Project N°: FP6-015905

Project Acronym: MOBIUS

Project Title: Mobility, Ubiquity and Security

Instrument: Integrated Project

Priority 2: Information Society Technologies

Future and Emerging Technologies

Deliverable D1.2

Framework- and Application-Specific Security Requirements

Due date of deliverable: 2006-08-31

Actual submission date: 2006-10-09

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

Organisation name of lead contractor for this deliverable: RU

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:
Framework- and Application-Specific Security Requirements

This document describes the results of tasks 1.3 and 1.4 of the MOBIUS project. The work in these tasks was security requirement engineering: the aim was to gather representative security requirements that we want the MOBIUS Proof Carrying Code (PCC) infrastructure to be able to certify, to categorise these security requirements, and to investigate ways of formalising them.

Task 1.3 considered framework-specific security requirements, by which we mean 'generic' security requirements that come with the use of a given framework. Here we focused on the MIDP framework, the dominant industry standard for mobile devices, and the most mature and widely-used framework for mobile code that exists today.

Task 1.4 considered application-specific security requirements, additional security requirements that are specific to individual applications that run on the framework or are part of the framework. Here we considered a range of applications with interesting security requirements, including components that play a crucial role in the trust infrastructure of the platform itself (e.g. for secure communication).

Contents

Chapter 1

Introduction

This document reports on work done in Task 1.3 and Task 1.4 of the MOBIUS project. Goal of the these tasks was to gather a broad range of representative security requirements and investigate ways in which various categories of such requirements can be expressed formally. Specification of security requirements in a rigorous, formal language is a prerequisite for certification by means of Proof Carrying Code (PCC). In other words, this documents answers the questions: 'What kind of properties would we want to certify?' and 'How can we specify these properties so that we will be able to certify them?'.

Task 1.3 considered framework-specific security requirements: these are 'generic' security requirements that are imposed on all applications running on a given framework or platform, or large classes of such applications. Framework-specific security requirements are typically meant to reduce threats to assets of the platform.

Task 1.4 investigated application-specific security requirements, additional security requirements specific to a particular application. This can be an application running on the platform, or a component of the platform itself. Application-specific security requirements can simply be more specialised versions of framework-specific requirements; many examples of such security requirements are discussed in Chapter 3.[1] However, additional security requirements may also be imposed to protect assets of the application itself, rather than assets of the platform. Moreover, its security requirements may involve (aspects of) its functional specification, especially in the case the application provides some security-critical functionality.

The work in Task 1.3 has resulted in one of the few examples in the public domain of a comprehensive set of concrete security requirements for a real-life industrial standard, included here as Chapter 3. We expect that this will be a useful catalyst for stimulating research, not only among the MOBIUS partners, but in a wider academic and industrial community[2]. Indeed, a similar but far less extensive set of concrete security requirements [MM01], for the Java Card framework, which was produced in an earlier EU project SecSafe, has in the past proved very successful in sparking of research into Java Card security.

The concrete applications from Chapter 4 will continue to serve as running case studies to test some of the ideas and tools developed in the course of the MOBIUS project.

[1] In Chapter 3 these specialised versions of framework-specific requirements for a specific application are called custom properties, as opposed to general properties that are required of all applications.

[2] For example in companies such as End User Panel member Parasoft, that develop dedicated code scanning tools.

The choice for MIDP

As already mentioned in deliverable D1.1, at the Kick-Off meeting it has been decided to concentrate work in the MOBIUS project on the MIDP platform.

The MIDP application framework is the main Java framework used on mobile devices, more specifically phones, throughout he world. It is present in one form or another in a large part of the world, and there are today around 1.2 billion Java-powered phones that have been deployed. No other mobile application framework comes close to that number.

MIDP is also a very active area for standardisation, development, and research. The framework is evolving permanently, and additional API's, covering new features, are added every month. The framework itself, already in its second major release (MIDP 2.0), continues to evolve, as work has started on the MIDP 3.0 specification.

The MIDP application framework is therefore today largely dominant on mobile phones, and it is likely that it will remain a significant framework for a few years. It also is based on the Java language and runtime environment, which is well suited for static analysis. Finally, the fact that this environment targets all categories of phones, including the least sophisticated ones, makes it a possible candidate as application framework for other devices used in ubiquitous computing, such as sensors[3] [SCC+06].

For all these reasons, the focus of our work is the MIDP application framework. This is by no means a restriction, as the security challenges for MIDP are representative of those for any widely distributed computing infrastructure that supports code mobility: it involves security concerns of various parties – end users, service providers, and network operators – and it comprises components from more or less trusted sources. Concentrating efforts on a concrete platform will maximise the results and impact of the project, especially since MIDP is already the most-widely used platform for mobile and networked devices today. Moreover, the extensive use of the MIDP application framework means there is a lot of experience and industry expertise that we can built on.

Outline

The outline of the rest of this document is as follows

- Chapter 2 discusses fundamental Java security vulnerabilities that Java applications have to protect themselves against, and which certification should ultimately prove the absence of. The chapter proposes some bottom-line security requirements for Java applications, which can be considered as framework-specific security requirements for any Java platform.

- Chapter 3 provides a comprehensive inventory of framework-specific security requirements for the MIDP platform, i.e. security requirements that apply to all or large classes of MIDP applications, and which are typically included in the operator security policies. The chapter also lists some examples of application-specific requirements, that are essentially just specialised, more restrictive, versions of the framework-specific security requirements that are listed; these are called custom properties.

- Chapter 4 investigates application-specific security requirements for a range of applications, namely

[3]Some adaptations of the technology have already been performed by Sun Microsystems.

  - a 'Pocket Protector' application for storing sensitive information,
  - an instant messenger application 'Mobber',
  - an application for remote electronic voting,
  - SSH,
  - a bytecode verifier, and
  - an access controller that is part of the MIDP framework.

- Chapter 5 investigates the possibilities of formally expressing the security requirements listed in the preceding chapters. The emphasis here is in formally specifying the security requirements in JML [LPC+05], as its bytecode cousin BML [BP06] is the core specification language used in the work on logic-based verification that provides the foundation of the MOBIUS Proof Carrying Code infrastructure.

Chapter 2

Java security

This chapter discusses Java security and some of the Java vulnerabilities that applications and platform components have to protect themselves against. The reason for including this chapter is that knowledge about these vulnerabilities is crucial if we want to certify security of Java applications. The earlier deliverable D1.1 gives an extensive discussion of MIDP security (the MIDP security model and its shortcomings are discussed in Chapter 2 and a threat model for MIDP is given in Chapter 3), but it does not consider vulnerabilities of the underlying Java platform.

