Mobius

Information Society Technologies

Project N^o: **FP6-015905**

Project Acronym: **MOBIUS**

Project Title: **Mobility, Ubiquity and Security**

Instrument: **Integrated Project**

Priority 2: **Information Society Technologies**

Future and Emerging Technologies

Deliverable D1.1

Resource and Information Flow Security Requirements

Due date of deliverable: 2006-03-01 (T0+6)

Actual submission date: 2006-03-27

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

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

Revision 285 — Final

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)	

Executive Summary:
Resource and Information Flow Security Requirements for MOBIUS

This document summarises deliverable D1.1 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 objective of this document is to concretely specify the resource and information flow security requirements relevant to global computing that will be studied and addressed throughout the project.

A project kick-off meeting in October 2005 included sessions on information flow and resource security, with presentations by the academic (CTH, INRIA, RWTH, UPM) and industrial partners (FT, TL). A significant outcome of the kick-off meeting was the decision to concentrate on a specific platform, the Mobile Information Device Profile (MIDP, version 2)[1] of the Connected Limited Device Configuration (CLDC)[2] of the Java 2 Micro Edition (J2ME). This specificity allows the requirements in this document to be concrete, and supports concrete case studies in later work packages. At least one third of the mobile phones in the world support MIDP. The MIDP security model, with off-platform bytecode pre-verification and digital signing, provides a convenient hook for use of Proof Carrying Code techniques. Applicability of the research to be carried out in this project is not restricted to MIDP.

Information flow security We identify sensitive information, its sources, and hidden channels that may transfer it in MIDP applications. Sensitive information includes personal data, geo-location, and password data. Potentially dangerous consumers of this information include the persistent store and the network. Information may be leaked by assorted channels, including those created by MIDP's multithreading.

We consider high-level information policies for multi-threaded languages, treating information leaks via scheduler interleaving decisions. The MIDP information-flow case strongly suggests the need for declassification policies.

We also treat bytecode-level information flow, considering the bytecode-level attacker view, the part of the security model that defines what bytecode-level configurations the attacker may (not) distinguish. Addressing information-flow policies in bytecode-level multithreaded languages with declassification is a long-term goal in the MOBIUS project, subject to further security requirement gathering within Work Package 1.

Resource security Central issues on resource security policies are: what resources they should describe; how resource policies can contribute to security; and what kinds of formalism are appropriate.

Notions of "resource" that are suitable for a formal treatment and also meaningful in real-world applications on MIDP include classical resources (execution time, memory space) as well as platform-specific resources such as text messages (SMS), persistent database records and 'billable' transactions. Many of the latter can be unified under an approach that regards particular system calls as the resource to be managed:

[1]A *profile* is a set of higher-level APIs that further define the application life-cycle model, the user interface, and access to device-specific properties

[2]A *configuration* is a virtual machine and a minimal set of class libraries providing the base functionality for a particular range of devices that share similar characteristics, such as network connectivity and memory footprint.

how many times they are invoked, and with what arguments. This fits with the existing MIDP validation model, where certain APIs are only available to trusted applications.

We develop scenarios for MIDP resource management, based on "block booking" of text messages, and investigate how safe use of such a facility this can be supported by static analysis of program code.

These analyses reveal a range of resource security issues where formal guarantees would enhance the trustworthiness of mobile code, but where they are not currently available. The MOBIUS project proposes to enable this with proof-carrying code, and we explore the technologies available to support generation and transmission of resource proofs. Building on previous research by consortium partners, we identify specific requirements for a MOBIUS resource security platform.

Contents

1 Introduction **7**
1.1 Task descriptions . . . 7
1.2 Kick-off Meeting . . . 8
1.3 Summary of Chapter 4 . . . 8
1.4 Summary of Chapter 5 . . . 9

2 Technological Background **10**
2.1 Java for embedded telecommunication applications . . . 10
2.1.1 Embedded profiles . . . 10
2.1.2 Configuration and profiles . . . 10
2.1.3 Profiles for embedded systems . . . 11
2.1.4 Mobile Information Device Profile (MIDP) . . . 11
2.2 MIDP characteristics . . . 12
2.3 MIDP security policy . . . 12
2.3.1 Use of digital signature to reduce the number of authorisation requests . . . 13
2.3.2 Limits of MIDP built-in security policy . . . 13
2.4 Validation of midlets by operators . . . 14
2.4.1 Basic security needs . . . 15
2.4.2 Validation in current practice . . . 15
2.4.3 Bytecode analysis and Proof Carrying Code . . . 15
2.5 MIDP Programming model . . . 16
2.5.1 Event-driven programming . . . 16
2.5.2 Concurrency between different kinds of events . . . 17
2.5.3 Concurrency in a midlet . . . 17
2.5.4 Concurrency patterns in MIDlet programming . . . 18

3 Threat Model **20**
3.1 MIDP threat model . . . 20
3.1.1 Assets to be protected . . . 20
3.1.2 Possible goals for attackers . . . 21
3.1.3 Possible countermeasures . . . 22
3.2 Common attacks on information flow security . . . 23
3.2.1 Disclosure of confidential data . . . 23
3.2.2 Disclosure of sensitive data . . . 25
3.2.3 Modification of sensitive data . . . 26
3.3 Common attacks on resource control . . . 26
3.3.1 Abuse of billable events . . . 27
3.3.2 Memory usage . . . 28
3.3.3 Control flow misuse . . . 29

4 Information Flow **30**
4.1 What are we trying to protect? . . . 30
4.1.1 Who are the malicious actors? . . . 30
4.1.2 What kind of information flow policy? . . . 30
4.2 Sensitive information sources . . . 31
4.2.1 Sensitive APIs . . . 31
4.2.2 Application specific data . . . 33
4.3 Dangerous information consumer . . . 33
4.3.1 Persistent store . . . 33
4.3.2 Network access . . . 34
4.4 Potential hidden channels . . . 35
4.4.1 Covert channels . . . 35
4.4.2 Taking native store area into account . . . 35
4.4.3 State listeners . . . 36
4.5 A scenario for information flow analysis: an itinerary planner . . . 36
4.5.1 Step 1: Identify your current position . . . 36
4.5.2 Step 2: Plan your route . . . 37
4.5.3 Step 3: Select an hotel . . . 37
4.5.4 Step 4: Booking a room . . . 38
4.5.5 Information flow constraints . . . 38
4.6 Detailed scenario for information flow . . . 38
4.6.1 Desired security properties . . . 38
4.6.2 Requirements . . . 40
4.7 A scenario for information flow analysis: a friend finder . . . 40
4.7.1 Description . . . 40
4.7.2 Where the information is stored . . . 40
4.7.3 The flow . . . 42
4.7.4 Information flow constraints . . . 44
4.8 Approaches to formalization . . . 45
4.8.1 Defining secure information flow . . . 45
4.8.2 Requirements on enforcing non-interference . . . 47
4.8.3 The need for more permissive information flow policies . . . 49

5 Resources **51**
5.1 Candidate Resources . . . 51
5.1.1 Java and resources . . . 51
5.1.2 Specific resources . . . 52
5.1.3 Some other issues . . . 56
5.2 Resource scenarios: Block booking text messages . . . 57
5.2.1 Application scenario I: Hotel reservation . . . 57
5.2.2 Application scenario II: Group messaging . . . 58
5.2.3 Desired resource properties . . . 59
5.2.4 Implementing block booking in MIDP . . . 60
5.2.5 Enforcing resource policies . . . 61
5.3 Approaches to formalization . . . 62
5.3.1 Resource-aware semantics . . . 62
5.3.2 Expressing resource properties of bytecode . . . 63
5.3.3 Generating resource proofs . . . 65

6 Conclusions **67**

A Kick-Off Meeting **72**

Chapter 1

Introduction

1.1 Task descriptions

The objective of this document is to concretely specify the resource and information flow security requirements relevant to global computing that will be studied and addressed throughout the project. Here is how it is described in Annex I:

Task 1.1 Information Flow Security Policies
Information flow controls are an attractive way of achieving end-to-end security properties such as confidentiality and integrity. For example, the lack of information flow from secret to public data implies confidentiality; the lack of flow from tainted to untainted data implies integrity. We will build an attacker model leading to high-precision security definitions in the context of concurrent and low-level languages (such as byte code).

Task 1.2 Resource Security Policies
This task addresses the question of ensuring that downloaded code can run securely within the resources on offer at a given terminal. For example: will an application operate within the memory and time a terminal can offer? Do the services of the terminal operating system and libraries provide sufficient functionality? Will it interact safely with other applications on the terminal? This task will focus on enumerating relevant resources, beyond memory space and execution time, on identifying scenarios in which resource control influences the security of mobile applications, and on establishing criteria for resource policies.

In order to make these requirements concrete we select a particular implementation platform, the Mobile Information Device Profile (MIDP, version 2)[1] of the Connected Limited Device Configuration (CLDC)[2] of the Java 2 Micro Edition (J2ME). Applicability of the research to be carried out in this project is not restricted to MIDP. In Chapter 2 we explain this choice and give some details of MIDP. Chapter 3 presents the threat model of MIDP.

Chapter 4 considers the requirements for information flow security in MIDP, and outlines known technical approaches. Similarly Chapter 5 presents the situation of controlling resource usage for security in MIDP. These are the main chapters that will impact work to be done in later Work Packages. There are summaries of these chapters in sections 1.3 and 1.4.

Chapter 6 concludes by relating this document to the overall MOBIUS project.

[1] A *profile* is a set of higher-level APIs that further define the application life-cycle model, the user interface, and access to device-specific properties

[2] A *configuration* is a virtual machine and a minimal set of class libraries providing the base functionality for a particular range of devices that share similar characteristics, such as network connectivity and memory footprint.

1.2 Kick-off Meeting

Tasks 1.1 and 1.2 both ran for six months from the start of the MOBIUS project, with activity throughout. The project kick-off meeting in October 2005 was a significant opportunity for partner interaction across MOBIUS, with two half-days devoted to Work Package 1. Preliminary discussions on the project's collaborative website led to discussion documents from UEDIN, FT and TL, circulated before the meeting. These contributed background information and proposed discussion points, drawing on the academic research perspective as well as the requirements of current industrial applications.

At the kick-off itself, there were separate sessions on information flow and resource security, with presentations by the academic (CTH, INRIA, RWTH, UPM) and industrial partners (FT, TL). The project website carries abstracts and notes from all these. The agenda is presented here in Appendix A.

As well these formal presentations, discussion sessions addressed a number of relevant issues identified by the partners, including:

- How to relate attacker models to semantic requirements for security policies;
- Connecting high-level policies with details of low-level Java bytecode;
- The significance of concurrency and threading for security of MIDP;
- Integration with the existing Java security policy framework;
- Supporting the wide range of variation seen between Java-enabled mobile devices.

A significant outcome of the kick-off meeting discussions was the decision by project partners to concentrate on MIDP as a specific platform for the major part of MOBIUS research.

1.3 Summary of Chapter 4

Chapter 4 summarises our work on Information-Flow Policies for MOBIUS, Task 1.1. This work targets the goal set in the description of work for Task 1.1, and elaborated at the kick-off meeting. Thanks to the industrial partners, we have gathered information-flow security requirements for the running example of the MIDP profile for mobile computing, including detailed scenarios. Thanks to the academic partners, we have collected and analysed technical approaches to information-flow policies.

Sections 4.1–4.4 identify sensitive information, its sources, its potential consumers, and hidden channels that may transfer it on the running examples of the MIDP profile. Sensitive information includes personal data, geo-location, audio capture data, and password data. Potentially dangerous consumers of this information include the persistent store and the network. Information may be leaked by assorted covert channels, including those that are only possible in the presence of multithreading.

Sections 4.5–4.7 describe scenarios for information-flow. The scenarios include an itinerary planner and a friend finder. With a non-trivial mixture of sensitive and public information, the scenarios indicate that tracking information flow is an important yet unresolved problem. The example profile, MIDP, offers no mechanism for enforcing information-flow control policies.

The goal of section 4.8 is to outline possible definitions of secure information flow in the context of global computing, and to indicate means of enforcing these statements. We identify several attacker models, that shall lead to different non-interference policies, some of which allow intentional information release, and see how both type systems (and related techniques such as static analyses and abstract interpretations) and program logics can be used to guarantee that programs abide to these policies. Section 4.8.1 makes the definition of *non-interference* precise by discussing the observational capabilities of an attacker. Section 4.8.2 discusses requirements on the enforcement methods to be developed, so that they integrate smoothly in the security architecture that shall be proposed within the project. Non-interference is seen to be too strict a policy, so section 4.8.3 discusses intentional information release, or *declassification*.

1.4 Summary of Chapter 5

Chapter 5 summarises our work on Resource Security Policies for MOBIUS, Task 1.2. This is based on presentations and discussions at the kick-off meeting, and draws on existing and current work by the project partners. The object of the task is to identify and characterize what we require of resource policies developed under MOBIUS. Central issues are: what resources they should describe; how resource policies can contribute to security; and what kinds of formal analysis can support this.

Section 5.1 analyses different notions of "resource": the challenge here is to identify resources that are suitable for a formal treatment, and also meaningful in real-world applications. The choice of MIDP as a concrete platform gives focus to this task, and the industrial partners have provided precise evidence about the importance of specific resources. Classic resources like execution time and memory space remain relevant, particularly on these constrained devices . Moreover, a rich platform like MIDP adds a range of platform-specific resources — text messages (SMS), persistent database records, 'billable' transactions — which are important for user security and suitable for formal analysis. Many of these can be unified under an approach that regards particular system calls as the resource to be managed: how many times they are invoked, and with what arguments. This also fits with the existing MIDP validation model, where certain APIs are only available to trusted applications.

Section 5.2 captures this in two scenarios for MIDP resource management based on "block booking" text messages. Current platform implementations enforce resource security by requiring the user to authorise individually every text message to be sent by a MIDP application. This makes standard remote transaction protocols impractical, and also opens up social engineering attacks based on user distraction. A more powerful approach would allow applications to book a fixed number of text messages at the beginning of a transaction; but we then need evidence that these will be sufficient to carry out the transaction, and that block booking is used correctly. The resource here is (pre-booked) text messages, and the scenario is to use static analysis of code to provide a proof, in the MOBIUS logic, that all messages are booked in advance. For MIDP validation the operator could independently check this proof to confirm that a third party's use of block bookings was secure, before signing it as a trusted application.

Section 5.3 surveys formal approaches to resource verification, looking at their potential application to the JVM and proof-carrying code. We focus in particular on existing knowledge brought in by partners, and recent research activity within the project:

- The work of UEDIN and LMU on *mobile resource guarantees*, with type inference for resources, the *Grail* bytecode logic, amortized complexity analysis of heap use by object-oriented code, and a general cost model based on *resource algebras*.
- Research on the use of JML, the *Java Modeling Language*, as an assertion logic. The INRIA Everest group have a translation from JML into a bytecode analogue BCSL; while UEDIN have implemented support for verifying JML specifications of heap space usage in ESC/Java2.
- A framework for resource policies that distinguish between what a code consumer requires and what a code producer is able to offer, in such a way that it is easy to check whether one satisfies the other. UEDIN have implemented this kind of flexible resource policy within the existing Java security model.
- Applications of abstract interpretation to code certification, where the program abstraction itself serves as a certificate. UPM have benchmarked this *abstraction-carrying code* in their *Caio* preprocessor. The INRIA Lande team have used abstract interpretation to validate resource usage in a large body of MIDP code, integrating this within PCC through a formally verified certificate checker.

Drawing these together, we identify a series of requirements for a formally-verified MOBIUS resource security framework.

Chapter 2

Technological Background

In the present chapter, we focus on Java embedded applications: applications present on the customer terminal. If we exclude SIM smartcards, Java is mainly available in its J2ME/CLDC [35, 31] variant. MIDP is a further specialisation of CLDC for network-connectible, battery-operated mobile handheld devices such as cell phones, two-way pagers, and mainstream PDAs. MIDP is the most precisely specified, and most widely deployed Java-based platform for mobile devices; MIDP is deployed on a third of mobile phones worldwide (source Nokia). For these reasons, we select MIDP as a specific platform for which to specify security requirements.

Applicability of the research to be carried out in this project is not restricted to MIDP, but we consider it to be a good model for the challenges that arise in security for mobile applications. Further, its precise specification and wide deployment allows specific scenarios to be considered in this workpackage, and possible case studies in other workpackages.

Specific devices may use optional packages[1] on top of MIDP, e.g. tailored for specific display hardware. Such packages are part of a device's firmware, so can be considered as part of the platform for purposes of security requirements.

2.1 Java for embedded telecommunication applications

2.1.1 Embedded profiles

The Java programming language was originally designed for embedded devices. Nevertheless, it took several years to standardize versions of Java for such devices. Surprisingly, the first such specification was the Java Card specification, whose first industrial version [57] was released in 1997. There are now many Java-based platforms for embedded systems.

2.1.2 Configuration and profiles

Java is not a single language but a family of closely related dialects sharing the same spirit but tailored for the computing resources available on a category of devices. These variants are called configurations and a given abstract machine (runtime environment) usually targets a single configuration:

- the Java Card configuration for smart cards,
- the J2ME/CLDC (Connected Limited Device Configuration) for low/medium end networked devices
- the J2ME/CDC (Connected Device Configuration) for high-end networked devices
- the J2SE or J2EE platform for regular desktop computers and servers.

[1] An *optional package* is a set of APIs extending the runtime environment to support device capabilities that are not universal enough to be defined as part of a profile or that need to be shared by different profiles.

Profiles are sets of libraries delivered with the configuration to specialize an execution platform for a given market segment. Some of those libraries contain hooks to the native functionalities of the platform and provide the link between Java and the outside world.

2.1.3 Profiles for embedded systems

Several profiles exist on top of the CLDC configuration:

- Java is deployed on many mobile phones with the MIDP profile (JSR 37 [36] - 118 [23] - 271)
- NTT DoCoMo compliant phones implement the i-mode DoJa [19] proprietary profile,
- Trusted applications like banking may use the mobile STIP profile [16] (strictly speaking the configuration used is not CLDC but can be considered as a subset of it),
- Environments for set-top-boxes include the current MHP profile [53] and the "On Ramp to OCAP" profile [33] (JSR 242) (OCAP being the American version of MHP),
- Java is still expected on triple play home gateways with a machine to machine profile like IMP (JSR 195 - 228 [29, 32]).

More powerful platforms, such as J2ME/CDC (Java 2 Micro Edition with the Connected Device Configuration) and J2SE (Java 2 Standard Edition) are not viable solutions for today's mobile devices, and will be out of the scope in the near/medium future for at least two reasons:

- Hand-held devices do not meet the performance requirements for J2ME/CDC, let alone J2SE.
- The security model of CDC is more complex but not adequate for the needs of embedded applications using critical resources.

We will concentrate on the MIDP profile.

2.1.4 Mobile Information Device Profile (MIDP)

After the specification of Java Card, a slightly larger virtual machine, the KVM, was developed. This R&D work became public as the CLDC (Connected Limited Device Configuration), whose first specification [35] was issued in 2001. This initial specification was soon extended with a specialization called MIDP (Mobile Information Device Profile), whose first specification [36] was issued at approximately the same time. The target for these specifications is an average mobile phone, up to low-end smart phones.

These specifications encountered great success; many small applications were developed. Many phones supported the technology by including a Java runtime environment. The technology presented the major advantage of allowing games and other interactive content to be downloaded to a mobile phone without having to include an open operating system which requires a MMU (and so more expensive hardware) and does not achieve the level of security (confinement of downloaded applications) offered by the sandbox model. Despite the initial success, this first round specification soon reached its limit: connectivity was restricted and few phone features were accessible, so no really interesting applications were developed.

By that time the development of Java technology, in particular in the embedded world, had been transferred by Sun to the Java Community Process, an industry-wide process that aims at defining Java-related standards. The next releases of the standards CLDC 1.1 and MIDP 2.0 [31, 23] occurred around the beginning of 2003. These releases remain the basis for all implementations of MIDP. The changes mostly consist of increased openness, together with enhancement of the security model.

