

Architectural compatibility. This analysis has not been fully studied yet.

Coverage of features. Needs to be adapted to Java bytecode.

Modeling features. Not completely relevant, as many operations depend on external issues and would need to be ignored or discarded.

Usable on standard MIDP applications. The fact that the most time-consuming operations are API methods, whose execution time is very difficult to estimate, make the application to real applications difficult. In addition, execution time is not a major constraint for MIDP applications.

Ease of inference. This analysis has not been fully studied yet.

Performance level. No information yet.

Coverage of requirements. This analysis could be interesting in order to identify unexpected behavior, in particular if the estimated execution time of a given method can be unexpectedly high.

Although this analysis is not directly related to security, TL is working on performance benchmarking of Java applications in other projects and the availability of a method to statically determine some bounds on the performance of a program could be interesting.

However, some additional work on this type system is required before to apply it in our framework, which may be difficult in the time frame of the Mobius project. In particular, estimating the time required for API methods is a very hard task.

4.4 Alias control

Alias control type systems introduce a hierarchical structure in the memory heap. The hierarchy is based on the notion of an owner of an object where an object may have exactly one owner. This discipline allows to view the heap as a tree structure the nodes (called contexts) of which are all objects with the same owner (we talk then about the owner of a context) and where if the context C_1 which contains the owner

of another context C_2 then C_1 is a parent of C_2. In order to express and check the ownership relation few annotation constructs are introduced. The ownership discipline allows then then to verify that an object can be accessed in a read write mode only by its owner or by objects from the context to which it also belongs. Different verification problems are addressed by control type systems, e.g. modular verification of object invariants and frame conditions as well as race freedom.

Architectural compatibility. No specific issue.

Coverage of features. Some features are not yet supported, such as static fields.

Modeling features. A significant work is required to model the entire API, but similar work has been performed before on other APIs, and tools exist to help.

Usable on standard MIDP applications. The type system can easily be adapted to the MIDP environment.

Ease of inference. The system normally requires the applications to be annotated specifically, but some partial inference may be possible. By partial we mean that some minimal human effort will be required so as to decide for some part of the heap its ownership relation. This could be problematic as we usually deal with bytecode programs which makes more difficult the annotation task than for source code programs.

Performance level. No information yet.

Coverage of requirements. This analysis could be interesting in order to prove properties about multithreaded application, and to then use these properties to make other analysis more efficient.

In the context of our industrial tool which deals with multithreaded programs, establishing that an object can not be modified in a thread can significantly increase the precision of our dataflow analysis. In this sense, it could be interesting to adapt the alias control type system introduced in Mobius to establish that some part of the heap is not modified by some thread. Such result can turn our analysis more precise.

Chapter 5

Conclusion and future work

We started Task 2.6 with a very limited prototype of type-based static analysis, based on an abstract interpretation framework. This prototype has been enriched to incorporate more precise algorithms and to be more flexible. Today, the precision of its string domain has been improved, and it has been extended in order to support the analysis of several abstract domains simultaneously. The current version has also been enriched with a domain for information flow analysis.

The prototype is the base for an industrial tool that is used by application providers and operators in order to automatically enforce their security policies. The focus of the tool therefore is to verify fixed sets of framework-specific properties on a very diverse set of applications, with very little influence on the developers.

The tool is now able to determine many interesting properties on actual MIDP applications, but some issues remain open, and will require further work. Among these issues are

- Although we have done improvement in the precision of the heap representation, it could be useful to infer more alias information which can then make possible to be get more precise results in the presence of multithreaded programs
- The basic information flow analysis that has been developed yields interesting results, but its precision needs to be increased, in particular through the consideration of implicit information flows.

In other Work Package 2 tasks, academic partners have worked on the design of type systems for various purposes. The main objective of these type systems is to be integrated in the proof-carrying-code environment developed in the context of Work Package 4, which should be able to verify the type certificates corresponding to these type systems. These results have been described in deliverable D2.3 (delivered at M24).