Note that when defining security requirements one has to consider the entire software stack one is dealing with, i.e. the Java language itself, consisting of the VM and the Java API, any additional platform APIs, and any applications running on top of this. For each level there may be assets to be protected, and vulnerabilities to protect against. For instance, Chapter 2 and 3 of deliverable D1.1 give a detailed security analysis of the MIDP platform, and based on this, Chapter 3 provides an inventory of security requirements that MIDP application should meet in order to protect assets of the MIDP platform, most of which are accessed via the MIDP API. Chapter 4 of this deliverable considers additional security requirements for specific applications and for specific components that are part of the platform. This chapter concentrates on vulnerabilities that exist on the lowest level, namely the Java language itself.

2.1 Java security architecture

When it comes to security, the Java language offers two lines of defence:

- the type system;
- access control via visibility modifiers.

Type-safety prevents unfettered access to raw memory, and restricts interactions between applications. The visibility modifiers private, protected, public - and the implicit package level of visibility - provide an additional basic mechanism for access control. Other memory- and type-safe languages aimed at the execution of untrusted code, for example (the verifiable part of) C#, include similar features.

Type safety is not only useful to prevent an untrusted application from interfering with other applications or the platform, but it is also useful inside a single application when it comes to certification. Rather than certifying a whole big application, maybe certifying only

part of the application is enough, if the access to this part of the application from the rest is controlled.

The Java execution platform offers additional lines in defence, for instance

- the sandbox, for standard Java;
- the simplified version of this for MIDP, with a distinction between trusted and untrusted MIDlets;
- a firewall, for Java Card and CLDC hotspot.

These additional lines of defence are enforced dynamically, i.e. at runtime, and not statically, and the policies that they can enforce are limited. For MIDP this has its drawbacks, as discussed in Chapter 2 of deliverable D1.1. Exactly the same drawbacks apply to the standard Java sandbox. (By the way, one obvious role for a PCC infrastructure would be to enforce these restrictions statically, as opposed to dynamically. This is in included as framework-specific security requirement G-7 on page 19 in Chapter 3). The security guarantees that the Java platform provides rely on for example

- correctness of the bytecode verifier;
- correctness of class loader;
- correctness of runtime monitoring code, eg. in the security manager;
- absence of native code.

The bytecode verifier, a prime example of a component one would want to certify, is considered in Chapter 4.5.

2.2 Java security vulnerabilities

One tends to think of a Java application as being constructed from components, which communicate in a controlled way via invocations of methods they provide as interface to the outside world. However, one should be aware that Java components can interfere with each other through other channels than method invocations. It is important to be aware of these possibilities, because if the MOBIUS certification infrastructure fails to take these into account, any guarantees it provides are void. These Java vulnerabilities listed should naturally come up when we establish the soundness of the certification approach developed in the project, but it is better to be aware of them to prevent any of them being overlooked. Indeed, it is not inconceivable that in the soundness proof one fails to take one of these avenues to attack an application into account ...

A limitation of access control in Java through the visibility modifiers is that it relies on programmer discipline. In the same way that we may not trust the user of a mobile phone to grant access to eg. sending an SMS, we may not trust an application programmer not to store sensitive information in public fields. The visible fields of a component are just as much part of its interface as its visible methods, and certification of components should not only make guarantees about (ab)use of the component via its methods, but also about (ab)use of the component via any fields that are exposed.

A further limitation of Java access control here is that access control is very weak for all fields or methods that are not private. Obviously, public fields are dangerous, and their inclusion in the language was probably a mistake. But access to all fields that are not private are not as well-controlled as one might hope or expect: protected fields are accessible from within the package, and there is nothing to prevent an attacker to declare hostile code in any package. One exception here are special packages, notably java.lang, to which the Java Class Loader will refuse to add new classes.

Note that by the reasoning above, code should not just be certified against abuse via its public methods, but also against abuse via its package or protected methods!

To counter the risk of attacker code directly accessing protected or package visible fields, standard secure programming guidelines for Java, eg. those in [MF99], require all fields to be private. This would be a good security requirement to enforce, but one should make an exception for final fields to allow the use of for instance final static integer fields as constants:

J-1	All fields should be private, except final fields that have a primitive type and final fields that are references to immutable objects (e.g. strings).

Rationale: Public fields cannot be protected from modification by hostile code, and neither can package-visible or protected fields, as hostile code may declare itself as belonging to a given package. Final fields can be public, but not if they are references to mutable objects. In particular, final static arrays that are not private, (as for instance used in the SSH implementation discussed in Section 4.4) pose a security risk, as anyone can modify them.

Because inner classes do not exist at the level of Java bytecode, bytecode verification cannot protect access to inner classes. A good security requirement to enforce is

J-2	No use is made of inner classes.

Rationale: Visibility modifiers, such as private, in inner classes are not enforced, as explained in [MF99].

Even bearing in mind these limitations, it is misleading to think that all interactions with an object, class, or package are via their visible interfaces, ie. the collection of methods and fields they provide that have a visibility that the client (or attacker) can see. There are other avenues that an attacker might exploit, and that a careless implementation might leave open.

A serious limitation of Java access control is caused by the use of references. The language cannot express or enforce any restrictions on the sharing of references. This has been the source of notorious security flaws in the past, for example the flaw in JDK1.1.1 which leaked a reference to the internal array containing the signers of a gives class (see http://java.sun.com/security/getSigners.html. A sound PCC framework should not allow the certification of any component that leaks references to critical internal datastructures. Work on alias control in WP2 and WP3 will of course address this issue. In general, any passing of references as arguments or results of method invocations may be a risk, unless they are references to immutable objects. It is of course no coincidence that many API methods take immutable strings as arguments. A good security requirement to enforce is

J-3	No non-private method takes references to mutable objects as arguments or returns reference to mutable objects as results. No non-private constructor takes references to mutable objects as arguments.

Rationale: This rules out methods and constructors that expose or import shared references.

These rules may be too strict in practice. Indeed, leaking or importing references to mutable arguments is fine as long as an application does not rely on any properties of such shared objects. The modular verification techniques considered in Task 3.4 will provide ways of relaxing this restriction, as a sound technique for modular verification will disallow relying on any properties of shared objects.

There are other avenues of attack for a hostile application. For example, one such avenue of attack is the introduction of new subclasses. If classes are not declared as final, an attacker can attack an application by declaring a subclass, and inject ill-behaved objects of this subclass into a subclass into an application. Similarly, deserialisation procedures and possibly also – though less likely – cloning can be used to construct ill-behaved objects. However, if we use a sound verification approach, the possibility of such attacks will show up. For example, if our certification relies on some properties of a non-final class that an attacker might subclass, certification of such a hostile subclass will fail. Similarly, if our certification relies on some assumptions regarding arguments of a (public) deserialisation method, these assumptions will show up in the preconditions of this method, and certification should not rely on any preconditions of public methods being met by malicious code.