2.2 MIDP characteristics

An MIDP application is called a *midlet*. The main characteristics of MIDP are:

- Like an applet, a MIDlet is a managed application. Instead of being managed by a web browser, however, it is managed by special-purpose application-management software (AMS) built into the device.
- A midlet is implemented as a class providing callbacks to handle the main events of its life cycle. Only one midlet is active at a time.
- The behaviour of the midlet is mostly built around its user interface implemented as a set of screens that contain commands. Activating a command will trigger a registered callback.
- Midlets are deployed and updated dynamically over-the-air (OTA). This is a hook on which PCC checking could be implemented. MIDP also enables a service provider to identify which midlet suites will work on a given device, and obtain status reports from the device following installation, updates or removal.
- There is a single unified communication framework. A communication channel is created by calling `Connector.openConnection` with a *string* containing a URL, i.e. the schema (protocol), the address of the destination and eventual arguments. The result is an object of a subclass of `Connection`. This subclass will contain the actual methods for sending data.
- Persistent data can be stored in a record store, a small database usually written in the flash memory of the phone. The default is that each database belongs to a given application, but they can share it with the others.
- Access to a critical method is protected by a warning screen requesting authorization from the user. The number of screens depends on the security level of the midlet and of the API. It can appear only once, each time a new session is started, for each call, or never if the security control is disabled (see section 2.3).
- Companion specifications define additional APIs for extensions of MIDP: wireless messaging [24, 30], multimedia players [25], bluetooth [38], PIM [37] (contact list, tasks), geo-localisation [27]. A good survey of all these specifications is the early draft of forthcoming JSR 248 [34] (Mobile Service Architecture for CLDC).

J2ME/MIDP is often presented as a fragmented environment; the scope of the MIDP standard (not limited to mobile phones), the number of companion JSRs, the way the standardization is made (JSR can only specify properties directly testable in Java) are the main reason for this. There are ongoing efforts to clarify what a MIDP phone should look like:

- JTWI [28] is the first attempt to clarify the relation between the different specifications.
- MSA [34] will be a platform specification. It presents what should be available on a mobile phone environment.

2.3 MIDP security policy

Java provides a way to install services developed by third parties, while providing a level of security to the customer. The KVM supports a sandbox security model similar to the JVM, although some aspects of bytecode checking are pre-computed at compile time to lighten the memory and compute load at bytecode checking time. This is supposed to ensure that no midlet can crash the platform it is running on (phone, PDA, . . .). Nonetheless, the use of some MIDP methods by a midlet may create risks for the user

- cost: SMS, geo-localisation
- access to private information: address-book, camera, audio recorder
- loss of data from on-device storage
- denial-of-service attack

Those method calls are grouped in function groups. To each function group is associated a permission level that states if and how access to the API is granted.

Allowed The API can be freely used

Denied The API cannot be used (not formally present but enforced for some very critical APIs when the application is not signed)

User permissions The user is prompted with an “intuitive, user-friendly” message asking whether to proceed or not:

blanket user authorization is given once and for all

session user authorization is valid until the midlet suite terminates

one-shot user is prompted each time

2.3.1 Use of digital signature to reduce the number of authorisation requests

In a midlet using sensitive APIs, the number of authorisations required to access these can be overwhelming. Moreover, there are APIs that are too dangerous or too complex to let the user decide whether or not to authorise the method call; e.g. access to the SIM card.

This is why MIDP introduces the possibility to digitally sign applications. Digital signatures use conventional public key infrastructures and a MIDP phone should contain root certificates for the operator and the manufacturer private keys. It may contain other certificates counter-signed by the root certificate or a chain. A signed midlet is checked with the public keys contained in the certificates and will belong to a *domain* corresponding to the level of trust associated with the signature: operator domain, manufacturer domain or third party domain. The level of confirmation required for each function group is defined for each domain. See section 2.4

The implementation details and the default settings are defined in the “Recommended Security Practice for GSM/UMTS Compliant devices”, an annex to the MIDP specification. It is not part of the core specification because it does not follow the rules of the JCP that implies that specification conformance can be completely tested from the Java environment (a Test Compatibility Kit must be supplied with each JSR) but it is endorsed by most carriers.

2.3.2 Limits of MIDP built-in security policy

The KVM bytecode checking gives a formal assurance that a midlet behaves well in some specified sense, at least assuming that the bytecode checking has been correctly specified and implemented. However, digital signing of midlets gives no formal assurance of any behavioural properties. Digital signing can increase confidence that a midlet was examined and/or tested by a third party code producer, the operator, or the manufacturer. But many tested programs have bugs. Proof carrying bytecode greatly extends the correctness policies that can be enforced, and the confidence that they are actually enforced.

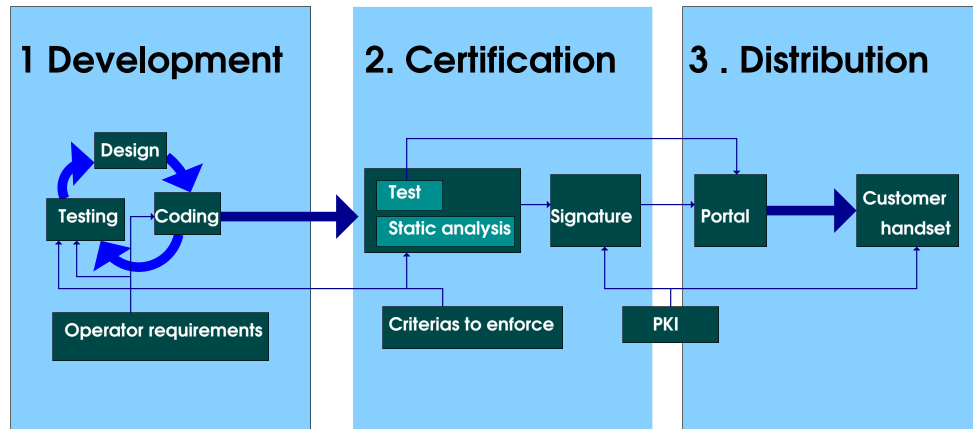

Figure 2.1: Midlet validation process

Lack of information-flow security policy

Information flow properties are not addressed by the MIDP *implicit*[2] security policy because there is no mechanism for enforcement. Information flow properties depend on all possible execution paths, so dynamic checks cannot prove anything about such properties. The only property specified in "Recommended Security Practice for GSM/UMTS Compliant devices" is the following:

> Blanket permission given for some combinations of Function groups can lead to higher risks for the user. For MIDlet suites in the Third Party domain the user MUST be notified of the higher risk involved and also acknowledge that this risk is accepted to allow such combinations to be set. The combination of Blanket permissions in function groups where this applies is: Any of Net Access, Messaging or Local Connectivity set to Blanket in combination with any of Multimedia recording or Read User Data Access set to Blanket.

This only informs the user that some possibly private information may go out of the phone, without giving any basis to decide whether to authorise it or not.

Lack of resource security policy

As mentioned above, MIDP gives control over access to some dangerous method calls, which can be considered resources. Other than this there is no control over space, time, ... There are some unenforced recommendations to prevent midlets from locking up the device display during time consuming operations such as making a network connection.

Lack of operator security

There is no provision for network security in the Java framework. Some MIDP APIs give access to identifiers that should not be used by end-users (IMSI, SIM contents, etc.) The main risk is denial-of-service attacks against the network. This is mainly a resource security issue.

2.4 Validation of midlets by operators

As mentioned in section 2.3 the security and correctness guarantees of MIDP bytecode checking cannot express or enforce all the desired properties of a midlet. Every midlet will need to be validated to meet some requirements more stringent than the MIDP bytecode checks, by some trusted entity: the software developer

[2] "Implicit" because it is described by the enforcement mechanism, not by its objectives.

(possibly a third party), the device manufacturer or the telecommunication operator. MIDP supports digital signing by a trusted entity as some evidence that this very midlet has been validated by that entity. See figure 2.1.

2.4.1 Basic security needs

When an operator signs a midlet, he will change the default behaviour associated to function groups, for example raising a one shot permission to a blanket permission. The meaning for the end-user is that some warning screens he would have seen when using the unsigned midlet will disappear on the signed version.

Signing means knowing and accepting the screens that would have been displayed. In practice it means:

knowing the calls to sensitive methods: because Java is an object oriented language with dynamic method lookup, this goes beyond class identification.

knowing their potential arguments: the basic example is the opening of a connection. There is a single method `Connector.open`; the description of the connection is in the string given as parameter. It contains the kind of connection to be opened (IRDA, SMS, etc.), the address (the cost may depend on the address, e.g. premium SMS) and other parameters for the connection.

knowing how many times they will be called: a user may accept to send an SMS to get a result but not hundreds of them. The number of calls to a billable method has a direct impact on the end-user bill.

knowing when calls occur: a midlet may be used forever and make an infinite number of billable method calls. What is important is that the number of billable events is limited for a given transaction.

2.4.2 Validation in current practice

The common practice is to test the application. The Unified Testing Initiative, founded by SUN and some manufacturers, and now involving some operators, defines test plans known as the Unified Testing Criteria [42] widely accepted in the industry.

Operators usually complete this evaluation with their own tests. In France, Orange test teams also use some static analysis tool developed by France Telecom R&D division, mainly to check the possible values of the parameters of methods considered as dangerous.

Finally download portals usually have a class database and a limited tool to analyse the contents of JAR files. Classes can be marked as dangerous or only available on a given handset. These rudimentary tools may help the classification of midlets on the download portal.

Unless the midlet is really critical (which should be the case only for operator midlets installed on the phone before sale), code reviews are usually considered too expensive. Static analysis is one possibility for evaluating the security of an applet; it can check a property on all execution paths.

2.4.3 Bytecode analysis and Proof Carrying Code

As explained in section 2.3.1, the signer of an application is either a trusted third party, the operator or the manufacturer. Usually none of these entities are the actual developers of the midlet; the application may have been sold to them by another entity who just acts as a content provider/aggregator. The signers may not even have access to the source code of the application. On the other hand, they are the ones that take the responsibility of signing a midlet. This is why they need tools to check that the midlet is not malicious by direct inspection on the bytecode. Those tools should at least give clues to answer the questions raised in the previous section.

From figure 2.1 we see that testing and/or static analysis of a midlet's bytecode, followed by digital signing, need not be carried out on a limited platform. This "static analysis" could well involve checking a correctness proof bundled in the .jar file, and this proof checking can take place on a powerful computer

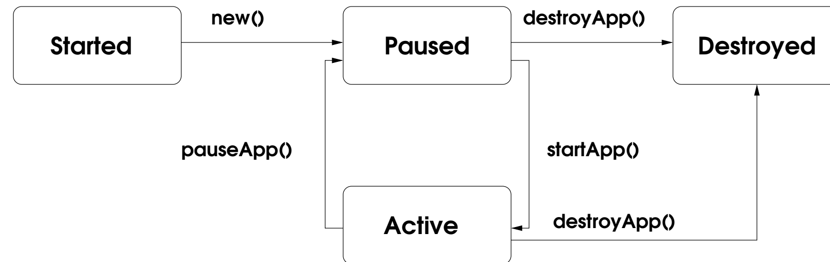

Figure 2.2: MIDlet life-cycle

using a powerful proof checking program. This correctness proof could have been developed (automatically or interactively) with reference to the source code, then transformed by a "proof transforming compiler" to an annotated .jar file. In this way, a third party software supplier can be required to provide some proof as evidence the bytecode behaves well, and the midlet signer can check the bytecode correctness proof on a powerful computing platform. Finally the signed bytecode can be trusted by an end user who trusts the signer, as any changes to the bytecode after signing will be detected as a signature failure.

2.5 MIDP Programming model

A midlet suite (the basic unit provisioned on the mobile phone) is bundled as a Java archive (jar file) and an optional descriptor file (whose contents is usually replicated in the manifest file of the archive). Each midlet of the midlet suite is a separate application implemented as an object of class MIDlet (package `javax.microedition.midlet`). It is referenced in the manifest file or the descriptor as a potential entry point with a name to be displayed by the application management system on the screen and an optional icon.

2.5.1 Event-driven programming

MIDP is based on an event driven model. The midlet object, handled by the application management system (AMS), will follow a life-cycle that can be described by a state automaton (figure 2.2). Transitions between states are triggered by the AMS when an event is available and the previous one has been treated. The AMS will call some methods, specified in abstract classes or interfaces defined in the MIDP specification. The programmer of a midlet creates actual code implementing these interfaces.

Midlet life-cycle

The main methods for callback belonging to the MIDlet class are:

- the constructor itself,
- `startApp()`,
- `pauseApp()`,
- `destroyApp()`.

`pauseApp()` is called when the Java virtual machine is interrupted by a task with a higher priority (answering a phone call, notifying the reception of a SMS); the application can take actions to pause, such as stop animations and release resources that are not needed while the application is paused. This behaviour avoids resource conflicts with the application running in the foreground, and unnecessary battery consumption. The exact behaviour (specifying on which events and how the events interleaves with events notifying the availability of the screen) is not fully specified and currently depends on the phone implementation.

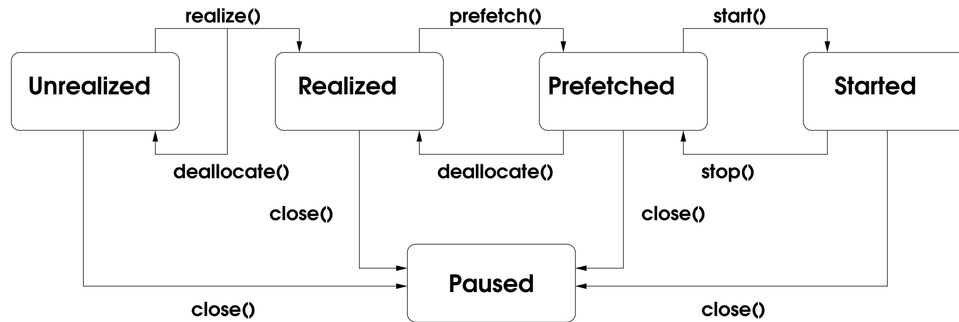

Figure 2.3: Player life-cycle

`startApp` is called when the midlet is started, and also each time the midlet exits the paused state. In practice, a global variable is usually used to distinguish those occurrences.

User interface

User interaction causes events and the AMS notifies the application of events by making corresponding callbacks. There are three kinds of user interface callbacks:

abstract commands of the high-level API: There is a package defining a set of high level widgets (forms, alerts, textboxes) that can register commands appearing as selectable soft keys or menus on the screen. Pressing such a soft key triggers an event.

low level events: There are classes for handling raw events and issuing graphic calls for drawing to the display.

application callbacks: The application can directly register callbacks in the event queue.

Callbacks are called serially. There is a single exception: when `Canvas.serviceRepaints()` is called, then the paint method is immediately called whether it is inside another callback or not.

Media player

JSR 135 [25] specifies a *Mobile Media API* (MMAPI) to handle external media players (sound, video, etc.) independently of the format used. MIDP2 specifies a subset of JSR 135 that should be available on any MIDP compliant device. A player is usually implemented as a closed class in the runtime environment following the `javax.microedition.media.Player` interface. A player has a life-cycle (figure 2.3) more complex than other midlets, as players may require data from an external source. A player may register an object of the `javax.microedition.media.PlayerListener`. The player listener callback will receive events for state changes and changes of parameters (volume, end of media, etc.).

2.5.2 Concurrency between different kinds of events

There is no reference in the specification to the concurrency model of the different kinds of event schedulers. The situation is summarized in figure 2.4. The current status is probably more the result of an overlooked issue in the specification rather than careful design decisions and should be dealt with in future releases of the MIDP specification.

2.5.3 Concurrency in a midlet

The CLDC configuration [31, 35], including MIDP, supports threads through the `java.lang.Thread` and `java.util.TimerTask` classes, including classical Java synchronization primitives (locking of objects) but

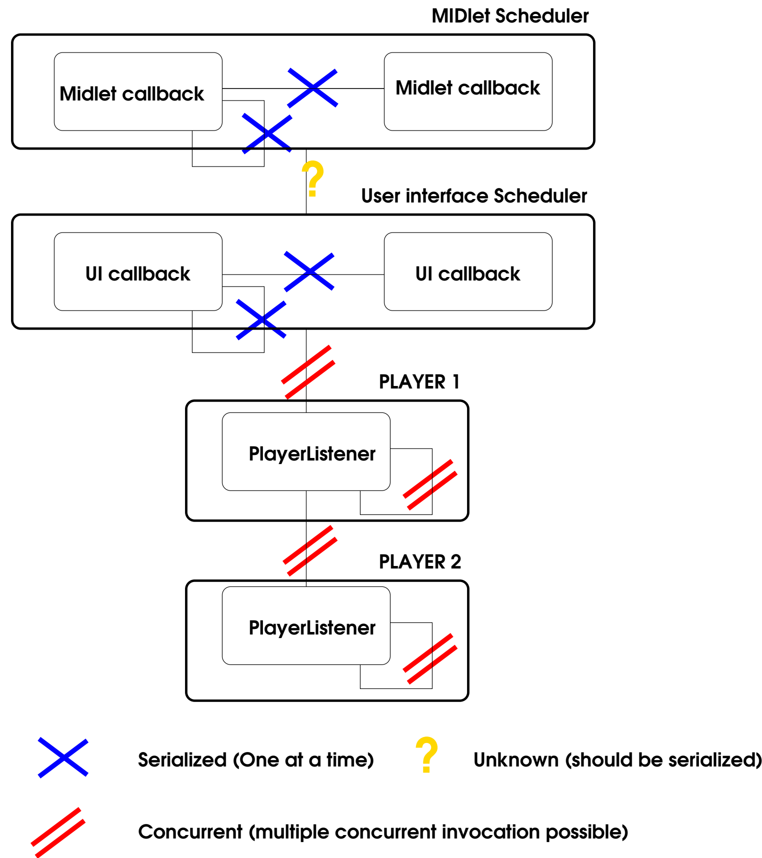

Figure 2.4: Concurrency between schedulers and callbacks

not including the notion of thread groups (handling of a group of threads as a unique object) or daemon threads (threads that are not counted to decide if the VM must stop).

There is no minimal or maximal number of threads specified in either the CLDC or MIDP specifications, but JTWI [28] specifies that an implementation of MIDP should support at least 10 concurrent threads. On the other hand, it makes the following recommendation for developers:

> Although many implementations can support more threads, MIDlet developers are encouraged to stay within these bounds. This requirement is not intended to require MIDP implementations guarantee at all times that 10 threads be possible, but that implementations not artificially limit thread creation to less than 10 threads. Application developers must still manage resource usage within the physical constraints of the device .

2.5.4 Concurrency patterns in MIDlet programming

The use of threads in a midlet can be arbitrary. It is possible to design midlets that exploit weird thread usage to create hidden information channels between those threads. Threads in a midlet may deadlock. On the other hand, it is reasonable to expect a midlet to follow some recommended usage patterns. We will first review cases where it is good practice to use threads.

User interface responsiveness

The MIDP style guide [12] states:

> Do not perform any task in a callback that has the potential to take a long time, such as network operations. Instead, arrange for the work to be done in another thread.

The example given in [46] shows how to use a worker thread to establish a network connection. The establishment of a connection (not only sending data) over GPRS networks can be very long and justifies this approach. A simpler approach is presented in [49] but it follows the same basic principle: network interaction must be performed on a specific thread.

With the Netbeans Mobility Pack, a SUN development environment for midlets, it is possible to request that the `commandListener` sends events to a specialized thread that handle all the events. The system thread won't be blocked but the action will not necessarily be performed immediately.

Midlet initialisation and worker thread

As a more generic programming tip, Sun recommends to use a separate thread for midlet initialisation (while a splash screen is displayed monitoring the progress of the initialisation) and a worker thread implementing the core of the computing performed by the midlet.

Complex timed graphical object display in 2D games

To program moving graphical objects in 2D games using Java game API (using sprites as supported by the `javax.microedition.lcdui.game.GameCanvas`), it is a common practice to use a thread loop that implements the delay between each movement and draws the new position of the graphical objects according to the changes in their internal state. The article [45] from the Sun online developer network is a simple example of this principle.

Reasonable usage of multi-threading for safe midlet programming

One of the advantage of Java programming environment is that it can be used to program security critical applications (banking, e-commerce, authentication, personal information handling) and multimedia/entertainment applications (support for sound, 3D and 2D animated graphics, asynchronous events). Both aspects can be used in a given midlet, but mixing them (using both aspects at the *same* time) is probably not a good idea because it is *confusing* for the user. Moreover a strict policy for thread usage during the critical phases of the application can help to avoid covert channels.

Chapter 3

Threat Model

The first stage to improve the security of a given service or application is the security analysis. The work to perform is to model the product and its environment, to identify the assets at risk and the cost of a potential attack. The goal of the security analysis is to help to choose the right security measures to apply and to justify the effort put on them and their cost. But the definition of the counter-measures is not part of this analysis. For an execution environment, examples of such measures are the definition of a security policy defining how the sensitive functionalities of the platform can be used to mitigate the risks and the means to enforce it.

The widely accepted standard in IT security, the Common Criteria [52] defines the notion of protection profile. A protection profile defines an implementation-independent set of security requirements and objectives for a category of products or systems which meet similar consumer needs for IT security. It also represents a consensus between vendors, certification authorities and users/consumers on the security objectives. Such a consensus requires a lot of time and work to be built and formalized.

