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 D2.2

Intermediate report on implementation of type systems

Due date of deliverable: 2007-03-01 (T0+18)

Actual submission date: 2007-03-28

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)	

Acknowledgement

Site	Contributed to
TL	All document

Executive Summary:
Intermediate 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`.

This document presents intermediate results that were obtained during the first 18th months of the MOBIUS project in Work Package 2, Task 2.6 (Type System Prototypes).

The main focus of task 2.6 is to confront the most forward-looking academic research work 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 first overlooked.

The present report focuses on the first stage of Task 2.6, which is a mostly preparatory phase, in which some work has been performed on the industrial prototype in order to increase its flexibility and adaptability, and in which the initial research work has been evaluated with industrial criteria.

The generic library developed by TL for the static analysis of Java programs is described in Chapter 2, together with the first architectures derived from it. Chapter 3 details the intermediate results of the implementation, and Chapter 4 discusses about the adaptation of MOBIUS' type systems (described in [5]) and related issues. Finally, Chapter 5 concludes on the advancement of the work package and discusses the future work in Task 2.6, and its interfaces with the future work in other tasks from Work Package 2.

Contents

Chapter 1

Introduction

This document presents intermediate results that were obtained during the first 18 months of the MOBIUS project in Work Package 2, task 2.6 (Type System Prototypes). 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 will be 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), and investigates the practical isssues 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 standalone 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 intermediate 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 [5], focusing on the issues that will need to be addressed for the integration into the industrial tool.

The final chapter describes the future work envisaged on the prototype, as well as the objectives for the demonstration of the implemented prototype, which is scheduled at T0+24.

Chapter 2

Static Analysis at Trusted Logic

The challenge of the 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, then the purpose of this chapter is to present an overview of Trusted Logic's technologies on static analysis.

2.1 Starting point

The starting point for the Java static analysis tools developed in the context of the MOBIUS project is a product line developed by Trusted Logic. Trusted Logic has a long history of building software that reasons about programs. One of its earliest success, back in 2000, was to build a bytecode verifier that was small and efficient enough to run on a smart card [2], which eventually won an IST Prize in 2001. In parallel, Trusted Logic had built several tools, including a tool for automatically generating functional tests from program specifications, which has become the TL-CAT product.

2.1.1 The Generic JVM

In parallel, a 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 [9].

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 [3] for bytecode verification. Then, 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.

Among the items that can be abstracted, we have the following:

- All values, with different characteristics for integral and reference values.
- The Java stack, with 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.

2.1.2 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 [6], 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 [8]. The analysis algorithm therefore is quite precise:

- The implemented architecture implements the *polyvariant* flow analysis algorithm, *i.e.*, 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 quite precise, and it includes a precise handling of loops, which can unravel small loops (others are approximated).
- The heap approximation is also quite 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, Java values are also quite precise. Integral values are represented by integer intervals, and reference values by the list of objects that the value can represent.

As a result, this analysis is very precise, and it yields good results on many 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.

2.1.3 The MIDP analysis

The analysis tool

In 2003, with the introduction of MIDP 2.0, mobile applications started having access to sensitive resources. A need could therefore emerge for the analysis of mobile Java applications [1, 7]. 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:

- 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.

- 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.
- 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 interface that verifies a set of rules, and a web-based interface that 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 operators, in particular as they were required to open ever 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.

The MIDP algorithms

Because of the relative complexity of the CLDC virtual machine, which supports a large subset of Java, with dynamic class initialization, multiple threads, and complex APIs, we have decided to limit the complexity of the analysis. Thus, 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 (PC) of a method, what are the possible values in the top Java frame, including the method's local variables and the Java parameter stack. Thus, the analysis is here *context-insensitive* or *monovariant*.

The level of complexity is therefore very 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.

The abstract execution model is not the only one that has been simplified. The representation of values is also 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 are only kept for reference, since the mandatory off-device preverification removes all uses of subroutines, and therefore of return addresses.

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

- 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.

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).

Modeling Java Strings

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

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

Definition 2.1.1 *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 if 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 off s_1, for any $s_1, s_2 \in \mathit{String}$ and $c \in \{\epsilon, *\}$.