2.2.1 Limitations in the MIDP setting

A characteristic of the MIDP platform that we are considering is that only one MIDlet is active at a time. This may simplify the attacker model; most importantly, it means that an application running on the platform cannot be threatened by another application running concurrently. This may mitigate some of the threats mentioned above in certain situations. For instance, if we trust the platform API, then exposing important data through public fields in a MIDlet is not a security risk, as there is no hostile application running concurrently that could read or modify these fields. In this case, we could drop restrictions rule J-1, rule J-2, and rule J-3 above.

Any decision to rely on this or possible other characteristics to simplify the attacker model should of course be made explicit. With the rapid evolution of platforms such characteristics are bound to disappear in the long run.

When certifying API components as opposed to MIDlets we should of course never assume that we are not running concurrently with hostile code, and we should make sure that we meet the security requirements listed above.

2.2.2 Modularity within an application

The requirements above are about restricting the interaction between an application and its environment to a well-defined interface. This is vital if we want to certify an application without knowing the context in which it will be used. The same requirements can also be applied within an application, restricting the interaction between two parts of an application, so that we only have to certify one of the parts and be sure the other part cannot interfere with it. This may be crucial to scaling the PCC approach.

2.3 Pointers to further information

There are many sources of information — in the literature, on the web and in (the documentation of) code analysis tools – that provide more information on these Java vulnerabilities,

or, more generally, typical (Java) programming errors that lead to security vulnerabilities. Below we mention the most important ones. There is a considerable overlap between some of these lists. Of course, such lists are never complete, but they are very useful when coming up with good sets of security requirements or when looking for security weaknesses in a particular application or platform.

- Sun's Security Code Guidelines
  http://java.sun.com/security/seccodeguide.html
- Chapter 7 of [MF99], also available as online article "Twelve rules for developing more secure Java code"
  http://www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules.html
- David Wheeler's Secure Programming for Linux and Unix HOWTO includes a section on the language-specific issues for Java
  http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/java.html
- The source code analyser FindBugs includes checks for security-related bugs that can lead to known malicious code vulnerabilities
  http://findbugs.sourceforge.net/bugDescriptions.html
- EUP member Parasoft has a more extensive list of secure Java programming guidelines (including some specific for J2ME applications) supported in their jtest source code analyser, but these are confidential.
- The taxonomy of software security bugs by Fortify Software,
  http://vulncat.fortifysoftware.com/
- [GvW03] and [HlV05] do not exclusively focus on Java, but are useful sources of information about general classes of the typical coding mistakes that affect security.

Chapter 3

Framework-specific security requirements

This chapter discussion framework-specific security requirements. It focuses mostly on the MIDP framework, as the best example of a widely used framework providing a global network of mobile applications that is in use today. Other frameworks are mentioned in the last section.

3.1 The MIDP framework

As explained in the introduction, the main focus of the present study is on the MIDP application framework. We start by a quick description of the framework itself, followed by a list of security properties, and we finally identify some of the challenges faced in the verification of these properties. The chapter concludes by a rapid description of the security requirements stemming from other frameworks.

3.1.1 Framework description

The MIDP application framework is the dominant application framework for mobile devices, with over 1 billion available devices over the world. The original work started at Sun in 1998; after the development of the first Java Card virtual machines, Sun started to work on a virtual machine that would use Kilobytes of memory, the KVM. This virtual machine was first released in 1999 on Palm and Motorola devices. The product drew considerable attention in the industry, and quickly led to the development of CLDC 1.0 [JSR00a], a specification for Java on small platforms[1], and of MIDP 1.0 [JSR00b], a complete application framework for the platform, released in September 2000.

This first framework allowed developers to write applications that would interact with the end-user using a simple and portable user interface, and that could open HTTP connections to Web servers. However, this framework was a '1.0' release, with a lot of remaining issues.

Most issues have been fixed in the following releases, CLDC 1.1 [JSR02b] and MIDP 2.0 [JSR02a]. In particular, this release added a deployment framework (including security features based on permissions and digital signatures), a low-level user interface API specifically designed for games, and extended connectivity. In addition, companion specifications defined a variety of APIs for text and multimedia messaging [JSR03a, JSR03d], 2-D and 3-D graphics [JSR05b, JSR03b], location-based services [?], Web services access [JSR04b, JSR05a], personal

[1]This specification, as well as all the other specifications mentioned in the present section, is available on the Java Community Process Web site, free of charge, but after accepting a specific license.

information management [JSR04d], security and trust services [JSR04c], local connectivity [?], etc. This set of specifications is now known as Java ME (for Mobile Edition).

With all these new APIs, fragmentation became an issue.

In its current state, the MIDP framework has reached a state of maturity that allows the development and the deployment of many applications. The framework keeps on evolving, and many issues remain to be addressed. One of them is device fragmentation; because of slight differences between the the implementations available on the devices, the devices are not really interoperable. Devices differ in subtle ways, such as hardware differences (from CPU performance to memory and screen size), and different software bugs. The result is that applications are not really portable, and often need to be adapted for each specific device. This issue mainly remains an engineering issue for developers, especially for deployment, since hundreds of versions of a single applications need to be produced (for instance, an application available in 10 languages on 20 different device families needs to be deployed with 200 different binaries).

This issue was first addressed by the industry through the definition of the Java Technology for the Wireless Industry specification [JSR03c], which specified a minimal set of requirements for an implementation of Java ME. This initial effort was not sufficient to reach the appropriate level of device interoperability, so operators and device manufacturers moved to the definition of the Mobile Service Architecture [JSR05c], which goes further in the specification of the mobile devices. This specification has been recently released, and it will be implemented by future generations of devices.

Another issue is security; most applications deployed so far are standalone games, with very limited security requirements. More features are becoming available, in particular related to network and local connectivity, but they remain seldom used by applications, mostly because of security concerns (most operators don't allow these features to be used by applications because of possible security issues).

3.2 MIDP security

The MIDP framework provides answers to many security requirements, e.g.:

- The executable is bytecode, which runs in a controlled environment, which addresses many of the issues caused by native applications.
- The Java environment includes many safety measures (exception handling, array bound checks, null pointer checks, etc.)
- The MIDP framework includes a permission mechanism, which means that sensitive operations need to be authorized by the end user or by the operator.