Security analysis is also an experimental activity. Most inputs on threats for a given target come from real attacks that have happened on this or a similar technologies or proof-of-concept attacks that have been prototyped in security laboratories.

Although J2ME profiles are not the targets of any Common Criteria protection profile (But STIP environments used in banking applications could be the target of such a profile), the methodology defined in this standard are still informally applicable.

Today, MIDP technology is the only J2ME based technology deployed to such an extent that such a consensus has aroused (mainly between Sun Microsystems, manufacturers, telecommunication operators (who are in contact with the customers through the customer care centres) and application editors (who represent developers)). Although it is not clearly formalized anywhere, it is at the core of the MIDP recommended security policy for GSM/UMTS compliant device, addenda to the JSR 118 standard [23], of the security requirements expressed in the Unified Testing Initiative (a consortium defining a common testing procedure for J2ME MIDP applications) [42], or of the restrictions imposed on developers by various operators [21]. This justifies the choice of the Mobius project to focus on helping to enforce the existing security policies for the MIDP technology rather than define new ones.

This chapter will present the current state-of-the-art consensus on threat model for MIDP applications. Although MIDP and its security framework could be extended in multiple directions, it only takes into account the current capabilities offered by execution platforms and the practical usage made of those platforms. The focus has been put on threats relevant for information flow and resource security properties.

3.1 MIDP threat model

3.1.1 Assets to be protected

The assets are the same for all MIDP applications. They can be split between operator assets and end-user assets.

Operator assets

The main assets from the operator's point of view are as follows:

Billable events. An operator wants to be able to bill any event that should be. Billable events are an asset in the sense that they are usually associated to a cost.

Support infrastructure. The support infrastructures is costly. When something goes wrong with an application, end users have a tendency to call the operator's support, which has a significant cost for the operator.

Reputation. Another very important asset for an operator is its reputation. As the mobile market saturates, the competition between operators becomes fiercer, and their reputation becomes more important to attract new customers, or to avoid losing some of them.

Network infrastructure. Of course, most importantly, the network infrastructure is a very important asset for an operator. Any attack that can take down or cripple a significant part of an operator's network would be highly damaging.

End-user assets

The end-user, on the other hand, has a different view on assets. For him (her), the important things are:

Billed events. Events that are billed to an end-user are basically equivalent to money. The end-user does not want any application to steal money from him(her), in a way or another.

Private information. A mobile phone contains a large amount of private information, in particular, a list of contacts which the end-user does not want to disclose.

Mobile phone. Some attacks may attack the mobile phone itself, making it temporarily or permanently unusable. The end-user values its phone, and wants to protect it against such attacks.

Mobile phone data. The data in a mobile phone (contacts, messages, call history, *etc.*) is often not backed up. If the mobile phone comes under attack, and simply needs to be reset to its initial state, this data may be lost.

Synthesis

Despite the interests of the operators and end-users may seem quite different (except for billable events), they are in fact quite similar. The satisfaction of end-users is important for the operators (who continuously fight to reduce churn). For this reason, the assets to the end-user are also important to the operators.

3.1.2 Possible goals for attackers

Attackers have different motivations, which are outlined below:

Make money. For instance, an idea is to make an end-user to call a premium number, and then collect the premiums from the operator.

Steal sensitive information. In this case, the ultimate goal is usually to make money. One way to do that is, for instance, by selling valid contact information to a spammer after stealing it from somebody's contact list (commerce of information). Another way to do the same is by phishing attacks, *i.e.*, steal important information (for instance an account number and the accompanying PIN code), and to then use this information to empty the end user's bank account.

Attack an operator. In this case, the objective is to hurt the operator. The actual attacker may be motivated by money, or simply by hate. Any kind of harm is good: disturbing the network, making some devices unusable, triggering many billable events, basically anything that targets a particular operator.

Perform a hacker stunt. The release of malware may also be a way for a hacker to display its skills. Since the MIDP framework has rarely been attacked directly, any such attack would fall in this category. For instance, Adam Gowdiak's attack [22] has made the headlines of the security industry for a while, even though he was very careful about disclosure, and he did not attempt to spread malware in the wild.

Perform a terrorist act. As the phone infrastructure becomes a vital infrastructure in the economy, it can attract the attention of terrorists. Even though an unrelated terrorist act is sufficient to take a network down (as it happened in New York, Madrid, or London). Terrorists could target the network in order to make the recovery efforts less efficient; a viral attack in the middle of an emergency could definitely have an effect on the public.

Of course, even though all these goals and motivations must be envisaged seriously, some of these scenarios are more likely than others. If we look at the malware available on PC's, money has become the main motivation for malware writers, mostly in the form of phishing and spam. On mobile phones, there may be additional ways to make money, for instance through premium SMS messages or phone calls, so these channels must also be considered with care. Actually, they are the main concern of operators today, because they are in their direct responsibility.

In addition, some API's that are seldom used by today's applications (for instance the API's that access the phone's personal information) will become more widely available on the devices, and will therefore be used by more applications. This means that more attacks will be possible on the user's personal information as the disclosure of it to spammers[1].

3.1.3 Possible countermeasures

In the phone industry, there are still few attacks, in particular on MIDP technology. This means that there are some countermeasures that remain effective. The two strongest countermeasures are accidental, yet effective:

Market fragmentation. There are different kinds of mobile devices available on the market. Since most of them are not compatible with each other, attacks are always limited to a small part of the overall market, and thus making them inefficient.

Private networks. The telecom networks are not directly connected to Internet, and the operators have been very careful not to allow direct connections between devices, except in a few well-controlled situations. This makes it much harder to spread malware.

In fact, malware authors must face the same issues as all software authors: portability of mobile applications is a difficult issue, and the operators do not allow all kinds of connections to be performed on their networks. However, as applications become enable to perform more and more actions, and as the interoperability of devices gets better, these countermeasures are going to become less and less effective, requiring the emergence of new countermeasures. There are two kinds of possible countermeasures:

Increase the difficulty (and the cost) of an attack. For malware, the idea is to increase the level of controls imposed on applications to make more difficult to get malware on the phone. The work in the Mobius project falls in this category.

[1] There is a market today for "active" addresses, *i.e.*, addresses that are guaranteed to be correct, such as those stolen from a phone, since they are valuable to spammers.

Increase the cost if caught. Such countermeasures are not efficient against the most decided villains, but they can be very effective against casual hackers. Few people are ready to risk prison time just to show off their skills. The countermeasures usually consist of new laws, and also of contractual measures from the operators, in order to ensure that the perpetrators can be easily identified.

Something very interesting in the mobile phone industry is that the security issues are well identified, even before they can be implemented. This is a significant difference with the PC industry, where most of the malware "innovation" occurs. The phone manufacturers and mobile operators therefore have the opportunity to build security countermeasures inside their devices and development environments.

Some of these principles have been followed in the design of the MIDP framework, which is based on sound security principles. One of the most important of such principles is the fact that software needs to be signed (*i.e.*, authenticated, and/or verified) in order to be granted privileges. This means that some privileges can only be acquired after showing the software to some entity. However, it is difficult today to perform any systematic check during these verification phases.

The objective of Mobius is first to define algorithms that can be used to perform systematic checks on these applications, and then to make sure that these algorithms can be systematically applied on the applications, with as little constraints as possible on the various actors, and in particular on end-users and developers. This means that the systematic checks should at least be automated, and at best be included in the mobile phone's software, making them transparent to the end-user while providing an adequate level of protection.

3.2 Common attacks on information flow security

Information flow is not the item with the highest priority on the operator's agendas. However, in the foreseeable future, it will become increasingly important, as the openness of platforms increases. In terms of information flow, there are three issues to consider:

Disclosure of confidential data. This concern is very classical, and it exists in mobile applications. However, MIDP applications are often not trusted enough to hold important secrets. After a rapid survey of applications, it seems that the only secrets that they manipulate are the passwords used to access some Internet sites. Nevertheless, these passwords are important.

Disclosure of sensitive data. This issue is less critical, but it is important for end-users. Sensitive information include in particular all the private information stored in the phone, including the information stored in the Personal Information Manager (PIM), and the local files. The user expects to control any disclosure of this data.

Modification of sensitive data. The flow of incoming information must also be controlled, mostly because it can be used to corrupt sensitive data. This term is here taken in its broadest acceptance, covering the sensitive data described above, but also the data that is used to access local or remote resources.

These issues, their possible exploitation, and the countermeasures that can be envisaged are covered in the following sections.

3.2.1 Disclosure of confidential data

At the framework level, the main issue is to identify the data that should be considered confidential, independently of the application's specification. There is very little data that can be labelled as confidential:

- In MIDP, it is possible to label a text field as `PASSWORD` or `SENSITIVE`. Even in the second case, we can consider the value as confidential, since the definition of the `SENSITIVE` modifier is that the implementation of the platform must never store the value in a dictionary or table for any reason (history, entry prediction, *etc.*).

- In MIDP, it is possible to use secure communication protocols (SSL, HTTPS). Data sent over such secure connections can be considered as confidential, making any disclosure of the same data on an unencrypted channel illegal.
- The SATSA API [26] defines a way for applications to cipher data. Once a data has been ciphered, the input can be considered as confidential. Similarly, the result of a decipher operation can be considered as confidential. In both cases, such data should not be disclosed, or at least not communicated as clear text.

At the application level, it is possible to be much more precise. However, there are only few applications that handle confidential data. Among these, we have the following:

One-time password generators and other authentication applications. By nature, these applications store sensitive data (used to compute the passwords), and they also often handle confidential data (the main password/ PIN code, which makes it a 2-factor authentication method). These data need to be protected.

Secure password storage applications Although a mobile phone may not be the best place to store confidential data, such applications exist, because mobile phones are so ubiquitous. For such applications, whose main objective is secure storage, information flow is a very important thing to monitor, including the way in which information is stored.

In terms of countermeasures, there are several ways to protect confidential data, depending on the level of security required. A few typical countermeasures are provided below:

Confidential data shall not be sent over a connection. This view is simplistic, and sometimes too restrictive. However, it is the desirable property for all applications that are supposed to only use confidential data (such as passwords) locally.

Confidential data shall not be sent in clear over a connection. This view is slightly less simplistic and involves, for instance, the situation when a user enters a code that is then transferred to a Web site. This rule could be applied as a framework rule, in particular if the definition of confidential data is not too wide.

Confidential data shall not be stored locally. This rule is also quite simplistic, but it is a reasonable requirement in some areas such as banking, where the PIN code is an important asset.

Confidential data shall not be stored locally without being encrypted. Once again, encryption, when available, is a way keep data confidential. However, encryption must be used adequately, which is far from obvious.

Confidential data shall not be displayed. This requirement comes from the fact that mobile phones are often used in public places, where malevolent people can peek over somebody's shoulder.

In terms of enforcement, and in particular for static analysis, the main issues are (1) to identify confidential data, and (2) to identify what constitutes a disclosure. We have already discussed the first item. For the second one, the issue is a very typical one in the field of information flow analysis, and it has been covered extensively during the Mobius KOM. The issue is here to figure out whether or not the result of an operation that depends on confidential information, if disclosed, would constitute a disclosure of the confidential information.

The issue is here that the answer to this is necessarily application-dependent. One of the practical issues in Mobius will therefore be to figure out how the various operations available in the MIDP environment affect the confidentiality of confidential data.

3.2.2 Disclosure of sensitive data

At the framework level, sensitive data mostly consists of data that is accessible through specific APIs. Among the APIs that handle such sensitive data, the most significant ones are the following ones:

The PIM API. This API allows a MIDP application to read the list of contacts, but also the agenda items, and the to-do list. All these information is sensitive, and should be protected against disclosure.

The file system API. This API allows a MIDP application to access the device's native file system. Although the content of files cannot be systematically labelled as sensitive, some files undoubtedly contain sensitive information.

The location API. The location API allows a MIDP application to get information about the localization of the device (and of its user). This is of course private information, whose access needs to be controlled. However, this information is interesting, because its main use is to be sent over Internet to get location-specific information.

The multimedia API (for recording). The multimedia API allows MIDP applications to control the recording of media (sound, pictures, video). The result of a recording is private information, whose disclosure needs to be controlled.

The issue with sensitive data is that it is difficult to be as affirmative as with confidential data. Its disclosure is normal under some circumstances: PIM information when synchronizing with a PC or a server, files if they are used to store non-sensitive information, location when the user is looking for nearby restaurants, and multimedia recordings when the user sends pictures with comments in a MMS. On the other hand, it is easy to think of bad uses of the same information.

Countermeasures are therefore difficult to design at the framework level, and they need to be adapted at the application level. For instance, an application that provides information on local restaurants should be allowed to send location information (possibly over a secure link), and also to add new contact information (for an interesting restaurant); however, this application should not be allowed to access an existing contact, or to access any other PIM information, such as the agenda and the to-do list.

Here are a few examples of rules that could be applied on the PIM API:

PIM data is not sent over network connections. This can be the default rule, and the easiest way for a developer to show that its PIM data should not be misused.

PIN data is not sent over insecure network connections. This is a slightly more open rule, which can be interesting for applications that rightfully handle PIM data.

[To-do list | Contact | Event items are not [accessed | communicated].] The MIDP authorization system is coarse-grained, and it does not make any difference between these kinds of items. Adding this distinction can be helpful in order to design application-specific criteria.

[Private | Confidential items are not [accessed | communicated].] It is possible to classify the PIM items as public, private, or confidential. A sensible rule would therefore restrict usage, based on this classification.

Field *F* is not [accessed | communicated .] The information in PIM items is organized in fields, which are identified by predefined numbers. An application may only need to access some fields and not others.

There are many other possible rules, and most of them are application-specific. These rules often sit between access control (some feature is not used or accessed) and information flow control (the information obtained through some feature is not disclosed). It is also very difficult to define a fixed set of rules, as they are very tightly linked to the function offered by the application. However, in terms of information flow analysis, the issue remains the same as for confidential data. The data obtained through specific API's needs to be tracked, in order to check how it is used and/or disclosed.

3.2.3 Modification of sensitive data

In this last section we will use a much wider definition of sensitive data. This definition covers the data mentioned above, as well as the data used to access to sensitive functions and services. In particular, the parameters to communication functions are considered as sensitive.

At the framework level, we can consider the following data as sensitive:

PIM data. PIM data is personal, and very few applications need to modify contacts, events, and to-do lists.

File systems. File systems may contain all kind of sensitive data, so applications should only be allowed to access them on a strict need-to-modify basis.

URLs. URLs are also very sensitive data, because they determine the way in which the application communicates with the outside. The way in which URLs are computed therefore needs to be considered as the computation of a sensitive data.

For PIM data and file systems, the rules are mostly application-specific, and they are quite symmetrical with the rules on disclosure. On the other hand, for URLs, there are a few basic rules that should fit most applications (*i.e.*, at least one of the rules should apply):

URLs are constants defined in the code. This rule can apply to all applications that only communicate for fixed reasons (getting help screens, sending score SMS to fixed numbers, *etc.*).

URL protocols and domains are constants defined in the code. This rule is much less restrictive, but it is very useful in order to determine the responsibilities in case of problem with content. Since the domains are hardcoded into the code, the owners of the application and domain can be held liable for all problems.

URL protocols are constants in the code. This rule is very basic, and it is interesting in many cases. The objective is here to ensure that the protocol used in a connection can always be determined.

[**URLs | URL protocols | URL domains** are only computed from information contained in the application.] In this rule, the constraint about the computation is relaxed, in the sense that the information is not necessarily a constant. For instance, it may come from a localization file.

[**URLs | URL protocols | URL domains** are only computed from information contained in the application or entered by the user.] Here, the objective is to ensure that the user keeps control over the application, and in particular, that an application that uses Internet is not a "virtual browser", which downloads URLs to open from external connections.

These rules, and in particular the last two ones, clearly are related to information flow. However, the objective is not the same as in many cases. Here, the objective is to determine which information is used to build the URLs that determine how the application communicates with the outside.

3.3 Common attacks on resource control

For operators, resource control is very important, especially with resources that are associated to costs. In terms of resource control, there are two main issues to consider:

Abuse of billable events. An application may abuse of some billable events, for instance by repeatedly sending SMS messages to a premium number. The user must in theory confirm each operation, social engineering is always possible. Same remarks apply to the initiation of phone calls and to network accesses.

Memory usage. In a few cases, it is interesting to figure out the amount of memory used by an application. Memory is considered here in its widest acceptance, covering both persistent memory (record stores, files, *etc.*) and Java memory (stack, heap, *etc.*).

Control flow misuse. More generally, the way in which some resources are accessed can produce portability and performance problems.

If we consider some of the most important assets for the operator (support time, reputation), it is perfectly normal to include such properties in the requirements. Since it is difficult to make the application writers liable for the damage made by their application, enforcing a strong safety and security policy can be a way to at least make them accountable for the most classical safety issues.

In the rest of this section, we will therefore study those three types of issues.

3.3.1 Abuse of billable events

In MIDP's classical framework, there are three main types of billable events:

Short messages. Such messages are by far the easiest target for an attacker. First, there are plenty of good reasons to use short messages in applications (*e.g.*, to register scores to a server). In addition, sending such messages to premium numbers is a standard channel for application providers to make money. Finally, the user needs to confirm the sending of each messages, but there are many ways to implement social engineering strategies on these confirmations. For example, the user may be overwhelmed by innocuous requests, and then (when she is no longer taking them seriously) asked to authorise sending a message to a premium number.

Phone calls. There are important difference between phone calls and short messages. First, phone calls are far less common in applications. Then, when an application establishes a call, the phone number is rarely a variable, which makes all verifications easier.

Network accesses. Network accesses also are billable events, but it is more difficult for attackers to directly make money from them. On the other hand, network accesses are the basis for all kind of phishing attacks and other problems, and it is therefore interesting to monitor them.

If we also consider all the possible extensions to MIDP, there are many other APIs that could be considered as well, such as the following:

- The SIP API can be used to initiate many kinds of connections, including phone calls, and should therefore be monitored carefully.
- The SATSA API, although not directly a billable event, allows applications to access the SIM card, which is a crucial element in the operator's infrastructure (in particular for billing purposes).

In terms of countermeasures, all the events mentioned above are subject to permissions in the MIDP security policy. In some cases, such as the sending of SMS and MMS messages, the user is systematically warned before a message is sent, unless the application has been signed in a security domain with high privileges.

Additional countermeasures have been defined in the Unified Testing Criteria [42]. These countermeasures simply consisting in asking the developers about the kinds of connections the application establishes, as part of the application characteristics. The information required includes:

- Whether or not the application creates any phone connections, and to which numbers.
- Whether or not the application sends any messages, and to which numbers.
- Whether or not the application send e-mails, and to which addresses.

- Whether or not the application create network connection, and to which URL, and using which encryption method.

The document does not define, however, any way to verify that these declarations are correct. An additional countermeasure would therefore be to verify these declarations.

The declarations above can all be verified (at least in part) using static analysis. In all cases, the challenge is to identify that a method is invoked (or not) with a given set of arguments.

3.3.2 Memory usage

Memory usage monitoring is a more complex resource issue, since it is quite difficult to figure out the memory consumption of a Java application. There are at least three main issues to consider:

RMS usage. Applications can use record stores to keep some data persistent across sessions. This mechanism is practical, but it must not be abused: the space available is limited, and some implementations are very inefficient.

Memory usage. Applications use memory in different spaces: first, in the stack, in which method invocation frames are stored; then, in the heap, where objects are allocated. In both cases, the exact amount of memory consumed by an operation is specific to each implementation, but it is possible to approximate this consumption.

Memory release upon request (pause). When an application is paused, it should release its resources, and in particular its memory. The application should therefore take some actions in order to release resources when paused (close connections, discard references, *etc.*).

The issue of tracking the use of memory is quite classical. In Java, there are a few specific issues, which are all linked to the Java programming model:

Memory releases are performed by overwriting references. When an object is not referred by any reference, it can be discarded by the garbage collector. In Java, it is considered good practice to allocate objects that have a very short life (for instance, exceptions).

Factory methods are quite common. When factory methods are used, some object allocations are "hidden" in methods that may look like standard API methods. In addition, even when constructors are used, it is difficult to tell whether or not nested objects are allocated.

On the other hand, the MIDP framework does not support reflection, and it is therefore not likely to explicitly allocate objects of unknown type. Finally, as a practical framework, it is important to make a distinction between a container object (for instance a `Connection` object) and the system resource that it represents. For instance, in the case of a connection, some resources may be released when the connection is closed, and some other when the object is released. In addition, if the object is released before closing the connection, the system may not be able to garbage collect it.