Chapter 3

Initial Work in the MOBIUS context

3.1 Objectives

The technology described in Chapter 2 is powerful enough to verify the simplest security requirements (as defined in [4]), 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 were able to identify a large proportion (between 75% and 90%) of the uses made of the target APIs. However, since an application makes several uses of these APIs, 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 to include results from MOBIUS research 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.
  In the context of Java mobile applications, the importance of determining as much as possible the elements of a connection URL is critical. Section 3.2 of this chapter describes the enhancement of the abstract domain for the strings.

- Multiple domains management.
  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 of the basic abstract value domain.

- Information flow domain.
  This section describes the first experiments about the management of an additional domain for information flow properties.

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

3.2 Enhancing 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. These connections

are made through Java's Generic Connection Framework (GCF), in which 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. This happens for instance on 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. 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. 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 simply is 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 the set on 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.
- $\top_\phi = \langle \emptyset_\phi, \ldots, \emptyset_\phi \rangle$ is 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 the undefined data flow control. This value is used in two cases: if the value is not initialized, and if the value in 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:
- $\forall x = \langle x_1, \ldots, x_n \rangle, y = \langle y_1, \ldots, y_n \rangle \in \Phi$,
  $x \sqcup_\phi y = \langle x_1 \cup y_1, \ldots, x_n \cup y_n >$

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 origin:

- `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.