However, these answers are not sufficient to address all issues, e.g.:

- The safety measures do not extend to the platform itself; buffer overflows may still be possible in the implementation of the platform.
- Since end users are allowed to authorise sensitive operations, there is an opportunity for social engineering.
- The granularity of the permission system is coarse, so attackers may abuse it.

- The implied security policy only addresses control issues (the access to a given resource) but not information flow issues (how the user data are used).
- Some security schemes, such as JavaVerified [Ini06] or operator security policy also include content control (repression of obscene, offensive, or illegal contents). At the coding level, this means that media resources used by the application should be trackable.
- Some security rules are tied to the operational environment (namely the operator network configuration: the numbering scheme of over-charged phone numbers, the country legal regulations on e-commerce or network use, the protocols available for Java contents, the additional services provided through service platforms, etc.).

Such issues can be transformed into security requirements that need to be checked on applications. The burden of proof typically is with the developer; operators require applications to verify their security policies, and developers need to comply.

The security properties can be at three different levels:

General properties. These properties apply to all applications developed on the MIDP framework, and they are often included directly in the operator security policies.

Device-specific properties. These properties apply only to a single device or a class of devices. They can be used by operators to avoid the exploitation of a bug in the execution framework on the handset or to solve issues raising from potentially different interpretations of the standard, or to provide added security on the use of a proprietary feature available only on some devices.

Custom properties. These properties are general framework properties that can be customised to better fit a given application. The rules are not exactly application-specific, in the sense that the only difference between different instances of the rules are a few simple parameters. Such rules could typically be included in an operator's advanced security policy, requiring developer to provide a specific argument for each parameter in a custom property.

As device specific properties are specific to a given product, they will not be addressed in this report except as broad sub-categories but they may have an important impact in real world validation as new security related bugs are discovered everyday on handsets and it is not possible to upgrade the firmware of most phones. The only practical answer to the security risk is to try to ban applications that may exploit the bug. So some new rules may appear during the life-time of a product.

An operator could quickly specify a new application security profile for a family of applications by selecting a set of general properties and a set of custom properties rules with their corresponding arguments.

One may also note that the general properties are not specific to a framework in their informal definition. However, they become specific to a framework when they are mapped to specific API calls.

For instance, a typical rule in the simplest policies is "No network access is allowed." This rule is generic, but it becomes framework-specific when it is worded as "The protocol component of the argument of the Connector.open method must not be a network protocol." A corresponding custom rule could be: "The only network URL's used as argument to the Connector.open method are HTTP URL's, with host www.somehost.com.

In the rest of this section, we cover a wide range of properties, which are organized in functional groups. For each functional group, we give a short description of the function it covers and of the security issues to be addressed, followed by a few examples of concrete rules. Although the emphasis in this chapter is on framework-specific rules, i.e. the general rules as defined on page 15, we also include examples of application-specific rules that are essentially customised versions of such rules, i.e. the custom rules as defined on page 15. When required, we also present the difficulties that we expect in the verification of some properties, giving some examples when necessary.

3.3 Commonly used concepts

Some of the requirements are intended to be self-sufficient and automatically checked, whereas others may in fact be used to check more complex requirement with a manual process. For instance, a requirement may state that "All graphic file types must be of a known type." Although useful by itself, the most common use of such a requirement is to be able to check the content of the files, for instance in order to verify (manually) that they do not include any aggressive or illegal content.

The most commonly used rules of these types are those that mandate that an argument to a method be constant or determined. These two concepts can be defined as follows:

- An argument is constant if it is defined by a hard-coded constant (without any computation).
- An argument is determined if its value can be determined by the analysis.

The first property is portable, in the sense that it does not depend on the analysis algorithm used for the program. On the other hand, the second property depends on the static analysis algorithms that are used, since a value is only considered determined when the analysis has been able to determine its value. Such a dependency may become an issue for developers, because of its variability. (Moreover, it pre-supposes the correctness of the analysis.) However, requiring an application developer to use only constants is often too restrictive, which is why this concept has been introduced. It may be refined during the MOBIUS project, in order to get to a more precise and portable definition of the concept.

Indeed, when it comes to formalising these notions in JML, we cannot readily discriminate between the notions of being constant and being determined, as explained in Discussion 5.1.2. The possible formalisations discussed there no longer depend on any analysis algorithm.

3.4 Basic framework rules

The rules in this first category concern the basic framework, i.e. the file format, virtual machine, and core API's. In this area, there are a lot of very common requirements: most of them are specification compliance requirements, which require applications to strictly adhere to the Java ME specifications, for instance regarding the contents of binary files.

This category also covers a range of general style requirements, for instance related to the appropriate use or URL's. We only provide a small subset of rules, as examples of the many rules that can be used there.

3.4.1 Binary file format

The rules that have been selected here are about the content of the file, and in particular the type of the resources included in the file. Many applications, and in particular games, use resource files to store application information, and in particular graphical resources. Requiring them to use only standard formats allows a reviewer to look at the graphics used by the application. Similarly, attributes included in the descriptor may be restricted to standard attributes.

Note that, when using application-specific properties, it is possible to go beyond these simple rules, and to describe the fact that an application only uses resource files of a given type.

General rules

G-1	All the resource types in the JAR file are of standard types.

Rationale: The objective is to check that the multimedia contents embedded in the application can be viewed to ensure that they are appropriate.
Implementation: Explore the structure of the Jar file (a zip archive). There is no analysis of the code required, but it may be necessary to check that each file complies with the format specification.

G-2	The descriptor and manifest files only define standard attributes.

Rationale: This rule ensures that there are no data in the JAD file. Unfortunately it may also hinder the portability of applications, as some developers use the manifest to put network/operator specific attributes (eg. name of payment platform). Note that the JAD file is never signed. Only attributes in the JAR can be signed.
Implementation: One must parse the Manifest and the JAD, check their consistency and verify that there are only standard attributes.

G-3	The resources are used according to their type.

Rationale: This rule ensures that there is no hidden contents in files attached as resource to the application.
Implementation: To check this rule, we must perform the following atomic checks:

- Verify that rule rule G-1 is enforced.
- All accesses to resources must be at least partially determined (For each use of getResourceAsStream, we must be able to check the suffix).
- Then we must check that the type of the stream opened corresponds to the kind of operations performed on it (for example, an image creation method for a PNG file (Image.createImage, a sound player for an AMR file Manager.createPlayer(stream,type) with type of the right kind, regular string reading for a TXT file, etc.).

Custom rules

C-1	All the resource types in the JAR file are in a list.

Rationale: This is a specialisation of rule G-1 to a more constrained set of allowed resources.
Implementation: Same as rule G-1 but the list of authorized types is no more fixed, instead it is extracted from the contract.