Concerning memory usage, we don't really intend to define countermeasures as such. We rather intend to define ways to identify various uses of memory, in order to build a usage pattern for the application. It is quite difficult to define general policies based on these principles; on the other hand, it is quite feasible to define rules that can be included in an "application contract" on an application-specific basis. Such a contract could contain rules like the following ones:

The application uses at most 5 record stores at any time. Some applications use temporary record stores as persistent cache. It is sometimes difficult to track the number and content of these record stores, so a rule like that would help evaluators.

The application stores at most 5kb of information in record stores at any time. This rule puts a limit on the amount of information stored in the record stores, *i.e.*, in the total number of bytes in the records. Note that this figure is only an indication of the space required for the record stores, because of the variable overhead.

The application never manages more than 5 open connections at any time. Open connections are one way to consume memory. It is here mentioned as an example of API that needs to be carefully monitored in order to control memory consumption.

The application closes all outstanding connections when a pause is requested. Releasing resources is a requirement when the application is paused. It is therefore very important to release open connections, which represent at the same time a resource and a significant amount of memory.

With such rules, the actual countermeasure consists in checking the contracts provided with the applications: first, by checking that they are compatible with an operator's policy, and then, by checking that the application code complies to the declared contract.

3.3.3 Control flow misuse

It would be very interesting to define rules that can be included in individual applications in order to avoid portability and performance problems. By enforcing them, the developers could put forward the quality of their application. Some examples of those rules are provided below:

The application does not block the GUI thread. This rule simply verifies that no blocking API is used on the GUI thread. The objective is here to ensure that the GUI remain responsive even when a time-consuming operation is running.

The GUI thread cannot be blocked through synchronization. If threads are used properly, then they must be synchronized in order to avoid simultaneous accesses to shared resources (see rule below). The application shall ensure that no blocking operation is performed by a thread while it holds a synchronization lock on a resource that may be required by the GUI thread.

Concurrent accesses are protected by synchronization. If several threads may use the same resource (object), they must protect the accesses to this resource by using the synchronization features offered by Java.

In addition to these "mandatory" rules, which can be considered as actual countermeasures (some applications could be designed as denial-of-service attacks). In addition to these, there are many additional rules that can be used by individual application in order to claim their level of quality:

The application catches and processes all exceptions. This security measure is a requirement on smart cards, and the main motive is that an unexpected exception could result from an unsuccessful attack. It is therefore important in such a context to catch all exceptions, and to take any necessary countermeasure. Sensitive applications could apply similar measures.

The application closes all the connections it opens. Another very classical requirement, which shows that the application is well designed and written.

The objective is making a large number of low-impact rules available to developers is to allow them to use sensitive features, while being able to prove that they make good use of these features. Although it may be less important than the rules that are systematically applied on all applications, it may be an important step to foster innovation.

Chapter 4

Information Flow

This chapter considers information security requirements. We begin by identifying relevant sensitive information (Section 4.1), its sources (Section 4.2), its potential consumers (Section 4.3) and hidden channels that may transfer it (Section 4.4) on the running examples of the MIDP profile. These are then illustrated by specific scenarios for information flow (Sections 4.5– 4.7). The final part (Section 4.8) investigates requirements for information flow policies, and how formal analysis can guarantee that programs abide by such policies.

4.1 What are we trying to protect?

4.1.1 Who are the malicious actors?

J2ME dynamic security policy is targeted to protect the end-user of the application. It is not suitable to enforce DRM (for example to prove that the user cannot bypass the rights given to him for using some data retrieved by the midlet)[1] or to protect the network (for example against a coordinated attack creating a denial of service).

A policy implemented purely by static checks cannot be a substitute to a dynamically enforced policy when the user is the main malicious actor. So we will not extend the scope of the study to such rules. Static information-flow security policy will have the only objective to protect the end-user.

Information flow policies can be used to track confidentiality or integrity. We will mainly concentrate on confidentiality issues. Integrity policies could be used to check that a given data is computed from safe sources (not from the network usually).

In our model, the attacker will be the developer who consciously or not has written a midlet that manipulates user assets in a way that leaks information to untrusted parties. Those will be either entities on the network or other applications.

4.1.2 What kind of information flow policy?

The main goal of this report is to identify known critical sources (objects that produce confidential information) and untrusted sinks (objects that should not consume confidential information).

MIDP provides some basic cryptographic operations. Some are in the core specification but most of them are in the complementary SATSA library. Encrypting a piece of information is considered as a *declassification* operation. Declassification means that the result can be released to anyone. Declassification of encryption results is safe provided the attacker may not guess secret keys or break the cipher.

In fact, all the keys that are used are external and so is their meaning. In the context of industrial automatic validation, this cannot be taken into account. If the validation is more involved with human

[1] It may change in future version of MIDP that could interact with native DRM protection mechanisms. The effectiveness of such schemes is still to be proven.

intervention (suitable for very critical application like banking), then the level of automation does not need to be as high.

4.2 Sensitive information sources

4.2.1 Sensitive APIs

In this category, we group APIs that access data naturally personal whatever is the context of their use.

Access to the personal phonebook and to the diary

Those APIs are part of the JSR75 [37] (extensions for PDA, now available on some high-end phones). The data are organised in records and fields are accessed by standardised names. It is important to understand how some of the fields are used to check if there is a privacy issue or not: addresses can be used to create URLs that will be used to establish connections, but they can also be stolen by a malicious program trying to send the list of contacts of the victim to a remote site. As seen later a URL can contain parameters.

Main methods and class

`javax.microedition.pim.PIM` the main class, an instance is obtained with the `getInstance` method

`openPIMList(int type,int mode)` opens one of the PIM list (contact, event or todo list).

`<PIMList> Enumeration items(...)` gets an enumeration of items either matching a field, an item or all

`PIMItem.getString(int field, int index)` gets the string value of a field at a given index (there may be more than one value). Field names are coded by number defined in the PIM specification. There are many other accessors for other types (Date, int, boolean).

`PIMItem.setString(int field, int index, int attributes, String value)` sets the string value of a given field at a given index with attributes specifying the meaning of data (ex work, mobile or home for a phone number).

Geo-localisation

Geo-localisation is the operation of computing the geographical position of the handset. It should be done using the standardised API defined in JSR 185 [27]. Approximate geo-localisation can also be performed using information on the identity of base stations the handset is connected to and the power of their signal. Those information are available in proprietary handset specific system properties (mostly Siemens phones). Proprietary system properties are dealt with later.

The information obtained is personal and should only be used by trusted servers because one could use it to spy the position of the terminal owner (in fact companies shipping parcels use such programs to monitor the position of their fleets of trucks).

Main methods and classes The package used is `javax.microedition.location`. The main methods are

`Location LocationProvider.getLocation()` get the current position of the terminal.

`LocationProvider.setLocationListener(LocationListener lst, int interval, int timeout, int maxAge)` registers a listener that will track the position of the terminal.

`LocationListener.locationUpdated(LocationProvider prov, Location loc)` the method called when the location listener is invoked.

Multimedia recorder

The MMAPI specification (JSR 135) introduces the notion of player. A player can be used to handle (play or record) multimedia contents. On a phone it can be used to take still pictures or videos with the camera or to record sound from the microphone depending on the capacity of the handset.

The data produced by those APIs is personal information and should not be sent on the network without a user explicit approval.

Main methods and classes for audio capture

`Manager.createPlayer("capture://audio")` gets the player for recording an audio sample from the microphone.

`Controllable.getControl("RecordControl")` gets the control over the player.

`RecordControl.setRecordLocation(String locator)` sets the output of the recording to a given output stream.

`RecordControl.startRecord()` starts the recording.

`RecordControl.commit()` finishes the recording.

Main methods and classes for video capture

`Manager.createPlayer("capture://video")` gets the player for recording a video from the camera.

`Controllable.getControl("VideoControl")` gets the control over the player.

`byte[] VideoControl.getSnapshot(String type)` takes a photo. Note that the type is defined by the video encoding standard (an example is `"encoding=jpeg&width=320&height=240"`).

System properties carrying private information

Some handset offer an access to otherwise private system properties: IMEI (unique identifier, usable as a unique user identifier), MSISDN, (user phone number), IMSI (subscriber and operator unique identification), name of nearby base station, etc.

Even if those information are not standard yet, one should take care of the security implication for revealing them to untrusted source.

`<java.lang.System> String getProperty(String name)` gets a named property

Access to the SIM card

The SATSA specification (JSR 177 [26]) opens new possibilities to access the SIM card. Such applications are usually very critical and must be signed by the operator to enable the access to the smart card. There is also a risk of blocking the card if an incorrect operation is performed.

The smart card can be used to encrypt or decrypt data, sign documents. The control of its use is probably beyond the scope of static analysis (see application specific policies). Moreover only tightly controlled operator or manufacturer midlets should have access to the SIM card.

`Connector.open` A connection to the SIM is a regular connection opened either in APDU mode or JavaCard RMI mode. The first one is much simpler and will be the only one supported by most implementations.

- in APDU mode (example URL is `"apdu:0;target=<aid>"` where `<aid>` is the application identifier and 0 is the slot number). The object is an `javax.microedition.apdu.APDUConnection`. The main function is `byte[] exchangeAPDU(byte [])`. Other functions are used to handle PIN identified by an id.

- in RMI mode (`"jcrmi:0;aid=<aid>"`) The object is an `javax.microedition.jcrmi.JCRMIConnection`. In that case objects in JavaCard are directly handled through class `javax.microedition.jcrmi.RemoteRef` objects that define a method `Object invoke(String method,Object [] params)`.

`javax.microedition.securityservice.CMSMessageSignatureService` is a class providing a signature service based on a security element (WIM). It provides the methods `authenticate` or `sign` to sign a given text or authenticate oneself using PKI.

4.2.2 Application specific data

The most sensitive information may not be produced by a critical API but be the result from the user input. Such an example is an application requesting the credit card number to do a transaction.

In the MIDP high-level user interface, there are only a very limited number of APIs to let the user input some data. In the low-level interface, it is always possible to rewrite a complete form library using keyboard events and drawing primitives on the canvas. As it is not a good programming practice, it is reasonable to forbid the use of such keyboard events to build data that will be sent to a dangerous data receiver.

One of the main problem in automatic analysis is to link the field displayed on a screen where the user enters his personal data and the method call in the source code that created that field. There are several solutions:

- analyse globally every input fields and let the evaluator decide which ones are critical;

- use hints such as the SENSITIVE (the meaning is that the data should not be cached by the handset) or PASSWORD (characters typed are not displayed) attribute of TextField objects: unfortunately there is no direct visual effect of making a field sensitive (there is no visual lock change as for an HTTPS page in a web browser) so the user cannot guarantee the link between the sensitiveness of data and the marking in the fields of the forms.

Main methods and classes

`TextField(String label, String text, int maxSize, int constraints)` This is the constructor for a text field in a form with constraints being an hint specifying the kind of contents (PASSWORD or SENSITIVE for example).

`TextBox(String label, String text, int maxSize, int constraints)` This is the constructor for a text box, a kind of `Screen` object where the user can enter data.

`TextField.getString()` **and** `TextBox.getString()` to access the contents of a text field or box.

4.3 Dangerous information consumer

4.3.1 Persistent store

General case

A RMS (record management system) is a small database of records created on the handset to store persistent data associated to a midlet. It has been advocated that it is dangerous to store very sensitive information such as credit card numbers on a mobile phone because it may be stolen.

Shared RMS

A shared RMS is a RMS accessible by all MIDP applications. A shared RMS is declared as such (Boolean argument in the constructor) when it is created and is referred by the name of the midlet suite that created it and the local name of the store.

Any kind of sensitive information should not be stored in a shared RMS.

Methods and classes

`openRecordStore(String name, boolean create, int authmode, boolean writeable)` to create a new record store. `authmode` determines whether the record store is shared or private.

`addRecord(byte [] data, int offset, int size)` to add a new record.

`setRecord(int id, byte [] data, int offset, int size)` to change a record value.

`byte [] getRecord(int id)` to access the value, there is another variant with a predefined buffer.

4.3.2 Network access

To create a connection, the application must call the method `Connector.open` with an URL describing the protocol, the address and eventually some parameters to create an object belonging to a subclass of the Connection class. This object contains methods to build the payload and perform the actual data transfer.

Functions building the payload (write for stream-oriented connection or send for message-oriented connection) are obvious data consumers. But the URL itself may convey information to an untrusted source (for example when one use parameters to a CGI in an HTTP GET request) .

Depending on the scheme in the URL, the connection established is either local (eg: irda, bluetooth, serial port) or distant (SMS, HTTP). The level of confidence in the destination will certainly depend on its location .

Methods and classes

`Connector.open("scheme:// ...")` where `scheme` is either `"http"` or ”https” (it is also possible to use raw sockets on some networks). The result is either an `javax.microedition.HttpConnection` object or an `javax.microedition.HttpsConnection` object.

`InputStream InputConnection.openInputStream()` creates an actual input stream to receive data. A variant is `InputStream InputConnection.openDataInputStream()`.

`OutputStream OutputConnection.openOutputStream()` creates an actual output stream to send data. A variant is `OutputStream OutputConnection.openDataOutputStream()`. There are direct methods in the `javax.microedition.io.Connector` class: `InputStream openInputStream(String url)` and `DataInputStream openDataInputStream(String url)`

`OutputStream.write(byte[])`

`OutputStream.write(byte[],int o, int l)`

`OutputStream.write(int b)` These are raw functions to send data; DataOutputStream contains more functions. There are also methods in the `javax.microedition.io.Connector` class: `OutputStream openOutputStream(String url)` and `DataOutputStream openDataOutputStream(String url)`.

4.4 Potential hidden channels

The goal of this section is to make a rapid survey of how information could be conveyed from a sensitive producer to an untrusted consumer without being tracked by a careful information flow analyzer treating sequential Java programs.

- information flows can be transmitted through careful use of timing between threads (this is known as covert channel),
- information flows can go through native APIs leaving the scope of the Java program to reenter it later,
- thread communication and use of native API can be mixed.

4.4.1 Covert channels

Threads can be used by a malicious programmers to convey information between two otherwise independent parts of a program (see 4.8.1). It is sometimes possible to avoid such channels by ensuring that there is only a limited number of threads running and that their synchronisation is controlled (see 2.5.4).

4.4.2 Taking native store area into account

in section 4.3, we explored how sensitive local storage area could be accessed and we were mainly interested in what was retrieved from the store (application seen as a consumer). But even if a local store should not contain sensitive data, it can be used to *exchange* data between midlets with different security levels, rights on the network access, etc. The goal of an attacker is to hide that there is a data flow from a sensitive data producer to an untrusted data sink. An external storage can be used to "wash" the annotation that data are tainted as confidential.

The use of such APIs as intermediate storage falls outside the scope of Java policy. In that case, we are interested by applications acting both as a consumer and a producer of the information stored in the local storage (and it is usually easier to identify a security threat on the producer side).

Use of sensitive local stores as intermediate storage

RMS As seen in the previous paragraph, the store can be shared among different midlet suites. An unshared store can also be used by an application to "wash" tainted data.

Access to PIM data As seen in 4.2, phones usually natively handle contacts and events lists. The JSR75 lets a program read and write those data but its access is heavily controlled and so it is not a very good intermediate storage from an attacker point of view.

Use of other native stores as intermediate storage

The Landmark store for geo-localisation This is another external storage specified in JSR 179. The landmark store is used to store locations that can be compared with the current location, but it can be used by a malicious application to store any kind of textual information. Note that the landmark store is also protected by a specific permission group but there is less reason to have a tight control over its use.

`static void createLandmarkStore(String storeName)` create a store (similar to a *shared* database creation). The store may also be shared with native applications.

`void addLandmark(Landmark landmark, String category)` add a new landmark.

`java.util.Enumeration getLandmarks(String category, String name)` access to a sequence of landmarks fullfilling a given condition. There are other more precise accessors.

`Landmark(String name, String description,QualifiedCoordinates coordinates, AddressInfo addressInfo)` Constructor for Landmark objects. Most fields can contain unconstrained string elements

Various string storage in `javax.microedition.lcdui` Graphical objects of the high level interface usually contain text to identify the components (labels, initial contents, etc.). Those values can usually be stored and queried. But those methods and objects are usually writen in pure Java code in the libraries (at least if the profile used is not a stub).

Images can be defined as mutable and are accessible (`Image.getRGB(int [] data, int offset, int scanlength, int x, int y, int w, int h)`). This is not the case for the screen contents that can be written but not read. Images or raw screen contents are native objects in most implementations of the MIDP profile.

4.4.3 State listeners

Several APIs propose a listener object to listen to state changes in objects such as media players, RMS, etc. Those object change states as reaction to action performed by calling other methods. It is then possible to establish a communication link between a thread using the object and another one listening to its state change. This is similar to timing sensitive covert channels (see section 4.4.1) but the MIDP implementation can also relie on some native mechanism to implement the state change that can be used by attackers to convey information.

As an example, section 2.5.1 presents the life cycle of a multimedia player with the function `realize`, `prefetch`, `start`, `stop` and `deallocate` in class `javax.microedition.media.Player` to change the current state of the player.

The following events should be generated in response to state changes : `STARTED` and `STOPPED` in `javax.microedition.media.PlayerListener` class. Those state changes can also be seen by querying the state directly with `Player.getState`. The result uses a different enumeration: `UNREALIZED`, `REALIZED`, `PREFETCHED`, `STARTED` and `CLOSED`.

4.5 A scenario for information flow analysis: an itinerary planner

This is an example of midlet with true concurrency in a first part and that will legitimately evolve to a more restricted user interaction when payment will be dealt with.

Google has released Google Local for Mobile. This is a version optimised for mobile handsets of the well known Google Earth desktop application and equivalent to ESRI services.

The basic service provided is a browsable map on which one can add layers. Layers represent a type of information like roads, positions of interesting sites, etc. First designed for geography institutes those systems are now reaching a wider audience and new layers like the position of stores, restaurants or hotel are being developed. With this new kinds of layers appear new opportunities for integrated services shared among different providers.

We present such a potential but yet imaginary service: today there is no use made of the internal geo-localisation API and there is no link between maps, itinerary planner and an hotel/restaurant booking services. However, as web-services standards evolve, the integration of such services should improve and the idea of an integrated services going from planning the itinerary to booking and paying the accommodation should become true.

4.5.1 Step 1: Identify your current position

The midlet uses geo-localisation (JSR 179) to get the position of the user. It initialises the context with this value. An animation is displayed during this time and the user can select the level of details he wants (restaurant/hotels/price category).

Figure 4.1: Threads involved in the itinerary planner scenario

4.5.2 Step 2: Plan your route

The midlet asks the user if he wants to find an itinerary toward a location or information around his current location. Those information are sent to GoogleEarth servers. A plan is given back and as the user moves his cursor around, some information on near-by restaurants/hotels are given (optimised image retrieval with worker threads handling the communication). The implementation would probably use a `GameCanvas` object.

4.5.3 Step 3: Select an hotel

The user selects some of the hotels displayed on the map and asks for a room. A communication with a booking service managing those hotels is established and pricing/availability information are given. At some point a hotel is selected. There could be more than one connection to more than one database. Database could give back small videos or audio clips (JSR135 player usage).

4.5.4 Step 4: Booking a room

Booking is a billable service using Premium SMS to charge the customer. The midlet sends the booking order to a SMS+ server. A ticket representing the transaction is generated.

A payment is initialised with a last server representing the bank (it should rather be a cancelable guarantee rather than a payment, but this is outside the scope of the scenario).

There is a single worker thread handling all the processing. The UI thread is limited to handling user interaction and displaying progress information. The payment could use the SIM card (using JSR 177 API) to authenticate the user.

4.5.5 Information flow constraints

The following constraints should be verified by the application:

- There should be no other personal information communicated to the hotel database or map servers (the program does not access any other source of private information);
- The location of the user should only be used for the itinerary computation or the map request (see 4.2.1).
- Information sent to the hotel database must be limited to what is strictly needed for booking a room;
- The credit card information should be taken from a sensitive text field and only be sent to the bank (see 4.2.2);
- When the payment thread is active, the only other active thread is the user interface thread (see 4.4.1).

4.6 Detailed scenario for information flow

This scenario corresponds to a small subset of the previous one. Its objective is to show the difficulties that are commonly encountered when analysing a simple application. It is based on an open source MIDP application (http://www.sreid.org/GMapViewer/), whose objective is to display Google-generated maps on mobile phones. The main advantage of this application is that it is a well-written open source application, which makes it easy to use as an example.

The application is very simple: it connects to a Web site, gets some map information, and displays it on the mobile phone. It also offers a few controls, which let the user move around on the map, or go to a specific point. It is a good example of an application that uses a network connection to do simple things.

The scenario illustrates that tracking information flow is important even for such simple applications.