For each variable is associated one triplet $\langle \texttt{CODE}, \texttt{NET}, \texttt{OTHER} \rangle$ 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);
...
```

Result of the analysis

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

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

- Origin of connection String parameter is CODE (success).
- Origin of connection String parameter is only NET (failed).

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:

- `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.

Java code

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

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

- If `prop` is `null`:
  - We defined:
    $\alpha(\texttt{"sms://+358401234567:6578"}) = \langle \texttt{CODE}, \neg\texttt{NET}, \neg\texttt{OTHER} \rangle$
  - Thus $\alpha(\texttt{address}) = \langle \texttt{CODE}, \neg\texttt{NET}, \neg\texttt{OTHER} \rangle$
- If `prop` is not `null`:
  - We defined:
    $\alpha(\texttt{"lieu"}) = \langle \neg\texttt{CODE}, \neg\texttt{NET}, \texttt{OTHER} \rangle$
  - So $\overbrace{\texttt{getAppProperty}}(\texttt{"lieu"}) = \langle \neg\texttt{CODE}, \neg\texttt{NET}, \texttt{OTHER} \rangle$
  - Thus $\alpha(\texttt{address}) = \langle \neg\texttt{CODE}, \neg\texttt{NET}, \texttt{OTHER} \rangle$

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 resolve

This information flow domain is very useful for the rules that apply on the origin of data. A set of examples of these types of rules (in [4]) 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 type of rules is very important for security of MIDP applications. 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

This first attempt to define an information flow domain is limited, and it has mostly been motivated as a way to put in practice the multi-domain framework without bringing too much complication. Nevertheless, this domain has proved itself useful, and could be the base for further improvements. In particular, we have noticed the following issues:

Lack of precision. This lack of precision comes from many reasons, and in particular from the fact that the analysis is context-insensitive, and from the fact that the heap representation is extremely trivial, leading to many unknown values in all domains, including the information flow domain.

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 Capitalization 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 current heap abstraction is extremely simple, and virtually all values stored in object fields come out as a $\top$ undetermined value. In particular, any value stored in a container is lost. We are currently investigating a way to make the heap abstraction more precise, and in particular to keep some information about the data stored in containers.

Enhancing the heap abstraction is feasible, but it requires significant changes in the analysis mechanism. In particular, it requires the introduction of some kind of a fixpoint algorithm in the analysis in order to ensure that the heap abstraction reflects all possible solutions. Because of this fixpoint, the depth of the abstract domains must remain quite limited, in order to ensure that the analysis converges fast enough.

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.

In addition, the heap abstraction may also be extended in order 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 are 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

As scheduled, most of the work performed so far on the static analysis tool has been preparatory work, with the objective of enhancing the existing tool, and of preparing it for the integration of new type systems. However, since the final objective of Task 2.6 is to integrate some of the type systems developed in the context of MOBIUS in an industrial type-based static analysis tool, some preparatory work has been performed on the various type systems defined in [5].

We did not expect the type systems defined in the context of this first deliverable to be directly applicable to an industrial product. The main reasons are as follows:

- 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 current structure of the tool limits the kind of analysis that it may support.

Nevertheless, it is important to perform this initial review of the theoretical work, since it will allow us to build more practical algorithms in the remainder of the work package.

In the present section, we will review the various type systems proposed in [5], and assess how these type systems could be used in the context of TL's type-based static analysis system. In addition, since the objective of Task 2.6 is to work on a prototype of an industrial product, some constraints are related to the fact that the proposed type systems must provide sufficient guarantees that they will actually be efficient on these products. The basic criteria that we have used are as follows:

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, 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 analysis, 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

A type system has been defined for Information Flow, whose purpose is to guarantee the confidentiality of all objects handled, by focusing on who accesses sensitive objects. The type system is interesting in the sense that it also tracks implicit information flows. The type system requires extra security annotations, but it is possible to generate these annotations through an appropriate static analysis.

About the typing judgment, the stack-based virtual machine should be extended to manage a security environment for each program point: this could be represented by a security-level abstract interpretation.

4.2.2 Suitability

We here review the type system 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 may require a difficult adaptation of our existing control-flow analysis.

Coverage of features. The current version of the type system does not cover arrays, and it does not support multi-threading.

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 functions.

Eaase of inference. No specific issue.

Performance level. Initial results tend to show 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

In the future work items for the type system, there are some interesting items, and in particular the following ones:

- Multi-threading is to be considered in the type system.
- Declassification also is to be studied in a general setting, which could bring some very 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. Allocation is not possible in recursive code.

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; in addition, the model is rather based on a used declaration.

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 organizes the heap in a hierarchical structure of nested non-overlapping contexts. Each context is characterized by an object. This is the definition of Universe Type System. Types are extended with Universe annotations, namely `rep`, `peer` and `any`. This hierarchical structure simplifies reasoning on the programs.

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 inference may be possible in the context of multithreading. However, the need for runtime checks makes the system difficult to use in practice.

Performance level. No information yet.

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

Alias control is a promising technique, especially in multithreaded environments. As most MIDP programs have a very restricted concurrent behavior, it could be interesting to infer alias information about the program and to then use this information in order to make the 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, a few have been outlined specifically:

- More work is required on the basic analysis, which requires in particular a more precise representation of the heap, and a more precise abstraction of heap-related operations.

- 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 it to be later integrated in the proof-carrying-code environment that is being developed in the context of Work Package 4, which should be able to verify the type certificates corresponding to these type systems. The first results on type systems have been described in deliverable D2.1 (delivered at M12).

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 issues report 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 conisdered 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.

For the remainder of Task 2.6, we will therefore need to focus on a few topics, and to integrate some of the most advanced research work in our industrial prototype. This selection is made difficult by the restrictions of the prototype, and by its strong industrial constraints. Nevertheless, 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.

In the context of Task 2.6, the main immediate work item will be on the representation of the heap in the analysis, because this is one of the major cause of imprecision of the prototype. In parallel, TL will collaborate with other WP2 partners in order to make the type systems developed in Tasks 2.2 (Mechanisms for safe information release) and 2.4 (Advanced resource policies and analysis) more suitable for use in the industrial prototype, while providing some input about the actual requirements on these two topics.

This collaboration is the heart of this task, even if it cannot be applied to all MOBIUS type systems. 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. On-card bytecode verification for Java card. In I. Attali and T. Jensen, editors, *e-Smart*, volume 2140 of *Lecture Notes in Computer Science*, pages 150–164. Springer-Verlag, 2001.

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

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

[5] MOBIUS Consortium. Deliverable 2.1: Intermediate report on type systems, 2006. Available online from `http://mobius.inria.fr`.

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

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

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

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