3.4.2 Class loading

It is possible to make a MIDlet that uses conditionally available system APIs more portable by checking the platform name (a system property) and depending on the result requesting available classes with the following idiom.

```
ItfType v = (ItfType) Class.forname(classname).getInstance();
```

This usage of the Class.forName method may interfere with other analyses as it makes static devirtualisation more complex. Here are potential framework rules in ascending order of complexity.

General rules

G-4	Use of Class.forName is forbidden

Rationale: (Too) strict restriction for developers.
Implementation: It is easy to check the use of a method with a very simple implementation of devirtualisation.

G-5	Argument of Class.forName is constant.

Rationale: This is the pragmatic rule and a reasonable compromise for developers and evaluators.
Implementation: Same as previous rule to identify uses of Class.forName. Then a local analysis is required on the calls found to find the value of the string argument. The information found must then be propagated in the class analysis algorithm.

G-6	Argument of Class.forName is determined.

Rationale: A very relaxed rule for developers, but harder to check. Is it really useful ?
Implementation: This choice can make the interactions between the analysis of the arguments and the class analysis very complex as one needs the other.

3.4.3 Permissions

Permissions are central to MIDP's security architecture. A MIDlet must request a permission in order to use a protected API (the permissions requested by a MIDlet are listed in the MIDlet's manifest).

In terms of analysis, permissions are interesting, because they fix some limits to the features that an application may use, without having to analyze the source code.

General rules

G-7	The application requests all the permissions it may need.

Rationale: This rule ensures that an application that will never be allowed to perform an action it will try to perform during its execution, will not be installed.
Implementation: Permissions are written in the JAD and the manifest files. Each permission is associated to a set of protected methods. The checker must verify that for each protected method used in the JAR, the corresponding permission is declared. For most permissions, it is just necessary to check whether a given method is used or not (devirtualisation problem). But for some methods (Connector.open especially), the permission used depends on some of the arguments used. These arguments must be determined (for Connector.open(url), the scheme of the URL must be available.).

G-8	The application does not need any permission.

Rationale: This rule ensures that the application runs in the closed sandbox model and so is secure.
Implementation: No protected method is used.

G-9	The application does not request any optional permission.

Rationale: An optional permission is a permission that is not mandatory for the application (it could run without it and it should gracefully handle the security exception).
Implementation: Permissions are written in the JAD and the manifest files

G-10	Each use of a method associated to an optionally requested permission is guarded by a Security exception handler.

Rationale: An optional permission is a permission that is not mandatory for the application (it could run without it and it should gracefully handle the security exception). The goal of this rule is to ensure that there is an alternative to their use in the code of the application.
Implementation: The set of optionally requested permissions can be retrieved from the JAD file and the set of associated protected methods can be deduced from it. The calls to the protected methods associated to the optionally requested permissions must be identified. An exception analysis must be performed on the context of the calls to the protected methods.

G-11	The application uses all the permissions it requests.

Rationale: The objective is here to ensure that an application is not granted a permission that could be used by an attacker exploiting a platform bug.
Implementation: This is the same as rule G-7 but it may be difficult to ensure the liveness of the calls.

Custom rules

C-2	The application requests exactly a list of permissions.

Rationale: Strengthening of the previous rule. It ensures that the application uses the critical APIs and only these. The list could be defined in a protection profile for a class of applications.
Implementation: This rule is the conjunction of the previous rule with framework specific rules stating that exactly the set of requested permissions is used.

3.4.4 System properties and APIs

There are a few APIs that give access to system resources and properties. These APIs are quite sensitive in the sense that they give access to system information, which could be used in an attack.

General rules

G-12	The application only accesses constant properties.

Rationale: This is the most important: by requiring all properties accessed to be defined by a constant string, it allows other rules to be defined, restricting the use of some system properties.
Implementation: The checker must find the use of System.getProperty and find how the argument is built. This must be a direct constant string. There is no good reason to extend it to a defined string.

G-13	The application only accesses MSA system properties.

Rationale: The forthcoming MSA JSR specifies which JSR a phone platform should support and each JSR defines its own set of permissions. A property not defined in MSA can be rightfully considered as a proprietary system property whose use can raise unexpected security issue (ex: the IMEI number that uniquely identifies the phone handset (privacy risk) are accessible as system properties on some phones).
Implementation: Check that the values obtained during the check of the previous property are in a given fixed list extracted from the MSA specification.

G-14	The application does not access a given system property.

Rationale: An operator can consider that a known and standard property is dangerous in a given context.
Implementation: Same as previous one.

G-15	The application does not use the System.currentTimeMillis method to time sensitive methods.

Rationale: A social engineering attack is to wait until the user disables security checks or answers security questions too quickly to execute a forbidden action.
Implementation: As a first approximation, two calls to System.currentTimeMillis are used to compute a duration when an expression uses the two values returned by two different calls (we will not check that it is a difference as there is no reason to compute anything else). If there is a continuous execution path with no user interaction, (ie an execution trace that does not span over different event callbacks) that contains a call to a protected API.

Custom rules

C-3	The application only accesses system properties from a list.

Rationale: An operator may define for a given applicative field the set of authorized properties.
Implementation: Same as rule G-13.

3.4.5 Threads

Threads are heavily used in MIDP applications, because they represent an interesting way to perform long operations on a separate thread, while keeping the main thread (typically used for the GUI) active and unblocked.

There have been security problems on MIDP-enabled mobile phones that exploit race conditions between threads: [DSTZ06] reports on a security problem where a race condition between two threads accessing the phone display can lure the user into giving permission to send SMS messages (see also [Gro03]).

It is difficult to forbid applications in general from using threads, so limits on the number of threads used by an application are application-specific. Framework-level properties are more concerned with the proper management of the threads.

General rules

G-16	The application's main thread is never blocked.

Rationale: This is an ergonomy rule but a responsive UI is an important requirement. Responsiveness of the UI has also an impact on the way the user reacts to security screens.
Implementation: Methods identified as blocking IO are never called in UI callbacks. This is a call-graph exploration.

G-17	The application is thread-safe (synchronisation is appropriately used).

Rationale: The behaviour should be coherent. It is an important rule if we want to check more complex rules depending on the control flow of multi-threaded MIDlets.
Implementation: Variables and fields used by each thread must be isolated. When multiple threads may access (write) concurrently a given variable, their use must be protected.

G-18	All the calls to protected APIs are done in a single thread or there is proper synchronisation that ensures they are always done in the same order.