4.6.1 Desired security properties

In this example, the objective would be to prove that the following statement is correct: "The application can only access hosts that are specified in the application's binary code, or that are provided by the end-user." The intent of this statement is to check that the application only does what it is intended to, and that it does not unexpectedly browse the Internet.

This property is quite simple to prove. We simply need to analyse the code, and look for places where connections are opened, *i.e.*, where the `Connector.open` method is invoked. In the program, there is a single invocation of this method:

```
String url = prefs.getString("gatewayURL", "")
   + "?p=" + prefs.getString("gatewayAuth", "")
   + "&a=" + action + params;
conn = (ContentConnection)Connector.open(url);
```

This is a fairly typical code, in which the URL is built by combining information that comes from several sources. Here, the beginning of the URL (which should contain the protocol and host names) seems to be getting from the object prefs, which seems to be some kind of object managing properties.

Indeed, this object is an instance of class Preferences, which is used to manage the user's preferences in this application. This class extends another local class Properties, which in turn is an extension of the standard Hashtable class, by adding to it a few additional features. In particular, the code to get an item is as follows:

```
public Object get(Object key) {
        if (containsKey(key)) {
                return super.get(key);
        }
        else {
                return app.getAppProperty("prefs." + (String)key);
        }
}
```

If a key is not associated to any object in the table, then the application looks up the application properties (from the descriptor file). Indeed, the descriptor file defines the property that we are interested in:

```
prefs.gatewayURL: http://www.sreid.org/gmapgw.php
```

It is possible to retrieve this property using static analysis, in particular since the hashtable key appears as a constant in the code. However, the real difficulty is to figure what the content of this hashtable may be. Generally, this content depends on information flow of data in the application. In this application, there are two ways to modify it.

The first way is in the user interface, which includes a dialogue for selecting the preferences. When the user confirms its preferences, the following happens:

```
private void savePrefs() {
        app.prefs.put("gatewayURL", gatewayURL.getString());
...
        app.prefs.savePreferences();
}
```

In this code, the gatewayURL field is a text field, which means that, in this particular case, the value affected in the hashtable has been provided by the user (which is acceptable for our criteria). However, the difficulty is here to prove that this element of the hashtable cannot be modified otherwise:

- All modifications must be tracked, and associated to a given key.
- If some modifications are performed on unknown keys, then it must be possible to prove that the elements we are interested in is not modified.

The difficulty is here that there must be a very precise representation of the possible content of hashtables, which can be resource-consuming. However, the second ways to modify the hashtable makes things even more difficult.

The Preferences class defines a way for the application to save the user preferences in a record store, and to later retrieve them. The hashtable is therefore initialised in an open loop:

```
public synchronized void load(DataInput in) throws IOException {
        int size = in.readInt();
```

```
            for (int i = 0; i < size; i++) {
                    String key = in.readUTF();
                    String val = in.readUTF();
                    put(key, val);
            }
    }
```

Here, in order to be successful, the analysis must have a good representation of the record store, and be able to recall the properties of the data saved in it, which is quite difficult.

4.6.2 Requirements

This scenario is quite representative of the practical problems faced by the static analysis of MIDP applications. Typical issues include the following:

Take all available information into account. This information includes the properties defined in the application's manifest, as well as the content of the resource files included in the distribution file.

Be able to manage complex containers. The interesting information is often stored in complex containers, such as arrays, vectors, and hashtables. Those are used to store the preferences, the list of last used files, and many other things. It must be possible to retrieve the information from these containers, if the developer programs correctly.

Take the persistent store into consideration. MIDP applications are used repeatedly, which means that they often have a persistent state, which may be saved in record stores, files, or even PIM entries. In all cases, it must be possible to take this information into consideration in the analysis

These requirements are not expected to cause major theoretical problems. However, they cause major methodological and practical problems, since the abstractions used in the information-flow analysis must be very precise, while remaining efficient. In addition, the API modelling work must not be too complex.

4.7 A scenario for information flow analysis: a friend finder

We provide a description of a J2ME application which handles sensitive data: a friend finder.

4.7.1 Description

This is an example of a midlet which transfers varying amount of sensitive data in different parts of its work flow.

The friend finder is a mobile service provided by some mobile (GSM, 3G) operators. The service allows an individual to find people who share some common characteristics and are located in close proximity to the user. After payment, the user is able to chat with the selected users.

The application uses location services (either local — JSR 179, or server side) to get the position of the user and interaction with the server side to locate other users.

The application allows several alternative usage scenarios which have different event/information flows. Only one of the alternative flows is described here. Other flows are stripped down variants and are omitted as less interesting.

4.7.2 Where the information is stored

In this scenario, information of several different kinds (and sensitivity levels) is processed, sent and stored.

To make the information flow more readable, here is the enumeration of different pieces of information which is processed, sent and stored at various steps of the information flow scenario:

1. user personal data — including selection criteria, visibility information; the data is read by the midlet from the user, stored locally and sent to the friend finder server;

   Although this kind of information does not need very high level of protection, it should only be sent to the trusted server and only in the appropriate phases of the scenario.

2. content of the chat sessions — information exchanged between the users when the system allows them to enter chat sessions; each message should be sent only to the user it is directed to;

   In general the contents is sensitive information, but of middle level of security, as the application is intended for entertainment use.

3. user location information — the information about locations of the users using the friend finder service; the information is either generated by the midlet itself (using geo-localisation API — JSR 179) or it is generated by the location server at the mobile operator side and transferred to the friend finder server;

   This information should be provided in a controlled way to (a subset of) the users of the service; this information has high level of sensitivity.

4. payment information — the information is read by the midlet from the user, stored locally and transferred only to the payment server;

   This information has the highest level of sensitivity as improper management may cause direct financial loss of the application user.

4.7.3 The flow

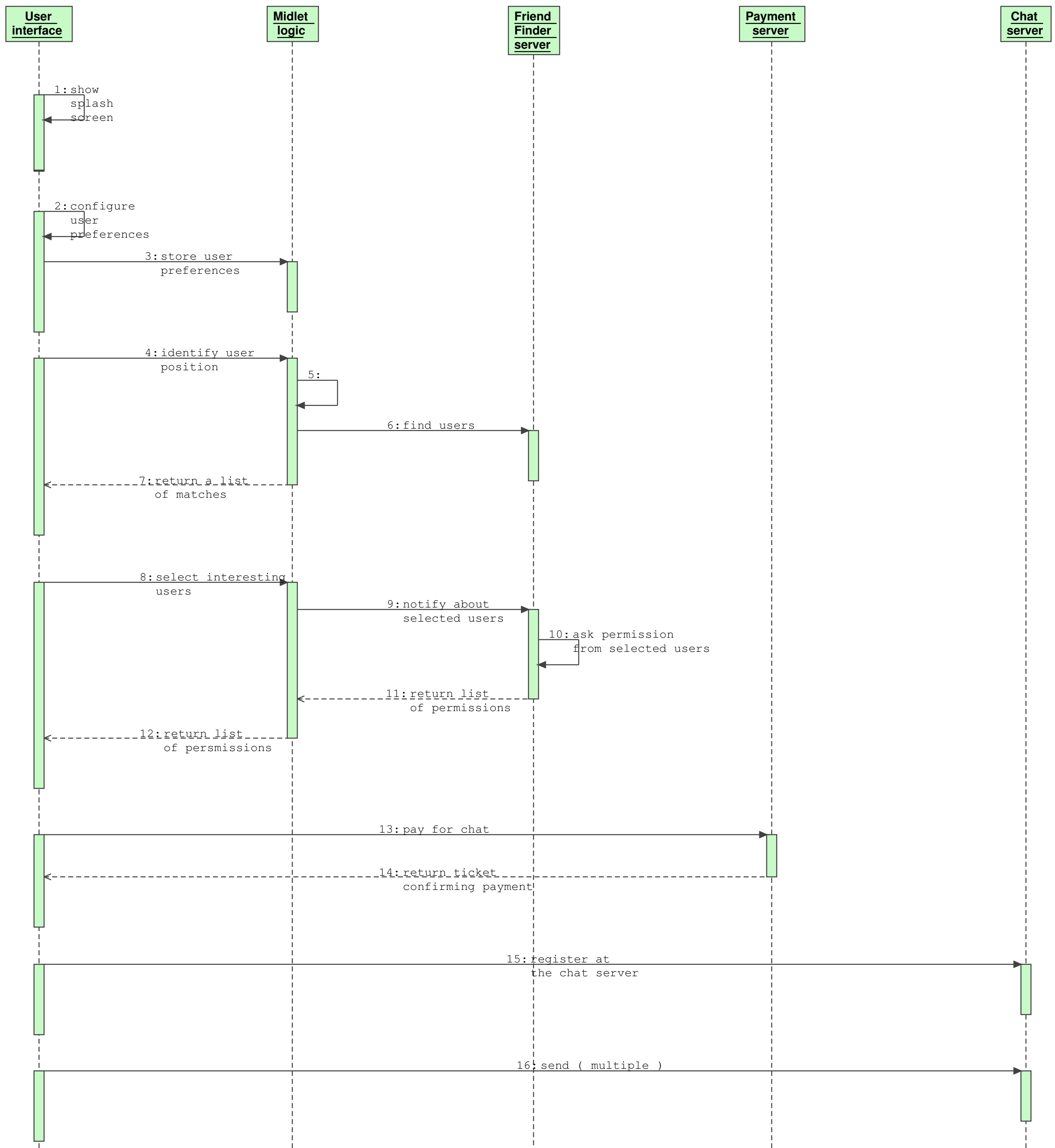

Figure 4.2: Activities involved in the friend finder scenario

Step 0: Configure the user preferences

The following information is required for setting up the friend finder service.

- basic personal data — a subset of the following set: sex, age, height, weight, hair colour,

- a short personal description — a free form text,
- personal interests — a selection of one or more parameters from the set predefined by the application (for instance: "likes films", "likes punk music"),
- selection criteria — a selection of one or more parameters from the set predefined by the application ("interested in meeting people who like punk music", "interested in meeting girls"),
- visibility information – information whether to allow to be searched for by other users of the service and whether the personal information should be given without authorisation.

The applications is stored locally by the midlet. When the application is used, the information is read from local storage and sent to the server. This kind of information does not need high level of security.

Step 1: Identify the current user position

The application identifies the user position using the geo-localisation API. In case the mobile device lacks this API, a request to the operator side location platform is made. The process takes a few seconds; an entertaining animation or a user friendly message can be displayed during this time.

If the information was obtained on the device (using the geo-localisation API), it is subsequently sent to the server for the friend finder service. If server side localisation mechanisms were used, the information is only transfered between the location server and friend finder server (both located at the mobile operator).

Step 2: Send the selection criteria

The midlet sends the location information and the selection criteria to the server of the friend finder service as the body of the request to locate other users. During this time, a message or animation may be displayed to the user.

Step 3: Find other users

After receiving the selection criteria, the server selects the users who are located close enough and who match the search criteria. The server sends back the information about each of the matching users. If there are too many matches, the server sends only a subset of data.

Step 4: Select interesting users

The application displays the list of found matches. For each match the application displays whether the matched user requires that personal information be sent back to him.

The user selects the matches to contact (using checkboxes or another similar GUI mechanism). Once the selection is complete, the user confirms sending his personal information if necessary (once for all selected matches). The application sends the information to the friend finder server.

Step 5: Obtain authorisation from matched users

The server side application sends request for authorisation to each of the selected users.

The midlet of each user displays authorisation request (together with the personal data of the asking user, if the data were requested). Each user either accepts or denies the request. The answer is sent back to the friend finder server.

Step 6: Pay for chat

The user gets back the information which users want to chat with him. As the time taken by each user to answer the authorisation request may vary widely, the answer is returned after a predefined time and the user may ask the midlet to refresh the list so that later answers from matched users can be shown.

The user selects which of those he wants to chat with. The midlet sends the payment request to the payment server and gives the number of units to pay for (the user pays for chatting with each selected user). The payment mechanism used depends on the mobile operator infrastructure and may for instance be based on sending a Premium Rate SMS or on accepting the advice of charge provided by the server (if the user has a postpaid billing relationship with the operator). The payment could also use the SIM card and the JSR 177 API for user authentication.

If the payment is successful, the payment server sends back a ticket to the midlet. The ticket contains a cryptographically protected guarantee of the transaction which can be verified by the chat server.

The midlet sends the ticket to the chat server, together with the list identifying the users to chat with. The chat server verifies the ticket, obtains the user IP addresses from the friend finder server and opens the chat sessions.

Step 7: Chat

The user midlet sends chat messages to the chat server and displays back the answers.

All chat sessions are performed concurrently.

4.7.4 Information flow constraints

1. The location information is sent only to the friend finder service (if the location is determined on the midlet side). In step 1 only the location information is sent.

2. All personal data and location data can be sent to the friend finder server application. In step 2, no other information can be sent; no other servers than the friend finder server must be contacted.

   The identity of the server side application has to be established using cryptographic means.

3. In step 4, only the list of selected users can be sent and only to the friend finder server. No other information may be sent by the midlet, no other servers may be contacted. Again, the friend finder server identity should be verified.

4. In step 5, the midlet should not receive information from any other application than the friend finder application. Only the binary reply (confirm, reject) or the request to refresh the information can be sent to the friend finder server.

5. In step 6, the payment exchange is performed only with the payment server. No other server may be contacted in this step, no other information than required to make the payment may be transferred.

6. In step 7, only chat messages are exchanged with the chat server. No other servers may be contacted in this step.

7. Each chat thread exchanges only the information related to the particular chat session.

8. The authorisation requests (step 5) are handled concurrently with other activities.

9. In each of the steps 1–6, the application has only two threads of control active: one worker thread for GUI actions and another worker thread for network communication.

   In step 7 (chat), the application may have multiple concurrent GUI threads and the corresponding network communication threads active. The implementation may use a single GUI thread for all chat sessions or have multiple GUI threads for multiple sessions. Network communication will be optimally

served by a pair of threads (one for sending messages from all chat sessions, one for receiving messages for all chat sessions); use of multiple threads for multiple chat sessions is possible but technically unjustified.

The discussion concerning the "application can only contact the appropriate hosts" presented in section 4.6.1 applies directly to this case.

It should also be noticed that the confidentiality of information sent by the midlet to the servers engaged in the communication flow depends critically on the behaviour of the server side applications which receive and process the information. Therefore each proof that confidential information is only sent between the appropriate parties and disclosed according to the requirements must use the model incorporating the security assumptions on the server side applications; for this purpose models developed to formalise multiparty security protocols may prove useful.

4.8 Approaches to formalization

As seen in the previous sections, there are realistic scenarios where information flow controls are highly desirable. Yet these controls are unsupported by current security mechanisms. The goal of this section is to outline possible definitions of secure information flow in the context of global computing, and to indicate means of enforcing these statements. We identify several attacker models, that shall lead to different non-interference policies, some of which allow intentional information release, and see how both type systems (and related techniques such as static analyses and abstract interpretations) and program logics can be used to guarantee that programs abide to these policies. It is the objective of Tasks 2.1, 2.3, and 3.2 to further develop the ideas presented in this section.

4.8.1 Defining secure information flow

Non-interference is a high level property that aims at ensuring that programs do not reveal to attackers any information about secret data during their execution. More concretely, it guarantees that an information flow policy, specified by assigning to data security levels and specifying the levels between which flows are permitted, is respected by a program, in the sense that all information flows that will be observed by the attacker are permitted. To make the definition more precise, it is necessary to determine the observational capabilities of an attacker by identifying which information he shall gather during the execution of a program. If we formalize program executions as possibly infinite sequences of states, some possible models of attackers are summarised in the following:

- The final values of all low variables after a run are observable but the values in intermediate states are not observable.
- All changes to values of low variables during a run and the order of these changes are observable.
- The values of all low variables *in all states* and the ordering of these changes is observable. This includes the possibility of observing stuttering.
- The values of all low variables, all changes of their values, and the precise time of each such change are observable.

Attackers shall be able to exploit different channels according to their capabilities. The examples below review some common channels that may leak secrets to attackers. For the sake of readability, examples are stated in a simple, high-level, concurrent language. Furthermore, we use a two-level security policy is used where variables that might contain confidential values are typed high and observable variables are typed low. The goal of information-flow control is to avoid initial values of high-typed variables to interfere with values of low-typed variables. The typing is indicated by the name of the variables (h:high, l:low).

explicit channels Secret values are published directly by printing them or making them otherwise visible, for instance by copying them into publicly observable variables like in

```
l:=h.                                                          (4.1)
```

implicit channels Secret values are published by indirectly influencing the low-observable behaviour of the program via control structures. The program

```
if h=0 then l:=0 else l:=1                                     (4.2)
```

copies the value of h indirectly into l. Explicit and implicit information channels [18] are often classified as *storage* channels. Barthe and Rezk [9] have identified further implicit channels that arise in the context of stack-based languages such as the JVM.

timing channels The program

```
if h>0 then l:=0 else { h:=h+1; l:=0 }                         (4.3)
```

is an example where measuring the run-time uncovers more information than permissible. While the computation of the first branch will take one time slice the computation of the second branch needs two time slices.

termination channels Similar to timing, the termination behaviour of a program might reveal secret values as the following program demonstrates:

```
while h>0 do skip                                              (4.4)
```

If h is initially less or equal to 0 the loop will terminate and, otherwise, it will not terminate. Here, we view termination channels as a special form of timing channels.

scheduler-specific channels These channels leak secret information if the low observable behaviour under a specific scheduler depends on high variables. This is the case for the following two programs:

While the example program

```
if (h>0) then
 fork (l:=0 | l:=1)
else                                                           (4.5)
 fork (l:=1 | l:=0)
```

from [55] is secure in the presence of a uniform scheduler that re-schedules after each atomic computation step, it is insecure under a round-robin scheduler that grants each thread queued in a FIFO-list a time slice to compute one single step. In contrast to this, the following program can be exploited under a uniform scheduler but is secure under a round-robin scheduler.

```
if (h=0) then
 fork (l:=1; l:=2 | l:=0)
else                                                           (4.6)
 fork (l:=1 | l:=0; l:=2)
```

To capture the attacker capabilities, we must therefore resort to an appropriate semantics of program executions. Following the philosophy of *extensional security* [50] and a long line of work in information flow security, the security policy is defined in terms of standard semantics as opposed to security-instrumented semantics. One such semantics has been defined formally for a sequential fragment of the Java Virtual

Machine, and is currently being extended to multi-threaded Java; it could serve as a starting point to define non-interference. However, it might be necessary to extend the semantics so that it captures some observational capability of the attacker, e.g. with respect to timing.

There are however a number of difficulties in formulating an appropriate notion of information flow security for the Java Virtual Machine; more precisely, the main difficulties are: multi-threading and distribution, object-orientation, and lack of structure of bytecode. Whereas each difficulty has been studied in isolation by consortium members, one goal of Mobius is to understand these issues in combination, and drawing upon previous work to provide permissive but sound policies for JVM programs. These policies shall be formalized in Deliverable D2.1, where we shall also present type-base mechanisms for enforcing these policies.

4.8.2 Requirements on enforcing non-interference

Devising enforcement methods for information-flow policies is the objective of subsequent tasks. The purpose of this section is to specify requirements on the methods to be developed, so that they integrate smoothly in the security architecture that shall be proposed within the project and in the development cycle of global computing applications.

One goal of the project is to provide a tight integration between two enabling technologies that can be used to analyse software, namely type systems and program logics. For the purpose of analysing requirements on enforcement methods, we shall however consider both technologies in isolation.

Requirements on methods to enforce non-interference with type systems

The requirements for a type-based information flow analysis at bytecode level are three-fold. Firstly, the analysis should be compatible with bytecode verification, in order to integrate smoothly with existing Java security enforcement mechanisms. Secondly, the information-flow analysis should be combined with other static analyses such as exception analysis or pointer analysis in order to retain some acceptable level of precision. Thirdly, the analysis should be coherent with existing tools to develop information-flow secure Java applications.

Compatibility with bytecode verification In order to precise the requirements on type systems for information flow, we briefly review the principles of the standard Java bytecode verifier.

Bytecode verification is a key security function of the Java architecture. Its purpose is to check that applets are correctly formed and correctly typed, and that they do not attempt to perform malicious operations during their execution. It consists on a two steps process. The first one, and the simplest, is a structural analysis of the consistency of the class file and its constant pool. The second one requires a static analysis of the program and is meant to ensure some basic properties, for example that: values are used with their correct type (to avoid forged pointers) and method signatures are respected; no frame stack or operand stack underflow or overflow will occur; visibility of methods (private, public, or protected) is compatible with their use; jumps in the program code remain in legal bounds. Ensuring such properties is an important step towards guaranteeing security, and the failure to enforce any of these properties may be exploited for launching attacks.