Integrating the type systems in the industrial prototype developed in Task 2.6 only represents an intermediary objective, because of the limits that are inherent to this prototype, and because of the current limitations of the type systems. In particular, the industrial prototype only works as an inference engine, and it only uses abstract interpretation, whereas some of the type systems rely on more sophisticated logic algorithms. In addition, many type systems either don't support essential features of the Java language yet, or they rely on an extension of the language, which is not supported by the kind of Java applications targeted by the prototype.

This partial mismatch between the basic algorithms and the industrial tools has been identified in the initial risk management of the MOBIUS project. The challenges reported in the present document are related to two clearly identified issues:

- *The enabling technologies are insufficiently performing.* This actually happens here, but the context is very specific to Task 2.6, and it is not expected to have any impact on the main prototypes. The enabling technologies are still expected to provide an adequate level of performance in the context of the prototypes to be developed in the context of Work Package 4.

- *Relevant security policies resist formalization.* Some of the issues are related to the fact that some features that are frequently used in actual applications may not be modeled correctly, leading to issues in the enforcement of some security policies. However, the initial review work is based on the first release of the type systems, and additional work is scheduled in the project on this topic; the feedback in the present report can therefore be considered as input by the partners who are developing these type systems.

Despite these issues, and even if the type systems cannot be immediately reused in the industrial prototype, the research work behind them provides interesting ground on which to build more limited algorithms, more suited for the particular use of the type system. Practical applications of static analysis can be highly effective even with algorithms that do not provide a full coverage of the issues, in particular in TL's context of security evaluations. In particular, two type systems show interesting potential:

Information flow. The information flow currently implemented in the prototype is interesting because it already goes beyond the single notion of confidentiality; however, it is far too permissive. On the opposite, the information flow type system is interesting because it covers implicit information flow; however, it is far too restrictive. As mentioned in Deliverable 1.2, the issue is here to find a way to handle sensitive data, by allowing a reasonable use of it, while forbidding abuse. To that end, the work on declassification will be very important.

Alias control. Alias control is indirectly related to the precision of heap analysis. By allowing the classification of objects in sets that exhibit strong properties with respect to concurrency, it becomes possible to make strong assumptions about how objects in the heap are updated, and therefore to make the analysis more precise. This may be longer term research, but its results are very important for the future of the industrial product.

Task 2.6 is an interface task between two different types of innovative work, with different focus, and more importantly, different timeframes. The industrial prototype focuses on results that are fully exploitable today, and which can significantly increase the value of a product in the near future. Such products are required for the early deployment of communicating mobile applications, which will later enable global computing. On the opposite, most of the innovation work in MOBIUS targets more long-term applications, and focuses on emerging technologies, which are not yet mature enough to be exploited in industrial products. A very positive result of Task 2.6 is that it establishes a bridge between these two works, with significant yields on at least two different aspects:

- It confronts the academic partners with constraints from the industrial world, and provides them some input for the future orientation of their work.
- It provides crucial information to the prototype developers, as the input from academic research allows them to consider new directions for the improvement of their tools.

Bibliography

[1] P. Crégut and C. Alvarado. Improving the security of downloadable Java applications with static analysis. In *Bytecode Semantics, Verification, Analysis and Transformation*, volume 141 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2005.

[2] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, December 2003.

[3] MOBIUS Consortium. Deliverable 1.2: Framework-specific and application-specific security requirements, 2006. Available online from `http://mobius.inria.fr`.

[4] MOBIUS Consortium. Deliverable 2.3: Report on type systems, 2007. Available online from `http://mobius.inria.fr`.

[5] Interoperability stepping stones, release 6. Technical report, SIM Alliance, 2006.

[6] E. Vétillard. Proactive security for mobile applications. White Paper PU-2004-RT-621, Trusted Logic, 2004.

[7] E. Vétillard and R. Marlet. Enforcing portability and security policies on Java Card applications. In *e-Smart*, 2003.

[8] E. Vétillard and R. Marlet. Method for determining operational characteristics of a program. Patent EP1700218, Trusted Logic S.A., December 2004.