Rationale: The user must understand what is going on and why he is asked to accept a call to a protected API. The only way to achieve this is that the MIDlet has the same behaviour between different invocations.
Implementation: Checking that all calls to protected APIs are performed in a given thread is rather easy but the second version of the rule is much more open.

Custom rules

C-4	The application uses a single thread.

Rationale: This is a very strict rule. It goes against the recommended practice for blocking IOs.
Implementation: There is no call to the java.lang.Thread class

C-5	The application only uses a given number of threads.

Rationale: JTWI specifies a number of threads (10) that an implementation should support. Checking that the number of threads is bounded improves the portability of the application and also avoids unexpected handset crash when the limit is reached.
Implementation: It may be hard to check if there are a lot of temporary threads with no constraints on their termination. If the programmer uses a finite number of worker threads declared "statically" this may be easier. Using long lived worker threads is a recommended practice.

C-6	Calls to protected API are done following a given automaton.

Rationale: The user must understand what is going on and why he is asked to accept a call to a protected API. The only way to achieve this is that the MIDlet has the same behaviour between different invocations. This is a refined but may be more implementable version of rule G-18.
Implementation: Model checking on the control flow graph. The use of multi-threading makes it more difficult if protected APIs are used in different threads, but even then the property can still be verified by logic-based verification and hence certified by PCC. Indeed, automata are a natural formalism to express security requirements, as discussed in Chapter 5.

The idea of specifying requirements using a automaton (or finite state machine), as in

rule C-6 above, is already used in the Unified Testing Criteria (UTC2.1) [Ini06]. The UTC, developed as part of the Java Verified^{TM} Program (http://www.javaverified.com), specify testing requirements for Java MIDP applications. It covers many aspects, most of which are not security-related at all; e.g., whether the Java Powered logo is displayed. The UTC require that the developer provides a flow diagram of the application. This is currently only meant as an aid to the tester, but it could treated as (the basis for) the automaton in rule rule C-6.

3.4.6 URL's

URL's are very commonly used in the MIDP framework as a way to locate resources, local or remote. There are therefore many requirements that apply to URL's, and these requirements are scattered through all API's and features.

There are however some general requirements about URL's regarding how it is possible to reason about them. In the simplest case (for the analysis), an operator may choose to impose constant URL's. This is of course not applicable to all applications; it is acceptable for applications that only use URL's to identify local resources or fixed remote resources, but it is not compatible with a network application, for instance if it uses HTTP to "browse" a site.

In these cases, there are several ways to go:

Only allow determined URL's. This gives more leeway to the developer, but it still does not allow computed URL's, such as those that are built from information retrieved from the network. In addition, the exact definition of determined depends on the analysis algorithms, which means that this property may be difficult to grasp for developers.

Only allow URL's with determined protocol (resp. host) component. In that case, the analysis can check which protocol is used, but it may not be able to extract any other information. In other cases, the requirement may be stricter, and it is possible to require the host to be determined as well. In addition to this basic requirement (which allows some sorting of the URL's), it is possible to define specific requirements for each protocol.

URLs as specified in RFC 2396 take the general form:

–scheme″:[–target″][–parms″]

A classical example of –target″ is :

–host″:–port″/–local path″

A parameter is a series of equals ;x=y. Depending on the kind of property to achieve, the check may consider any of the parts of the URL.

General rules

G-19	The application only uses constant URL's.

Rationale: See above. This is the strictest rule. This rule forbids partially computed URL's.
Implementation: Calls to methods using URL's must be identified. Then the argument supplied in the call is checked (simple local analysis).

Requirement	Code
Constant URL	conn = Connector.open("http://www.somehost.com") ;
Determined URL	host = "www.somehost.com" ; conn = Connector.open("http://" + host) ;
Determined protocol	dest = userInput + "/data.jsp" ; conn = Connector.open("http://" + dest) ;
Determined protocol	if (url.startsWith("http://")) conn = Connector.open(url) ;

Table 3.1: Examples of URL's

G-20	The application only uses determined URL's.

Rationale: See above. This is the relaxed rule. It may be mixed with the previous one requiring that the URL is fixed for some protocols and allowing computed URL's for others.
Implementation: Calls to methods using URL's must be identified. Then the argument supplied in the call is checked (more complex and potentially global analysis).

G-21	The application only uses URL's with a determined protocol component.

Rationale: This ensures that only a given protocol is used. Usually this rule is used to forbid protocols that lead to costs for the user or that are not available on the operator network. There is no attempt to check the target host.
Implementation: The expression computing the URL must be statically determined and the prefix up to the first colon must be a determined string equal to the protocol.

Examples

We provide a few examples of URL's in table 3.1. In these very simple examples, we see the difference between a constant URL, a determined URL, and a determined protocol. In that last case, we first consider that the host is determined by some kind of unpredictable user input, but that the protocol is provided as a constant; we then consider that the value of the protocol is verified before the URL is actually used.

These examples are described in greater details in section 3.15.1.

3.5 Wireless messaging rules

Wireless messaging is commonly used by applications, in particular because it is a way to generate revenue. By sending a SMS to a premium number, an application provider can get some revenue for its application users. The medium is also interesting because it is under tight control of the operators.

The MIDP framework includes many protections against abuse of wireless messaging. In particular, for all applications except those signed by an operator, an explicit end-user confirmation is required before any message is sent; this confirmation includes the destination number, which allows end users to check the cost associated with the message. However,

social engineering attacks remain possible, in which the attacker somehow convinces the end user to accept the sending of several premium messages.

3.5.1 Destination numbers and ports

Destination numbers are very important. It is often requested that messages be sent only to determined numbers, matching some pattern (some operators forbid the use of premium numbers, some mandate them and add some control). A common requirement also is that messages shall not be sent in a loop.

Naturally, this is an area in which it is interesting to customise properties for an application, since this allows a developer to make interesting claims, such as claiming that a single destination number is used.

General rules

G-22	The application sends only messages to determined ports.

Rationale: Ensures that the application will not try to access specific system ports (this should be enforced by a compliant MIDP implementation).
Implementation: Check the port number in the target of the URL. The initial segment of the URL until the port number must be determined.

G-23	The application sends only messages to determined numbers.

Rationale: Ensures that the phone number used by the application are known.
Implementation: Check the target of the URL

G-24	The application does not send MMS messages

Rationale: Only accessible in WMA2.0, MMS are more expensive than SMS and more complex to handle.
Implementation: Check the shape of the URL.

G-25	The application sends messages to a number matching a given pattern.

Rationale: Refined version of previous rule usually used to check the cost of the SMS sent (In some countries, the cost of an SMS depends on the first digit and the length of the phone number). Note that this property is general although it is parameterized, because it is often required by operators for all their applications.
Implementation: Same as above.