Bytecode verification [48] is a data-flow analysis of a typed virtual machine which operates on the same principles that the standard JVM except for two crucial differences: the typed virtual machine manipulates types instead of values, and executes one method at the time. The data-flow analysis aims at computing solutions of data-flow equations over a lattice derived from the subtyping relation between JVM types, and uses to this end a generic algorithm due to Kildall [44]. In a nutshell, the algorithm manipulates so-called stackmaps that store for each program point a history structure that represents the program states that have been previously reached at this program point; different history structures can be used depending on the accuracy required from the analysis. The history structure is initialised to the initial state of the method being verified for the first program point, and to a default state for the other program points. One step of execution proceeds by iterating the execution function of the virtual machine over the states of the history

structure. Each non-default state is chosen once and the result of the execution of the typed virtual machine on this state is propagated to its possible successors in the history structure.

Bytecode verification computes the fixpoint of the dataflow equations that are generated from the program using the abstract transition relation. However, it is also possible to avoid computing fixpoints, which is expensive. One solution adopted in the KVM [15] is to rely on lightweight bytecode verification, initially proposed by Rose, in which the program comes equipped with the solution to the dataflow equations; by analogy with Proof Carrying Code, the solution is called the certificate. Then, the role of the lightweight bytecode verifier is to check that the certificate establishes the correctness of the program to which it is attached. Lightweight bytecode verification is sound and complete w.r.t. bytecode verification, in the sense that every program for which a certificate exists will pass bytecode verification, and conversely one can provide a certificate for every program that passes bytecode verification (the certificate is the result of the fixpoint computation). It has been announced by Sun that lightweight bytecode verification will be generalised to all Java technologies in the future.

We are now in a position to formulate our requirements for type-based enforcement of secure information flow:

- verification should be performed method by method and through the generic data flow analysis of Kildall. W.r.t. the second point, it entails in particular it should not commit to a particular history structure; w.r.t. the first point, it entails that method signatures should be extended with appropriate security information that shall be used by the abstraction transition relation to simulate method call (and return);

- it should be amenable to efficient checking through the establishment of lightweight verification procedures. For example, secure information flow verification for bytecode are likely to use *security environments* that map program points to security levels. It shall be important to develop both standard verifiers that compute these security environments as fixpoints, and lightweight verifiers that, for efficiency purposes, merely check the correctness of security environments, which are provided as additional information to applications;

- it should rely on other analyses that provide additional information about the programs to be verified. For example, previous work on secure information flow for fragments of the JVM rely on control dependence regions to avoid implicit flows, such as assignments to low memories in branches of the program that depend on high values. Tracking (implicit) information flows requires some knowledge of the control structure of the program; this knowledge is embodied in the notion of *control dependence regions*. These regions can be approximated safely [9, 54], but in order to be approximated precisely, it is important to develop new control dependence regions analyses that exploit the results of other analyses, e.g. exception analyses. One requirement is thus to develop a lightweight checker for precise control dependence regions.

In addition, it is important to guarantee the compatibility of our analysis with JFlow, an extension of Java with a flexible and expressive information flow type system [51]. Because of the flexibility of its type system, especially with respect to declassification, it has been difficult to characterise formally the security properties that are verified by typable JFlow programs, and not all typable JFlow programs shall be compiled into programs that are accepted by our analysis. However, Banerjee and Naumann [5] have devised for a Java-like language in the spirit of Jif isolated a type system that enforces *non-interference*. While their analysis does not directly address mobile code security (it applies to source code, whereas Java applets are deployed in compiled form as JVM bytecode programs), it is important to ensure compatibility between the analyser we shall develop and the analysis of [5]. Indeed, JFlow offers a practical tool for developing secure applications, and in particular for ensuring to developers that applications meet high-level policies about API usage. In contrast, the information flow type system we propose to develop shall augment the Java security architecture to provide assurance to users that applets respect high-level policies about API usage. The value of these two lines of work can be greatly increased by connecting them via a type

preservation result, showing that programs typable in a suitable fragment of JFlow will be compiled into bytecode programs that pass information-flow bytecode verification. The interest of such a result is to show on the one hand that applications written with JFlow can be deployed in a mobile code architecture that delivers the promises of JFlow in terms of confidentiality, and on the other hand that the enhanced security architecture we shall develop can benefit from practical tools for developing applications that meets the policy it enforces. Preliminary results in this direction can be found in [7].

Requirements on enforcing secure information flow with logic

Previous work by members of the consortium have shown the benefits of using logical characterisations for simple imperative languages, and for fragments of the JVM [6, 17, 20]. The approach has been extended by Naumann to handle a more realistic policy for a larger fragment of the JVM. The goal of Task 3.2 is to extend these works to more substantial fragments of the JVM. Ideally, the logical characterisation of non-interference should be expressible in the bytecode logic that shall be developed in Tasks 3.1 and 3.2, and that shall be supported by the Mobius IVE to be developed in Task 3.6.

Aiken and Terauchi [58] have built upon previous work by consortium members and proposed a type-based transformation technique that minimises the overhead of code duplication incurred by the self-composition approach. It could be of interest to extend their transformation to the JVM, and integrate it in the Mobius IVE.

4.8.3 The need for more permissive information flow policies

As suggested by the scenarios discussed earlier, an important requirement on policies for information flow controls is to address intentional information release, or *declassification*. For example, once an applet encrypts a secret credit card number with sufficiently strong encryption and a secret key, it is safe to declassify the result of encryption as public and send it over an untrusted medium such as the Internet. Although such an applet is intuitively secure, it is rejected by non-interference. The reason is that variation in the credit card number and secret key cause variation in the result of encryption.

A challenge for declassification is to relax non-interference to more permissive policies with the important requirement that the attacker should not be able to launder information through declassification mechanisms. For example, program that encrypts with some secret key `k` sensitive data stored in variable `secret`

$$\texttt{l := declassify(encrypt}_k\texttt{(secret))} \quad (4.7)$$

should be accepted as secure. On the other hand, program

$$\texttt{if h>0 then l := declassify(encrypt}_k\texttt{(secret))} \quad (4.8)$$

should be rejected as to disallow laundering information about secret `h` through the declassification mechanism for results of encryption.

Inspired by ongoing work by consortium members [56], we identify the following *principles* as desirable for declassification policies.

- *Semantic consistency* The semantic consistency principle suggests that security definitions should be invariant under equivalence-preserving transformations. Definitions that satisfy this requirement offer high modularity. The principle is compatible with the philosophy of extensional security where the behaviour (semantics) of the system fully determines the security of the system.

- *Conservativity* The conservativity principle states compatibility with non-interference. Non-interference is a common baseline policy for programs without declassification. This principle states that the definition of security should be a weakening of noninterference: when a program contains no declassification then the security of the program degenerates to non-interference. For example, forgetting to encrypt the secret in Example 4.7 would result in program `l := secret`. The conservativity principles implies that this program should be rejected because it violates the noninterference property.

- *Monotonicity of release* The more data is declassified the weaker the policy is. This principle requires that that adding declassification annotations to code cannot make a secure program become insecure. Conversely, removing declassification from an insecure program should not make it secure.

- *Non-occlusion* This informal principle requires that the presence of declassifications cannot mask other covert information leaks. This principle is illustrated in Example 4.8 where declassification is intended for `encrypt`$_k$`(secret)` but can be used to reveal information about `h`.

Different kinds of declassification are needed for different reasons. For example, declassification of encrypted values is different from declassification in an information purchase protocol. In the latter, sensitive information is entirely released but only under the condition that payment has been transferred. An important aspect of this security policy is *when* information is released. We shall investigate dimensions of declassification according to *who* releases *what* information, *when* and *where* in the system. It is the goal of Task 2.2 to develop a general declassification framework and to apply it to bytecode languages.

Chapter 5

Resources

In this chapter we address the requirements of security policies that deal specifically with resources, building upon the general background of Chapters 2 and 3. Resource management is obviously of general importance on mobile devices, where computing power is tightly constrained; but it also has a particular impact on security. First, some platform-specific resources are intrinsically valuable — for example, because an operator will charge money for them (phone calls, text messages, internet packets) — and so we want to guard against their loss. Further, overuse of limited resources on the device itself may quite easily compromise *availability*, leading to vulnerability in denial of service.

The chapter outline follows the detailed implementation plan given for Task 1.2 in the original MOBIUS project description. We begin with an analysis of possible notions of "resource" (Section 5.1), addressing both MIDP and other Java platforms. This covers a variety of candidate resources, investigating their importance for particular devices and how they might be handled by formal analysis.

Section 5.2 presents two specific resource management scenarios for a MIDP device, based on "block booking" of text messages. This is a convenient technique for ensuring that transactions have sufficient resources to run to completion; but currently it cannot be applied in MIDP because of security concerns. We illustrate its use in realistic applications, and explore how formal resource policies could provide the necessary security guarantees.

We conclude with an analysis of formal methods for resource verification (Section 5.3), and in particular their potential application to proof-carrying code for Java. Building on previous research by consortium partners, we are able to set out requirements for the MOBIUS resource security platform.

5.1 Candidate Resources

In this section we will consider some quantities which can usefully be regarded as "resources" and for which it would be useful to have static bounds in advance of program execution.

5.1.1 Java and resources

We will deal with resources both in general and in the specific setting of the MIDP profile. MIDP has been described in some detail earlier (see Chapter 2), but there are other Java environments for platforms with resource properties which differ significantly from MIDP:

JavaCard. JavaCard is a Java environment for smartcards, where resources are extremely limited. A smartcard generally has an 8-bit processor with no floating-point operations; only a few kilobytes of RAM will be available, but there may be persistent storage available in the form of a few tens of kilobytes of flash memory. The JavaCard platform is a Java implementation specifically designed for use on smartcards. The JavaCard API is very small and is specialised for smartcard applications. JavaCard applications are written in standard Java, but converted into specialised JavaCard bytecode prior to installation on card. The

JavaCard JVM (JCVM) is significantly different from the standard JVM: for example, the only primitive types available are `boolean`, `byte`, `short` and (possibly, but not necessarily) `int`.

J2SE and J2EE. These are Java environments for devices with considerable resources available. J2SE is the Java 2 Standard Edition, as found on desktop machines and servers; J2EE is the Java 2 Enterprise Edition, used for the implementation of powerful commercial applications involving Web services and related technologies. In both of these environments, applications will generally be running on machines with fast CPUs and generous amounts of storage.

5.1.2 Specific resources

It should be apparent that the resources which we wish to consider may vary depending on the area of application. For smartcards it is important that we have good control over memory usage and execution time (we may also need a more refined notion of memory since much more time is required to write data to flash memory than to RAM). For cellphone applications it may be more important that we be able to quantify API calls, such as requests for sending SMS messages. For Java applications running on desktop machines we may again be concerned with execution time and CPU usage, particularly in the case of mobile code, where denial of service is an issue.

Nonetheless, there are certain issues which are important in all situations, and we believe that a good understanding of these issues will be helpful for making predictions of resource usage on different platforms. The issues which we will consider are as follows:

- CPU usage and execution time.
- Memory usage, perhaps with some refinements.
- Limits on numbers of calls to particular API methods.
- Bounds on parameter values for method calls.
- Cumulative bounds on parameter values.
- Numbers of threads

CPU usage

It may be difficult to obtain precise bounds on CPU usage for a given program. The actual time required to execute a given program will depend strongly on the available hardware and the JVM implementation in use (indeed, a bytecode program may even be compiled to native code and run without the use of a virtual machine). However, it seems likely that one could obtain good bounds on the number of bytecode instructions executed by a given program, and this will give information about time complexity which is likely to be of value irrespective of the particular means used to execute the program. Such information would also be of use to programmers, highlighting methods which are particularly resource intensive and which would benefit from optimisation. Information on time complexity would also be of use in program design.

- In JavaCard, computations must happen very quickly. Typically, a card will be inserted in a terminal (perhaps in a shop, or in a public transport system) and some transaction will be performed. It is clearly important that this should happen quickly: for example if passengers are using smartcards to pass through turnstiles in a railway station then authorisation must occur effectively instantaneously. The benefits of static bounds on execution time are clear.

- We have seen above (section 3.3.3) that in MIDP applications it is important not to block the GUI by performing time-intensive operations in the main thread of a MIDlet: static information about time requirements would be helpful in deciding which tasks should be given their own threads (especially bearing in mind that there may be limits on the total number of threads which can be created).
- In desktop and enterprise systems there will generally be a lot of CPU power available. However, execution time is still an issue. For example, static bounds on execution time (accompanied by a PCC certificate) would be helpful in preventing denial of service attacks. Similarly, such information would be of great use in Grid applications where users download programs to remote high-performance computers; the services provided by such machines are scarce and expensive and it would be useful to know in advance that a user's program is well-behaved.

Memory usage

It would be useful to have information about the memory usage of bytecode programs. On the simplest level this might involve static bounds on heap-space usage; the benefits of such knowledge are similar to those described above for CPU usage. Some refinements might be required in specific situations. For example, a smartcard has several different types of memory (RAM, flash) with different resource-usage characteristics and we would need to be able to treat objects in different types of memory separately. The situation in MIDP is similar: objects in main memory should be treated separately from objects in an RMS record store or objects containing video or audio information which may be stored in dedicated memory.

Note also that some handsets have *real memory leaks because of bogus or limited implementations*. Then it is important to carefully check how some operations (when, absolute bounds, etc) are done (for example image loading).

Because of hardware limitations, some handsets cannot allocate objects (arrays) bigger than a rather small fixed limit.

Resources and API calls

The usage of many candidate resources (for example, files, GUI windows, network connections) is mediated by calls to methods in the Java API, and hence we wish to consider static analyses of such method calls. In the simplest situations, we may only require knowledge of the number of times a particular method is called; in more complex situations we may wish to have bounds on the values of method arguments, or to know about the cumulative value or the arguments with which a method is called. As an example, we will consider the question of network connections in MIDP in some detail.

Billable events: SMS messages. It is possible to send short messages with MIDP phones (SMS). The API is defined in the WMA specification [24, 30].

The following methods are used for sending an SMS (text in square brackets represents optional parameters, package names are not given: either `javax.microedition.io` or `javax.wireless.messaging`):

`Connector.open("sms:// ...").` This creates an instance of a MessageConnection object whose default destination address is the one given in the URL. If no URL is given, the object can be used for receiving messages.

`Message MessageConnection.newMessage(String type[, String address]):` this creates a new message object. A message object is defined by its type (either binary or text), its address (destination phone number) and its payload (contents). By default the address is inherited from the MessageConnection object.

`MessageConnection.send(Message m):` this method sends a message.

`Message.setAddress(String text):` changes the destination address of a message to the one given as argument.

`TextMessage.setPayloadText(String data):` this sets the contents of a message if it is pure text.

`BinaryMessage.setPayloadBinary(byte [] data):` sets the contents of a binary SMS.

Because each SMS message costs the customer money, authorisation is required: before an SMS message is sent, a confirmation screen warns the user and asks for authorisation to perform the sending. This dynamic authorisation policy has some drawbacks:

- The user does not know how many SMS messages will be required to complete a task (eg. sending a form representing a registration),
- When the user has begun a complex task, he may be reluctant to stop it but be angry because of the price of the communications involved.
- The user does not necessarily know the price of the SMS.

A better solution is to count in advance the number of SMS messages which will be sent. Note that most MIDP applications never terminate. This means that clear interaction points must be isolated and communication events counted between them.

There are some other issues in connection with SMS messages:

- The cost of an SMS depends on the number called. In some countries (France), but not all (UK), it is easy to compute the cost from the phone number.
- A message which is too long is cut into pieces (at most 3). Each chunk is charged as a regular SMS.

This means that it is also important to know the phone number used and the size of the message.

Remark: a telephone number is never computed. It is either a constant written in the code or a value retrieved directly from the environment (from the network, from a property file or from the user interface). On the other hand the URL used to open the connection is often computed as the simple concatenation of a URI schema (`"sms://"` here) and a telephone number eventually followed by a port number.

Billable events: HTTP connections. In contrast to SMS messages, which are discrete events, the use of HTTP connections (and more generally, other kinds of IP connections when they are available) is charged on a volume basis. Both outgoing and *incoming* traffic are charged. One may want to know an upper approximation of the behaviour of a midlet.

Usually only the traffic sent can be approximated; however, the traffic received is often more important (images, multimedia contents, etc.). One could always count the number of requests knowing that the cost depends on the size of the answer.

The following methods are used to create HTTP connections and transmit and receive data:

`Connector.open("scheme:// ...").` Here `scheme` is either `"http"` or `"https"` (it is also possible to use raw sockets on some networks). The result is either an `javax.microedition.HttpConnection` object or an `javax.microedition.HttpsConnection` object.

`InputStream InputConnection.openInputStream():` creates an actual input stream to receive data. A variant is `InputStream InputConnection.openDataInputStream()`.

`OutputStream OutputConnection.openOutputStream():` this creates an actual output stream to send data. A variant is `OutputStream OutputConnection.openDataOutputStream()`. There are direct methods in the `javax.microedition.io.Connector` class: `InputStream openInputStream(String url)` and `DataInputStream openDataInputStream(String url)`.

`OutputStream.write(byte[]), OutputStream.write(byte[],int o,int l), OutputStream.write(int b):` these are raw functions to send data; DataOutputStream contains more functions. There are also direct methods in the `javax.microedition.io.Connector` class: `OutputStream openOutputStream(String url)` and `DataOutputStream openDataOutputStream(String url)`.

Note 1: The difference between methods in `Connector` and methods in `HttpConnection` is that the later class gives also access to the header fields. The `HttpsConnection` class gives also access to security parameters stored in a `SecurityInfo` object.

Note 2: there are other ways to open connections using higher level APIs. For example, a multimedia player can be created with the name of the source from which to fetch information: the static method `javax.microedition.media.Manager.createPlayer(String url)` returns an object of type `javax.microedition.media.Player`

API calls. From the discussion of MIDP methods above we see that different levels of detail may be required in the analysis of method calls. In the case of SMS messages, the basic information required is the number of calls to `Connector.open`; however, this quantity in itself is not sufficient to calculate the overall cost of a transaction. Firstly, only calls to `Connector.open` with a parameter beginning `"sms://"` are relevant. Secondly, the cost of an SMS message depends on the telephone number dialled (and the cost may depend on the number in some complicated way). Thus we need knowledge not only of the number of calls to `Connector.open`, but also of the arguments involved.

The situation is somewhat different in the case of HTTP messages. Here, the basic connection is obtained from a call to `Connector.open` with an argument `"http://..."`; if we wish to output some data then the `Connector` object must then be converted to an `OutputStream` object (`os`, say) by a call to `openOutputStream`, and then data is finally transmitted by calls to the instance method `os.write`. The total cost will depend on the cumulative value of the arguments to `OutputStream.write`, but we also need to keep track of the fact that the stream `os` has been obtained from an object which was created by calling `Connector.open("http://...")`.

We propose that in order to analyse resources which are allocated and consumed by calls to methods in the Java API we should investigate static analyses for the following three quantities:

- Limits on numbers of calls to particular API methods.
- Bounds on parameter values for method calls.
- Cumulative bounds on parameter values.

Static bounds on these quantities would be useful for many resources in addition to the ones considered above. There is a virtually unlimited number of possibilities, but here are a few examples.

- The number of files open at a particular time can be found by counting calls to appropriate methods.
- Similarly, one could bound the number of windows opened by an application.
- Graphics objects should only be drawn if they lie entirely within a specific window; this could be guaranteed by static prediction of bounds for arguments to graphics calls.

- Array accesses should only occur within the array bounds (this is dynamically checked in Java, but a static guarantee would rule out the run-time exceptions which occur when array bounds are violated). Array access does not take place via method calls, but rather via special bytecode instructions; however, the problem is very similar to that of checking bounds for method arguments.

- Usage of the MIDP Record Management System. RMS is a small database API for midlet persistent data, and it would be very useful to evaluate the use of RMS by a midlet: firstly, it is a shared resource and saturation may lead to a denial of service for others or may block the midlet; also access to RMS may be very slow on some handsets. All access RMS record stores is via calls to methods in `javax.microedition.rms`, and knowledge of the number of calls to these methods and the size of the method arguments would be helpful in evaluating the overall RMS usage of a MIDlet.

Of course, the general problem of determining bounds on method arguments is undecidable. However, in practice the arguments to resource-related API calls are often calculated in a straightforward way, and static analysis may be quite feasible. (Consider the case of sending an SMS message: the argument to `Connector.open` is a string which in principle could be obtained by an arbitrary computation; in practice, the argument will probably be a literal constant in the program, or will be obtained by concatenating the literal string `"sms://"` and a telephone number).

Number of threads. As explained in section 2.5.3, control of threads is particularly important in MIDP applications. Understanding how many threads are used and when could be useful to enhance information flow analysis: during the most critical parts of a midlet, only one thread should be active. Moreover, it is a bad programming practice to have a potentially unlimited number of threads. JTWI recommends to limit the number of active threads to 10. Good thread usage examples along those lines are given in [46].

Note that in MIDP programs all threads are explicitly created by the programmer, by creating objects from classes which extend the `Thread` class or implement the `Runnable` interface. Thus to bound the number of active threads it would suffice to bound the number of objects from these classes which are created; however, this strategy is complicated by the fact that threads are never explicitly destroyed, but are simply reclaimed by the JVM when they terminate.

In the case of general Java programs, it might also be useful to have information on thread usage (particularly of user-generated threads), especially in the context of denial-of-service attacks. However, counting threads may be substantially more difficult for non-MIDP applications because threads are often created automatically by objects created by API calls: for example, a Swing GUI will create its own threads in order to handle events efficiently.

5.1.3 Some other issues

There are a few other resource-related issues which might be considered in the context of MIDP applications.

Power consumption. This issue is almost intractable. Power consumption is really implementation specific. If one forgets about video (mobile 3D), the consumption of a device may depend on the use of its memory banks, internal caches, locality of addresses, etc. As it seems very hard to model such behaviour, it will be harder to evaluate applications to find an approximation of their consumption.

Screen size. There are too many different screen sizes. An application should be specialised to accommodate a range of screen sizes and centre its logical screens on the available physical screen. Analysis could be used to check that all coordinates are relative to the centre of the physical screen and within bounds.

Concurrent accesses to shared resources. The number of concurrent accesses to resources such as network connections, RMS database is limited on many handsets.

5.2 Resource scenarios: Block booking text messages

Frequently, applications require to send multiple text messages in a short time, for example to notify a group of addressees, or to implement a protocol that requires the exchange of multiple messages. The MIDP security policy, as defined in the annex "The Recommended Security Policy for GSM/UMTS Compliant Devices" to the MIDP specification [23], requires that each event of sending a text message be individually authorised by the user clicking away a pop-up screen.[1] However, these pop-up screens defeat user-friendliness, and also enable social engineering attacks based on user distraction, as described in Section 3.3.1. A user-friendly application should "book" all the required text messages "en bloc" in advance (including getting the user's authorisation) and then send the booked messages quietly without pestering the user. As a scenario for resource management we describe transactional applications which use such block booking of text messages. In general, such applications perform the following steps.

1. Start transaction by booking N text messages. This step includes getting the user's authorisation to send the messages.

2. During transaction, send up to N messages without asking for further authorisation.

3. End transaction. This step includes voiding the authorisation to send text messages in case less than N messages have been sent so far.

Accordingly, the challenge in this scenario is to provide a proof that (i) during the transaction no more than N messages will be sent, (ii) N messages will suffice to complete the transaction, and (iii) no messages will be sent (without further authorisation) before the start or after the end of the transaction.

In the following, we present two application scenarios which make use of block booking.

5.2.1 Application scenario I: Hotel reservation

The first application scenario forms part of the itinerary planner scenario from Section 4.5; it implements the room reservation protocol (Step 4 of the itinerary planner) in another way. Figure 5.1, which is meant to replace the lower part of Figure 4.1, shows the threads, servers and messages involved in the protocol.

At the beginning of the scenario, we assume that the hotel selection part of the itinerary planner is completed and has provided the MIDlet with information how to book a room, in particular with the phone numbers for sending the text messages containing the reservation request and the payment information. The scenario comprises the following steps.

1. The user interface asks for the credit card number.

2. The user interface pops up a screen asking to authorise the sending of two text messages: one to the hotel and one to the bank.

3. If the user clicks OK then the reservation and payment transaction starts. A thread is spawned off for handling the actual transaction. Meanwhile, the user interface displays an info screen (e.g. showing a 30 second timeout count-down).

4. The payment thread sends a text message to the hotel.

5. The hotel generates a ticket for the transaction, which it sends first to the bank using any protocol, thus notifying the bank of the expected payment. Then the hotel sends a text message containing the same ticket back to the MIDlet.

6. The payment thread sends a text message containing the credit card number and the ticket to the bank.

[1] Some countries even enforce such one-shot authorisation by a legal requirement demanding that the user is informed about the cost of the application at any time and before that cost is billed to him.

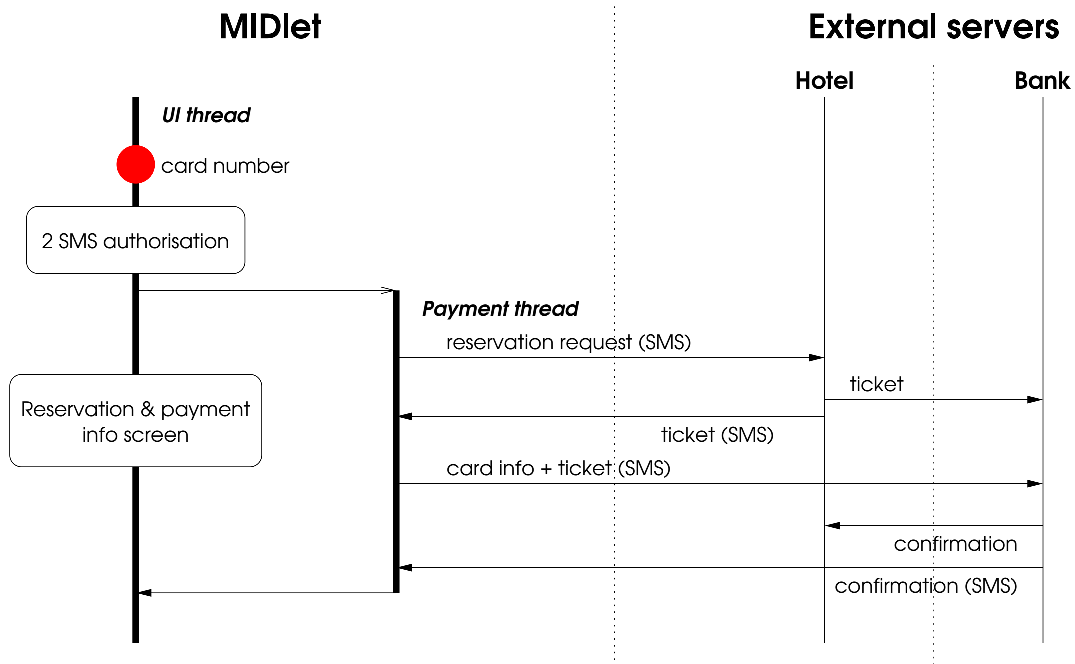

Figure 5.1: Hotel reservation scenario involving block booking of 2 text messages.

7. The bank processes the credit card data and eventually sends two confirmation messages, one to the hotel using any protocol and one to the MIDlet using a text message.[2]

8. The payment thread terminates after receipt of the confirmation message, which completes the reservation and payment transaction.

In this simple protocol, the application block books two text messages, which it sends without user interaction during the transaction. Besides being more transparent and user-friendly than a solution which requires authorisation of each message individually, block booking also increases the atomicity of the transaction, since the user cannot abort it by granting the first message but denying the second. Moreover, with block booking the transaction is likely to complete within few seconds, as the dominating delays are likely due to network latencies and the time that the bank requires to process the credit card data. Without block booking, however, the dominating delay is most likely caused by the MIDlet waiting for the user to authorise a message, resulting in the transaction probably taking more than 30 seconds.

5.2.2 Application scenario II: Group messaging

The second application scenario is about bulk messaging, i.e. sending a message to a group of recipients. Figure 5.2 shows the abstract workflow of a group messaging MIDlet, which proceeds in the following steps.

1. The user types a message template using a text editor. The template may contain placeholders for name, title, et cetera, of the recipients.

2. The user selects a group of recipients from his phone book.

3. The MIDlet computes the total number N of text message to be sent, where N depends on the number of recipients and on the size of the message template.

[2]The bank communicating with the MIDlet via text messages (rather than an IP protocol) provides an extra means of authentication: The bank may not accept the payment request unless the phone number of the sender matches the phone number of the card holder.

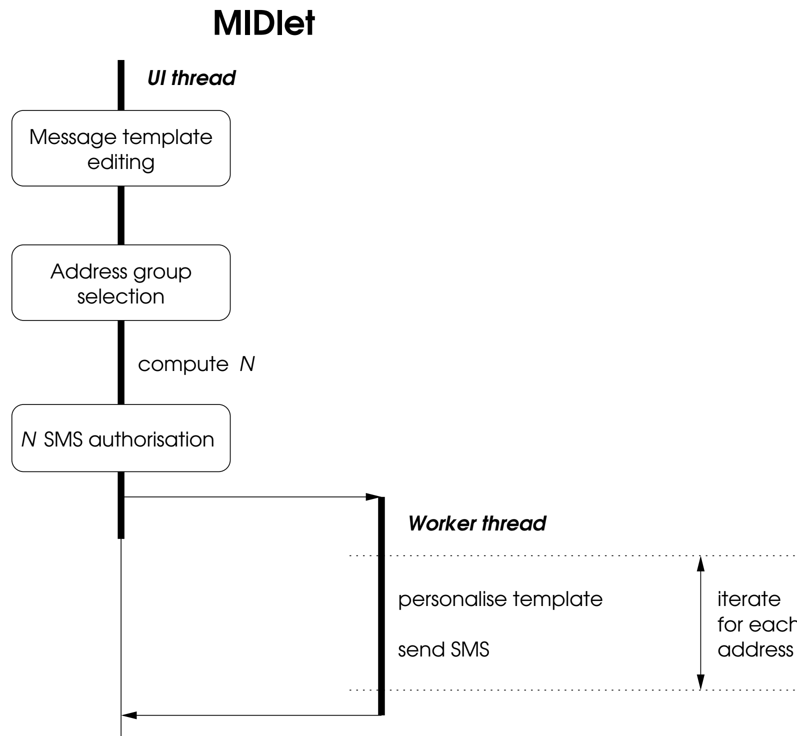

Figure 5.2: Group messaging scenario involving block booking of N text messages.

4. The user interface pops up a screen asking to authorise the sending of N text messages
5. If the user confirms then the message send transaction starts. A worker thread is spawned off to handle the actual transaction. The user interface thread may already terminate at this point; it need not wait for the worker thread.
6. The worker thread iterates through a loop, for each of the recipients in the group performing the following steps.
   (a) Personalise the message template, filling the placeholders with the recipient's entries from the phone book.
   (b) Transmit the personalised message by sending one or more text messages (depending on the size of the personalised message) to the recipient's phone number.
7. The worker thread terminates, which completes the message send transaction.

In this scenario, the application books a block of N text messages, which it then sends one after the other without user interaction. Again, block booking is more user-friendly (less authorisation clicks) and transparent (authorisation informs about total cost of transaction) than one-shot authorisation. Besides, it prevents social engineering attacks based on user distraction, in particular if the message is to be sent to a large group of recipients. Note that the number of booked text messages is not an a priori known constant (as in the previous scenario) but is computed from user input.

5.2.3 Desired resource properties

In the above scenarios, we are interested whether the "block booked" resources (i.e. the text messages) are used correctly. Spelt out in detail for the hotel reservation scenario, we want the following four properties to hold.

- No text message is sent prior to authorisation.
- In between authorisation and the end of the payment thread at most two text messages are sent, one to the hotel and one to the bank.
- No text message is sent after the termination of the payment thread.
- The payment thread does eventually terminate, provided that the environment (i.e. the hotel, the bank and the network) is cooperative (i.e. it produces the messages the thread is waiting for).

The first three of the above properties are resource invariants that must hold when the application attempts to send a text message before, during or after the transaction, respectively. In contrast, the last property expresses conditional termination (dependent on the behaviour of the environment). Though termination is not a resource property, in most cases it can be strengthened into one. In the hotel reservation scenario, for instance, we might instead demand that the number of instructions executed by the payment thread is uniformly bounded. This is a resource invariant, which obviously implies termination.

For the group messaging scenario there is a similar set of resource properties, again split into invariants and termination properties. Since the main concern in the above scenarios is with the cost of billable events, other resources like memory usage or number of threads are not of interest here.

Note that a specification of correct resource usage tends to be far less complex than a full specification of a MIDlet's functional and behavioural properties. For example, the resource properties for the hotel reservation scenario only constrain the number of messages sent but do not specify their order or relate their contents.

5.2.4 Implementing block booking in MIDP

Block booking of text messages cannot be implemented in the current version of the MIDP standard because the MIDP security policy (see annex to the MIDP specification [23]) requires one-shot messaging authorisation for all applications, regardless of whether they are unsigned or signed by a trusted third party.[3] As a consequence, the MIDP method call for sending a text message always pops up an authorisation screen, and there is no way of convincing the framework that authorisation has been obtained earlier.

The demand for block booking resources has been recognised by manufacturers and operators involved in the MIDP standardisation process. FT plans to extend MIDP with explicit "permission objects" which count the resources (in particular, specific method calls) that a piece of code is allowed to consume. We illustrate the use of permission objects on the above hotel reservation scenario.

1. To start, the user interface thread creates an object `perm` of class `Permission` and initialises it with the capability to send two text messages, one to the hotel and one to the bank.
2. The user interface asks for the credit card number.
3. The user interface thread calls `perm.enable()` to enable the capabilities of `perm`. This call pops up an authorisation screen so that the user can grant or deny the capabilities; denial will result in an exception being thrown (which the application should catch).
4. The user interface forks off the payment thread, to which it passes the relevant data (phone and credit card numbers) and the object `perm`.
5. The payment thread uses method calls `Connector.open(perm, "sms://...", WRITE)` to send the two text messages. These calls check whether the requested capability is present in the object `perm` and throw a `SecurityException` if it is not. Then they consume (i.e. decrement) the capability before finally opening the connection.

[3] Applications signed by the manufacturer or operator need not suffer from the one-shot authorisation restriction. However, the MIDP security policy *recommends* that they still follow the same one-shot messaging authorisation scheme.

6. Before terminating, the payment thread destroys the object `perm` in order to void any unused capabilities.

The `Permission` class is designed in such a way that an application can initialise instances to hold arbitrary capabilities for accessing resources (not just text messages), but, after enabling these capabilities can only be consumed by the respective MIDP method calls. Thus, objects of class `Permission` act like explicit resource counters, and the object encapsulation ensures that the application code cannot tamper with these counters.

Besides being resource counters, permission objects naturally introduce a notion of transaction into MIDP. A transaction corresponds to the life time of an enabled permission object, i.e. it starts when the `enable()` method call terminates successfully, and it is completed when the permission object is destroyed. By this fact, we can formulate the following framework-specific resource properties for transactions (using a permission object `perm`).

No unauthorised use. Whenever resources are used, `perm` is enabled.

No overuse. Whenever resources are used, `perm` holds enough capabilities.

Completion. Eventually, `perm` is destroyed.

5.2.5 Enforcing resource policies

Simple resource policies such as "an application must not send more text messages than it was authorised to" can already be enforced using the above permission objects. As the capabilities are checked at runtime before the requested resource is used, any unauthorised attempt will result in an exception. However, this raises the question what to do with the exception. Consider the hotel reservation scenario and assume that the payment thread attempts to send a second text message to the hotel before sending one to the bank. Since the capability to send a message to the hotel is already gone the system will throw a `SecurityException`. The application may catch this exception, but what should it do now? The options are

- to continue the transaction although the cost is higher than expected, or
- to abort the transaction although some cost has already accrued.

The user may be asked to take this decision but she may find both options unsatisfactory.

While runtime checks can enforce that an application does not use more resources than it was granted, they are not sufficient to enforce resource policies in the way that users expect, i.e. to ensure that an application does not even *attempt* to use more resources than it was granted. Therefore, the best way to enforce resource policies is to prove the desired resource properties (by static analysis, type inference, or logical deduction) from the application's bytecode. For example, to prove that the hotel reservation application meets the first three resource properties in Section 5.2.3, we need to statically check on the bytecode that

- only the payment thread calls the method `Connector.open(perm, "sms://...", WRITE)`, and
- whenever `Connector.open(perm, "sms://...", WRITE)` is called then the `perm` object is enabled and has the capability to send a text message (i.e. the resource counter for sending text messages is non-zero).

Likewise, to prove the fourth property, we need to do static analysis or type inference to establish an upper bound on the number of bytecode instructions that the payment thread requires to execute.

The evidence (which may take the form of proof trees or typing derivations) obtained in verifying the resource properties can be used in two ways:

Retail PCC. The proofs are bundled with the bytecode and downloaded onto the handset. The handset checks the proofs before installing the application.

Wholesale PCC. The proofs are checked by a trusted third party (e.g. the operator), who signs the application, as described in Section 2.4.3. In this case, the proofs are not downloaded onto the handset, and the handset just checks the signature before installing the application.

5.3 Approaches to formalization

The previous sections have explored a range of resource security issues where formal guarantees would enhance the trustworthiness of mobile code, but where they are not currently available. In response to this, we now survey some techniques that can provide such guarantees in the setting of proof-carrying code. We highlight the need for a formal semantics of resource usage, and describe technologies that support the generation and transmission of resource proofs.

The ideas examined in this section set a basis for research in later project work packages: in particular Tasks 2.3, 2.4, 3.1, 3.2, 4.3 and 4.4.

Taking account of the candidate resources and scenarios described earlier, we have identified the following requirements for the MOBIUS resource security platform:

- An *instrumented* operational semantics for JVM bytecode, for precise calculation of resource usage.
- Formal languages for expressing and reasoning about resource properties of JVM bytecode.
- Methods for automatically creating proofs about resources, possibly directed by high-level types or logical assertions.
- A framework to embody all these as proof-carrying code; where clients can set resource policies and reliably check incoming code against them.

The later parts of this section address each of these requirements in turn. To these we can also add two nonfunctional requirements, for *modularity* and *trustworthiness.* For true global computing, it is important that the MOBIUS framework be modular wherever possible: allowing the composition of separate units of resource-guaranteed code; and supporting different kinds of resource, annotation, or proof technique. Furthermore, for each of the steps listed above trust in the system is increased if we can provide a mechanical proof of correctness: for example, proof in a formal system like Coq or Isabelle that a resource logic is sound with respect to the instrumented bytecode semantics. There are already some such proofs for existing systems, and we expect that they will be part of the further MOBIUS development.

The partners in MOBIUS bring experience from existing European projects in this field, and here we have drawn in particular on the following: *Mobile Resource Guarantees* (MRG) of UEDIN and LMU [3]; INRIA work on a bytecode specification language [8, 13] and certified abstract interpretation [14]; and the Ciao system of UPM [1].

We naturally focus here on research related to the project at hand: for a more general review of work on formalizing safety of Java and the JVM see, for example, [39].

5.3.1 Resource-aware semantics

Any formal work on resource security in Java application must be grounded in a semantics of the language that tracks the use of resources. As we have seen in Section 5.1, this needs to cover a variety of different kinds of resource; ideally, we would like a flexible system that can adapt to several of these.

The MOBIUS objective of proof-carrying code also requires that this resource semantics be at the level of JVM bytecode: for this is what a PCC verifier will be checking, and it will not in most cases even have access to higher-level source code. This does not rule out annotation and proofs on Java source code; but they must all translate meaningfully to the bytecode semantics.

Developing a suitable resource-aware formal semantics for JVM bytecodes is an objective of Tasks 3.1 and 3.2. Here we review some existing work on the problem by partners LMU and UEDIN, and describe how it can be applied in MOBIUS.

Grail bytecode

The Java Virtual Machine allows very free control structure for arbitrary bytecode, which can be a challenge for resource analysis — even though the bytecode generated by compilers is usually highly stereotyped, with structured control flow. The MRG project took account of this with a functional form of JVM bytecode named *Grail* [11]. Grail retains the object and method structure of the JVM, but represents method bodies as sets of mutually tail-recursive first-order functions. The format shows many similarities with A-normal form and Static Single Assignment (SSA), widely used in compiler implementation. This language has the full computational power of JVM, but analysis of control flow and stack use is greatly simplified.

MRG implemented a formal semantics for Grail in the Isabelle theorem prover. This is an instrumented semantics, carrying information about basic resource usage: specifically instruction count and heap allocation. Building on this formalization, Isabelle was used to machine-check the correctness of the Grail semantics and these resource assertions.

For MOBIUS, we aim to address the full JVM language, and a wider range of possible resources. This is more ambitious undertaking, but the approach remains valid and we anticipate that carrying out mechanically-supported correctness proofs will be a part of assuring confidence in this large formal system.

Resource algebras

An instrumented semantics like that described above is sufficient for basic reasoning about a single chosen resource. However, for MOBIUS we want a more general system to handle a variety of resource types. One promising approach is to use *resource algebras* [2].