G-26	The application sends only MMS messages to addresses of a given kind (phone numbers for example or e-mail address or IPV4 address).

Rationale: An operator rule if only one kind of addressing scheme is considered as authorized.
Implementation: Check the shape of MMS URL with the right regular expression.

G-27	The application does not send messages in a loop.

Rationale: Each message is charged. Sending in a loop will create a prohibitive cost for the user. There may be exceptions to this rule (when a user wants to use a MIDlet to send an SMS to several correspondents for example taken in the address book).
Implementation: This rule corresponds to the identification of loops, and/or in the counting of some API calls.

Custom rules

C-7	The application only sends messages to a given number.

Rationale: This rule checks that an application only use the SMS service platform specified in the contract with the operator.
Implementation: The start of the URL must be sms:{tel number}

3.5.2 Receive Connections

More complex applications may need to receive messages as well. Point-to-point (SMS) and broadcast (CBS) messages are available in MIDP. An application may either register statically to receive messages (in which case the connection is described in the descriptor), or it may register dynamically, through an API. It is quite common to require applications to register statically, in order to increase the control on messaging.

General rules

G-28	The application does not receive messages.

Rationale: The application may be allowed to send messages but not to receive messages. This is a strict rule especially for SMS premium services that usually answer to a query with an answer SMS.
Implementation: There is no SMS connection opened on a sms://:port URL. One can also check if the program tries to use MessageConnection.receive but if the URL is not of the above shape, it will not succeed (Connection is said to be in client mode).

G-29	The application checks the origin of incoming SMS.

Rationale: The application can receive an SMS coming from any platform or even from another phone. It may be safer to check the origin of an answer SMS.
Implementation: The application accesses the origin field with Message.getAddress and compares it with a constant value. If the value is not the expected value, then the message object should be discarded. Checking this rule relies on a good understanding of the control flow. A programming rule could impose that all these operations are done in a single method to simplify the verification.

G-30	a Message listener callback does not contain any blocking method call.

Rationale: This is more a dependability/style rule imposed by JSR118
Implementation: Check the call graph of each MessageListener.notifyIncomingMessage method

G-31	The application only registers to receive SMS (resp. CBS) messages through the push registry mechanism.

Rationale: The application can only be waken up again but will not perform an active wait. This rule improves the battery usage.
Implementation: Reception of messages is handled by a separate MIDlet and is correctly registered in the PushRegistry mechanism.

Custom rules

C-8	The application only receives SMS messages on a specific port.

Rationale: This ensures that there will be no collision with other applications. Ports could be assigned for SMS as they are for IP services (there is already a list of reserved port numbers).
Implementation: Check the shape of the URL for messageConnections. There should be only one such URL used.

3.5.3 SMS message types

General rules

G-32	The length of an SMS message sent does not exceed the payload of a single SMS message.

Rationale: SMS messages can be split in shorter messages and then reassembled but the cost paid by the sender is proportional to the number of fragments.
Implementation: Resource usage rule: one must compute the maximum length of a string.

G-33	The application sends only text (or binary) messages.

Rationale: Such a rule ensures that the message created is of the right kind.
Implementation: The message objects created are of the right kind.

G-34	Multipart messages are only used for MMS connections.

Rationale: This is a code correctness rule.
Implementation: Check for each MessageConnection the kind of messages sent on it and the URL used to create it.

G-35	Some headers of MMS are not accessed

Rationale: An operator may restrict the headers used (for example to avoid a denial of service attack with a fixed X-Mms-Delivery-Time header)
Implementation: Check for each MessageConnection the kind of message sent on it and the URL used to create it.

G-36	The parts of an MMS message are correctly defined and of a given type

Rationale: For example an operator could restrict MMS messages to images (from the camera ?) and disallow jar codes (protection against viruses).
Implementation: The MessagePart object must be created with the right mime type and the contents must be identified as of the right kind (eg. an image capture with the right kind). This rule is an umbrella for multiple simple rules for each mime type. The mime type should be coded as a constant.

Custom rules

C-9	The application sends no more than a number messages in each session.

Rationale: Check that the application will not send too many messages.
Implementation: Resource counting problem with an upper bound.

C-10	The cost of messages sent during a session is less than a number euros.

Rationale: Refined version of the previous rule. There may be different costs associated to different phone number regular expressions.
Implementation: Resource counting problem with an upper bound.

C-11	The application sends at least a number messages in each session

Rationale: Rule to check that a service is really paid by the user. Can be useful if part of the money received for the service is given back to a third party.
Implementation: Resource counting problem with an lower bound. But taken as a progress property, such a rule cannot be true (a user can (should be able to) stop a MIDlet.

3.6 Network access rules

Many applications require some kind of network access. In fact, a large number of MIDP applications are used as smart clients for network applications. In such cases, networking is crucial for the applications.

However, operators have been reluctant to open the use of network connections, because there are many risks associated to it. First, networking has a cost, which is difficult to control, since operators often bill users on the volume transmitted over a GPRS connection. In addition, a network connection can easily be transformed into a communication channel for any kind of information, authorized or not. The use of network connections is therefore strictly controlled by most operators.

3.6.1 Categories of Connections

MIDP supports many categories of network connections, ranging from the classical HTTP and HTTPS connections to low-level socket and datagram connections, and even to server connections.

The properties that apply to categories of connections mostly consist of constraints on the protocol part of the URL.

General rules

G-37	The MIDlet only establishes HTTP connections.

Rationale: An operator may restrict long range connections to HTTP
Implementation: The scheme of URL's used to establish connections is http

G-38	The MIDlet only establishes secure connections.

Rationale: Such a rule could be imposed for sensitive applications, such as e-commerce MIDlets.
Implementation: The scheme of URL's used to establish connections is https or ssl

G-39	The MIDlet only establishes HTTP or HTTPS connections.

Rationale: Same as above but relaxed to authorise both kinds of connections
Implementation: Same as above

G-40	Only POST request are done to send queries.

Rationale: Classical security rule for web application. This should be used to ensure that the URL will not contain user supplied data.
Implementation: Check how the request is done and the target URL.

3.6.2 Destination of the Connections

In some cases, it may be interesting to know the destination of the connection, i.e., the host part of the URL. Some operators may put some requirements on the host or domain and, more importantly, some application providers may be interested to demonstrate that they only connect to a fixed host or domain.

General rules

G-41	The application connects only to its origin host.

Rationale: This rule is inherited from classical security policy for applets. There is an issue about the definition of the applet's origin. One possibility is to consider the URL's provided in the JAD and manifest for this purpose.
Implementation: Check that connections are opened on a given URL with the right target part

G-42	The application connects only to its origin domain.