A resource algebra provides exactly the data required to instrument JVM instructions and allow quantitative calculation of resource use by bytecode programs. For every kind of resource there is a characteristic resource algebra, and algebras can be combined to track several resources at once. This provides a modular approach to instrumenting code, and supports future extension to additional resource types.

For example, the paper [2] presents the following as resource algebras: execution time, heap space, maximum stack depth, methods call counts, and maximum frequency of specific method calls. More exotic resource algebras include complete execution traces; method guards, which are boolean resources that check whether some test holds on every call of a certain method; and even parameter policies that validate specific method arguments.

5.3.2 Expressing resource properties of bytecode

An instrumented bytecode semantics provides raw information about use of resources; but to reason about these, or express requirements and properties of bytecode, we need some richer language. In MOBIUS this language will provide the material for writing resource policies, and for supporting modularity where one piece of code can specify what resource behaviour it requires of other components in a system.

In this section we look at some existing approaches to this problem, which we expect to contribute to the development of the MOBIUS logic in Tasks 3.1 and 3.2.

Bytecode logic

The MRG project developed a general-purpose program logic for the Grail language of JVM bytecode [2]. At its simplest, a judgement in the logic takes the form $G \rhd e : P$ and states that a program expression e satisfies an assertion P, dependent on a context G which stores assumptions for recursive program structure. The assertion P may include statements about resources, described by an appropriate resource algebra. The logic has rules for building up judgements about arbitrary bytecode programs, and in particular takes account of method calls and object manipulation.

There is a formalisation of the bytecode logic in Isabelle/HOL, which has been proved sound and complete with respect to the instrumented operational semantics described earlier. This is a shallow encoding, so that assertions and proofs in the logic can employ the full power of higher-order logic.

Derived assertions

The logic just described is extremely powerful, and we expect some such proven system to form a basis for trust in the MOBIUS framework. However, it is not necessarily suitable for automatic generation of proofs, nor for lightweight proof-checking by PCC clients. One way to handle this is to define a specialised proof system over the base logic, that more closely matches high-level assertions about program code.

A system of *derived assertions* like this, relative to the Grail bytecode logic, is described in [10]. There the derived system contains assertions that correspond to particular source language datatypes: for example, a predicate $h \models_{\tau list} a$ stating that an address a in heap h contains the start of a linked list of objects. Rules for building derived assertions are similar to syntax-directed typing, and it is possible to turn high-level type-checking into an effective proof tactic.

Each derived assertion expands into an expression in the bytecode logic that directly handles the low-level resource assertions. To show the correctness of the derived logic, it is enough to prove each of its rules sound with respect to the bytecode logic; once this is done, we can work directly at the higher level of the specialised system. In MOBIUS we anticipate that this technique will be important in lifting trust from level of instrumented operational semantics to a language for specifying resource policies.

Resource policies

Any global computing scenario like those envisioned by MOBIUS, where code may be sent to different devices, or a single device may combine code from different providers, requires flexible and modular proof checking. Work in [4] explores a general form of *resource policies* supporting this which may be appropriate for MOBIUS.

Two forms of policy are used: *guaranteed* policies which accompany code and *target* policies which describe limits of the device. A guaranteed policy is expressed as a function of method inputs, which then determine a bound on resource consumption: for example

> "for positive integer inputs `n` and `m`, executing the method call `calc(int m,int n)` requires at most $16 + 42 * \mathtt{m} + 9 * \mathtt{m} * \mathtt{n}$ JVM instructions to be executed."

A target policy is defined by a constant bound and input constraints for a method. For example:

> "for all inputs $\mathtt{n} < 10$ and $\mathtt{m} < 10$, executing the `calc(int m,int n)` method must take no more than 2000 instructions."

In the simplest case, a recipient of mobile code chooses whether to execute it by comparing its guaranteed policy with the local target policy. More generally, since delivered code may use methods implemented on the target machine, guaranteed policies may also be provided by the platform; and a client can also use this to compose several code units from different providers, each providing resource guarantees required by the other.

The Java platform already provides a rather sophisticated security policy management scheme, and [4] also shows how resource policies can be brought within that: by implementing a security manager class that assesses resource proofs and policies rather than checking cryptographic signatures.

Modeling languages for resource policies

The *Java modeling language* (JML) [47] is a specification language for annotating Java source code with assertions that express its desired behaviour. It is actively used in research projects worldwide, with a number of Java tools supporting JML as a common annotation language.

A natural place for Java programmers to provide information about resource usage and requirements is as part of source code, and we expect JML annotations to be an important form of resource specification in MOBIUS. In this mode, UEDIN has studied the feasibility of specifying heap space usage with JML in Java source code, and verifying such specifications with ESC/Java2.

In fact JML has existing "`working_space`" and "`space`" annotations, but these were previously implemented purely as comments. UEDIN extended the ESC/Java2 machinery to give force to these: placing additional information about memory consumption in the ESC/Java2 semantics, and generating appropriate verification conditions to check correct heap usage. To support the range of resources described earlier, MOBIUS Task 3.2 and ultimately Work Package 4 will need to carry out similar enrichment of JML and supporting tools.

As noted earlier, for proof-carrying code MOBIUS must develop tools that work on bytecode, and so we shall need to map JML annotations to this lower level. INRIA has developed a Bytecode Modeling Language (BML) [13], a variant of JML tailored to bytecode. BML has also been used to reason about resources, specifying memory consumption behaviour Java applets [8]. Once described in BML, these behavioural assertions are passed to a verification condition generator, and then to an automated proof engine or, if necessary, an interactive proof assistant. We anticipate following a similar framework within MOBIUS, with BML extended to support more general resource reasoning, and formally validated against an instrumented JVM semantics.

5.3.3 Generating resource proofs

We have seen in the previous section a range of techniques for proving resource properties of programs. In the simplest cases we can construct such proofs by hand, or in interactive proof assistants. For more general applications, we aim that wherever possible the MOBIUS framework should support automatic creation of resource proofs; possibly guided by high-level source annotations. Developing technology to do this will be a part of Tasks 4.3 and 4.4, and we review here three different approaches.

Type systems for resources

Work at LMU and UEDIN in MRG included extensive development of resource types, in particular to describe heap space usage [40, 43]. This is done by enriching conventional function types, which indicate the argument and result of a function, with information about the memory space required during its execution. Type inference for this system, together with a standard linear program (LP) solver, can automatically generate heap space bounds for code to manipulate data structures like linked lists or trees. Moreover, the type checking procedure for this system can be translated to a resource proof on JVM bytecode in the derived assertion language described earlier [10].

Recent work [41] extends this to a resource type system for object-oriented programs. This is based on an amortised complexity analysis, and is sufficient to reason about space usage of more complex structures such as doubly-linked lists.

High-level resource logics

Logic-based verification can in general establish a richer set of properties about program behaviour than purely type-based methods. However this increase in expressiveness comes at the cost that it is not so easily automated as type inference. JML specifications can assist this, by giving program developers a way to express preconditions, postconditions, invariants and frame properties to aid logic-based resource verification of Java source code.

A source-level proof, however, is not sufficient for proof-carrying bytecode. We propose to bridge this gap with *proof-transforming compilation*: an advance on the certifying compilation of classic PCC, by compiling JML specifications and proofs into BML proofs. Developing these compilation techniques is the aim of Task 4.4. For the particular case of resource proofs, one notable challenge is the issue of optimization: any code rearrangement by the compiler may affect resource usage, and a proof-transforming compiler must be able to demonstrate that any such changes respect resource guarantees.

Abstract interpretation

Abstract interpretation provides another method for statically bounding program resource usage. UPM have investigated *abstraction carrying code*, where an abstracted model of a program, computed by standard static analysis, serves as the certificate to be checked by a code consumer. This can then be checked by a simplified single-pass abstract interpreter before client execution. There is an implementation based on constraint logic programming in the Ciao preprocessor [1].

Separately, INRIA have developed algorithms for analysis of resource usage, verifying that programs execute in bounded memory [14]. This is based on loop-detection and data flow to estimate loop repetition, and experiments on a test suite of midlets indicate that this approach can verify the safety of a majority of applets. Current work is directed at extending this analysis to access control for resources, as in the scenario of Section 5.2 above.

Chapter 6

Conclusions

We have considered the resource and information flow security requirements relevant to global computing that will be studied and addressed throughout the project.

An important conclusion for the rest of the project is the decision to focus on the MIDP platform, a specialization of Java, widely deployed on mobile phones. MIDP is general enough to raise a wide range of security problems; e.g. unwanted information flow from on-device databases to SMS or network connections, control of billable resources, and deadlock. MIDP is also one of the most tightly specified Java based profiles. Finally, the MIDP security model, with off-platform bytecode pre-checking and digital signing of bytecode to indicate trust level, gives a hook for implementation of PCC techniques without changing platform implementation (Work Package 4). For these reasons, interesting case studies are possible with the MIDP platform (Work Package 5)

The work in chapter 4 strongly suggests the need for declassification policies. Addressing information-flow policies in bytecode-level multithreaded languages with declassification is a long-term goal in the MOBIUS project, subject to further security requirement gathering within Work Package 1. The requirements set out in chapter 4 will be used in Tasks 2.1 (type based enforcement of information-flow policies), 2.2 (mechanisms for safe information release), and 3.5 (combining static analysis and logic-based methods for security).

The work in chapter 5 shows that many interesting resources (e.g. number of SMS messages, number of network connections, use of persistent storage) are all controlled by certain method calls with parameters. This is the basis for a mechanism for reasoning about usage of these resources. The requirements set out in chapter 5 will be used in Tasks 2.3 and 2.4 (resource types) and 3.2 (logics for resources).

We have presented scenarios for both information flow and resource security. These scenarios, developed with our industrial partners, connect the theoretical aspects of the work with real life application problems.

Chapters 4 and 5 also discuss state of the art in theory about information flow and resources, including recent work by MOBIUS partners, and projected future work. A significant aspect to be studied in later workpackages is the use and development of Java Modelling Language (JML) to adequately annotate source programs for the properties needed for secure program behaviour. From annotated programs we will derive verification conditions. Another significant aspect of MOBIUS is then to prove that these verification conditions hold in as automatic a fashion as possible.

The requirements set out in this document may be extended or refined if work on later Work Packages suggests changes. There is another requirements document, Deliverable D1.2, due later, covering higher level Framework-specific and Application-specific security policies.

Bibliography

[1] Elvira Albert, German Puebla, and Manuel V. Hermenegildo. Abstraction-carrying code. In Franz Baader and Andrei Voronkov, editors, *LPAR*, volume 3452 of *Lecture Notes in Computer Science*, pages 380–397. Springer, 2004.

[2] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. A program logic for resource verification. In *TPHOLs2004*, volume 3223 of *LNCS*, pages 34–49, Heidelberg, September 2004. Springer.

[3] David Aspinall, Stephen Gilmore, Martin Hofmann, Donald Sannella, and Ian Stark. Mobile resource guarantees for smart devices. In *CASSIS 2004*, volume 3362 of *LNCS*, pages 1–26. Springer, 2005.

[4] David Aspinall and Kenneth MacKenzie. Mobile resource guarantees and policies. In *Proc. Intl. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS 2005).*, 2005. to appear.

[5] A. Banerjee and D. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15:131–177, March 2005. Special Issue on Language-Based Security.

[6] G. Barthe, P. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In R. Foccardi, editor, *Proceedings of CSFW'04*, pages 100–114. IEEE Press, 2004.

[7] G. Barthe, D. Naumann, and T. Rezk. Deriving an information flow checker and certifying compiler for java. In *Symposium on Security and Privacy, 2006*. IEEE Press, 2006.

[8] G. Barthe, M. Pavlova, and G. Schneider. Static analysis of memory consumption using program logics. In *3rd IEEE International Conference on Software Engineering and Formal Methods*. IEEE, IEEE Computer Society Press, September 2005.

[9] G. Barthe and T. Rezk. Non-interference for a JVM-like language. In M. Fähndrich, editor, *Proceedings of TLDI'05*, pages 103–112. ACM Press, 2005.

[10] Lennart Beringer, Martin Hofmann, Alberto Momigliano, and Olha Shkaravska. Automatic certification of heap consumption. In Andrei Voronkov Franz Baader, editor, *LPAR 2004*, volume 3425 of *LNCS*, pages 347–362. Springer-Verlag, February 2005. ISBN: 3-540-25236-3.

[11] Lennart Beringer, Kenneth MacKenzie, and Ian Stark. Grail: a functional form for imperative mobile code. In *Foundations of Global Computing*, number 85.1 in ENTCS, pages 1–21. Elsevier, June 2003.

[12] Cynthia Bloch and Annette Wagner. *MIDP Style Guide for the Java 2 Platform, Micro Edition*. The Java Series. Addison-Wesley, 2003.

[13] L. Burdy and M. Pavlova. Java bytecode specification and verification. To appear at SAC'06, 2005.

[14] David Cachera, Thomas Jensen, David Pichardie, and Gerardo Schneider. Certified memory usage analysis. In *FM 2005: Proceedings of the International Symposium of Formal Methods Europe*, LNCS 3582, pages 91–106. Springer-Verlag, 2005.

[15] CLDC and the K Virtual Machine (KVM). `http://java.sun.com/products/cldc`.

[16] STIP Consortium. GDP/STIP 2.2 Specification for Java. Technical report, Global Platform, 2004.

[17] Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors, *Proc. 2nd International Conference on Security in Pervasive Computing*, volume 3450 of *LNCS*, pages 193–209. Springer-Verlag, 2005.

[18] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[19] NTT DoCoMo. DoJa 1.5 overseas edition - DoJa java content developer's guide. Technical report, NTT DoCoMo, 2002. Available from `http://www.doja-developer.net`.

[20] G. Dufay, A.P. Felty, and S. Matwin. Privacy-sensitive information flow with JML. In R. Nieuwenhuis, editor, *Proceedings of CADE'05*, volume 3632 of *LNCS*, pages 116–130. Springer-Verlag, 2005.

[21] Orange France. Annexe 4 : Charte de realisation d'un service gallery. A similar document is downloadable from Bouygue Telecom as 'Annexe 12 - Charte de telechargment Gallery' at `http://leskiosques.bouyguestelecom.fr/p25.php`, 2004.

[22] Adam Gowdiak. Java 2 micro edition (j2me) security vulnerabilities. In *Hack In The Box Conference*, Kuala Lumpur, Malaysia, 2004.

[23] JSR 118 Expert Group. Mobile information device profile (MIDP), version 2.0. Java specification request, Java Community Process, November 2002.

[24] JSR 120 Expert Group. Wireless messaging API. Java specification request, Java Community Process, 2003.

[25] JSR 135 Expert Group. Mobile media API. Java specification request, Java Community Process, June 2003.

[26] JSR 177 Expert Group. Security and trust services API for J2ME. Java specification request, Java Community Process, September 2004. Final release.

[27] JSR 179 Expert Group. Location API for J2ME. Java specification request, Java Community Process, September 2003.

[28] JSR 185 Expert Group. Java technology for the wireless industry (JTWI), release 1. Java specification request, Java Community Process, 2003.

[29] JSR 195 Expert Group. Information module profile. Java specification request, Java Community Process, July 2003.

[30] JSR 205 Expert Group. Wireless messaging API (version 2.0). Java specification request, Java Community Process, June 2003.

[31] JSR 218 Expert Group. Connected limited device configuration (CLDC), version 1.1. Java specification request, Java Community Process, 2002.

[32] JSR 228 Expert Group. Information module profile - next generation (imp-ng). Java specification request, Java Community Process, September 2005.

[33] JSR 242 Expert Group. On ramp to OCAP profile. Java specification request, Java Community Process, August 2005. Digital Set top box profile — proposed final draft.

[34] JSR 248 Expert Group. Mobile service architecture (MSA). Java specification request, Java Community Process, 2005. Early draft.

[35] JSR 30 Expert Group. Connected limited device configuration (CLDC), version 1.0. Java specification request, Java Community Process, 2000.

[36] JSR 37 Expert Group. Mobile information device profile (MIDP), version 1.0. Java specification request, Java Community Process, 2000.

[37] JSR 75 Expert Group. PDA optional packages for the J2ME platform. Java specification request, Java Community Process, June 2004. Final release.

[38] JSR 82 Expert Group. Java apis for bluetooth. Java specification request, Java Community Process, March 2002.

[39] Pieter H. Hartel and Luc Moreau. Formalising the Safety of Java, the Java Virtual Machine and Java Card. *ACM Computing Surveys*, 33(4):517–558, December 2001.

[40] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *POPL'03*, pages 185–197, New Orleans, LA, USA, January 2003. ACM Press.

[41] M. Hofmann and S. Jost. Type-based amortised heap-space analysis for an object-oriented language. In *Proceedings of the ESOP 2006*, LNCS. Springer, January 2006. To appear.

[42] Unified Testing Initiative. Unified testing criteria for java technology-based applications for mobile devices, version 2.0. Technical report, Sun Microsystems *et al.*, May 2005. `http://javaverified.com`.

[43] Steffen Jost. `lfd_infer`: an implementation of a static inference on heap space usage. In *Proceedings of Second Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE 2004)*, 2004.

[44] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of POPL'73*, pages 194–206. ACM Press, 1973.

[45] Jonathan Knudsen. Creating 2D action games with the game API. Sun Technical Articles and Tips `http://developers.sun.com/techtopics/mobility/midp/articles/game/`, 2002.

[46] Jonathan Knudsen. Networking, user experience, and threads. Sun Technical Articles and Tips `http://developers.sun.com/techtopics/mobility/midp/articles/threading/`, 2002.

[47] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, and Joseph Kiniry. JML reference manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, July 2005.

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

[49] Qusuay H. Mahmoud. Preventing screen lockups of blocking operations. Sun Technical Articles and Tips `http://developers.sun.com/techtopics/mobility/midp/ttips/screenlock/`, 2004.

[50] J. McLean. The specification and modeling of computer security. *Computer*, 23(1):9–16, January 1990.

[51] A.C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of POPL'99*, pages 228–241. ACM Press, 1999. Ongoing development at `http://www.cs.cornell.edu/jif/`.

[52] Common Criteria Sponsoring Organisations. Common criteria for information technology security evaluation. Technical report, Government security agencies of Canada, France, Germany, Netherlands, UK, USA, August 1999. Also ISO15408:1999.

[53] DVB project. Digital video broadcasting (DVB); multimedia home platform (MHP) specification 1.2.1. Technical Report TS 102 812 V1.2.1, ETSI, May 2005.

[54] V. P. Ranganath, T. Amtoft, A. Banerjee, M. B. Dwyer, and J. Hatcliff. A new foundation for control-dependence and slicing for modern program structures. In Mooly Sagiv, editor, *Proceedings of ESOP*, pages 77–93, 2005.

[55] A. Sabelfeld. Confidentiality for multithreaded programs via bisimulation. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2890 of *LNCS*, pages 260–273. Springer-Verlag, July 2003.

[56] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 255–269, 2005.

[57] Sun Microsystems. The Java Card 2.0 language subset and virtual machine specification. Technical report, 1997.

[58] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In C. Hankin and I. Siveroni, editors, *Proceedings of SAS'05*, volume 3672 of *LNCS*, pages 352–367. Springer-Verlag, 2005.

Appendix A

Kick-Off Meeting

The MOBIUS kick-off meeting took place at INRIA Sophia-Antipolis, on 6–8 October 2005. There were separate sessions on Information Flow and Resource Security Requirements, Tasks 1.1 and 1.2, both open to all MOBIUS project participants. Each session was chaired by the task leader, with presentations by various task members as well as open discussion periods. The session agendas are below; slides from all the prepared presentations are available on the project website.

Information Flow Security Policies

Participants: CTH (leader), IC, INRIA, IoC, RWTH, TL, FT, TLS

- Introduction (CTH)
- TL and TLS report input on attacker models (responsible: TL)
- FT reports input on attacker models (responsible: FT)
- INRIA, IC, and IoC report their input on termination-insensitive security policies (responsible: INRIA)
- RWTH and CTH report their input on timing-sensitive security policies (responsible: RWTH)
- Discussion on consolidating information-flow security requirements (moderator: CTH)

Resource Security Policies

Participants: FT, INRIA, LMU, TLS, TL, UEDIN (leader), UPM.

- Session plan (Stark)
- Candidate resources (Stark)
- Scenarios for resource control (Crégut)
- Approaches to formalization — achievements so far in applying resource policies
  - Edinburgh: Mobile Resource Guarantees and Policies (MacKenzie)
  - INRIA Rennes (Jensen)
  - UPM (Puebla)
  - INRIA Sophia-Antipolis: Memory Consumption Policies (Pavlova)
- Discussion: Criteria for Mobius resource policies (Stark)