Rationale: Relaxed version of the previous rule
Implementation: Same as above, the check is done with a regular expression.

G-43	The application does not use local connections (like 127.0.0.1) .

Rationale: Local connections are usually used to talk to native services and this is used on some handsets with open OS (eg. Symbian) to defeat the whole purpose of the sandbox security model.
Implementation: Check the shape of URL. The target should not be the localhost.

Custom rules

C-12	The MIDlet only establishes HTTP connections to a specific URL or a specific domain.

Rationale: Variation on rules rule G-41 and rule G-42. The name is specified in a contract.
Implementation: Same as for the general rules.

C-13	The MIDlet should check the certificate used by an HTTPS connection and the certificate in use is from a given issuer/subject .

Rationale: A secure application should check that the expected certificate and not another one available on the phone has been used for the connection. The absence of such checks is a common source of security problems, as discussed in [HlV05]; in fact, the SSH MIDlet considered in Section 4.4 was found to lack a comparable check.
Implementation: It may be hard to check that the verification is effective. Moreover, the check must prove that the connection object will not be used if the check has not succeeded. The correct sequence is a connection is created, a call to HttpsConnection.getSecurityInfo() is done, then to SecurityInfo.getServerCertificate(), then to Certificate.getIssuer(). The result must be compared to a constant string and only if the comparison succeeds, the HttpsConnection object may be used.

3.7 Push registry rules

The push registry maintains a list of inbound connections. These connections are monitored by an application while it runs, and by the AMS (Application Management System) otherwise. It allows in particular an application to be launched automatically when specific events occur.

The main events that are in the push registry are alarm events and network connections. It is possible to register connections statically (by a declaration in the JAD file), or dynamically (through an API).

In terms of security, the push registry opens many possibilities for attackers, although the triggering of an application is normally subject to an action from the user.

3.7.1 Push Registry usage

General rules

G-44	The application registers only to alarms.

Rationale: The application should not wait for an external network event but can be waken up periodically
Implementation: PushRegistry.registerConnection is not used and there is no connection registered in the JAD file.

G-45	The application registers only to network connections.

Rationale: An application that should react on a network event.
Implementation: PushRegistry.registerAlarm is not used.

G-46	The application makes only static registers.

Rationale: This ensures that the set of potential incoming connections is completely controlled.
Implementation: The method for registering push events PushRegistry.registerConnection are not used

G-47	The application makes only dynamic registers.

Rationale: See also next rule.
Implementation: There are no registered connections in the JAD file.

G-48	The application provides a way to unregister each dynamic register.

Rationale: Otherwise the user may have to uninstall a MIDlet he has paid for to temporary stop the registration. A stricter rule could be the existence of a MIDlet in the suite to unregister connections and of another one to install them.
Implementation: Check for the existence of pair registerConnection and unregisterConnection. The main challenge may here be to check the liveness of the call to unregisterConnection, i.e., to verify that the the unregister operation can actually be performed as a result of an end-user operation. This may require an actual interactive test.

3.7.2 Push Registry usage with alarms

G-49	The date for waking up the MIDlet is not a fixed date but is dependent on the current time or is supplied by the user.

Rationale: This rule protects the operator against a scheduled attack to create a deny of service against a platform of the network.
Implementation: The argument of registerAlarm should be either computed from a DateField object (supplied by the user) or of the form System.getCurrentTimeMillis() + x. It is also necessary to check that the date field is really used by the user and if initialized, is initialized to the current date.

G-50	The delay between two periodic invocations of the MIDlet is at least T.

Rationale: The periodic use of a MIDlet can quickly exhaust the battery of the phones. It is necessary to control the resource usage of a periodic MIDlet.
Implementation: If the argument of registerAlarm is System.getCurrentTimeMillis() + x then x is greater than a given value. This can be a reasonable use of constraint propagation or abstract interpretation on intervals. A stronger requirement, but simpler to check, is that x is either a constant or the sum of a constant and an arbitrary unsigned integer.

3.7.3 Push Registry usage with connections

General rules

G-51	The application registers only to a type of connection (SMS, MMS, ...).

Rationale: This rule ensures that the application will not try to take over the control of all incoming connections.
Implementation: Check all the URLs (dynamic and static registers). There should be only one kind of prefix (scheme and host).

G-52	The application does not register to socket local port.

Rationale: Local socket and ports are a way to interface with the native application. Without an in-depth device specific analysis this can lead to unexpected security issues (for example, it could bypass the native Java security policy by accessing the primitives implementing protected methods at a lower level).
Implementation: Control the shape of URL. Requires the scheme and then the host.

Custom rules

G-53	The application registers only to a list of port.

Rationale: This rule implies that the application only registers for some services.
Implementation: Check all the URLs (dynamic and static registries). The port parts must be available and comply with the rule.

G-54	The application registers only with a specific filter.

Rationale: The filter specifies the host allowed to connect to the application.
Implementation: Control the shape of URL filter. All filters must be the same string.

G-55	The application registers only to a determined host — domain.

Rationale: Restrict the incoming connection to a given strict pattern. It further specialises the previous rule.
Implementation: Control the shape of URL filter. Requires the host. A regular expression may be used.

3.8 Local connections rules

MIDP phones are often fitted with local connectivity. The most common media are Bluetooth, serial links over infrared or cable. MIDlets are able to exchange data through these connections, but operators may restrict their usage in their security policies.

3.8.1 General properties

General rules

G-56	The application does not use Bluetooth or IrDA connections.

Rationale: This rule disallows local connections that could be used to transfer private data out of the phone or to transfer malicious applications (worms) to other phones.
Implementation: Check the scheme of URLs.

3.8.2 Bluetooth connections

Custom rules

C-14	Limit discovery to a set of UUID (identifier for a bluetooth service) for the service and names for the devices.

Rationale: Limiting discovery enables a stricter control of the effect of environment on the behaviour of the application.
Implementation: Arguments in the call of method DiscoveryAgent.searchServices

C-15	The MIDlets acts as a bluetooth client (resp. as a bluetooth server).

Rationale: Limiting to a client role ensures that the device does not enter in connectable mode. On the other hand, in server mode, the initiative of the connection is not on the MIDlet side but on the other device.
Implementation: This is specified when the connection is opened as different schemes are used in the URL. Moreover a server should/could update its service record in the BCC. A client should rather use a discovery request with DiscoveryAgent.searchServices to create its URL.

C-16	Impose security requirements (encryption, authentication) for server or client connections.

Rationale: These requirements ensure that only trusted client can connect to a service offered by a MIDlet.
Implementation: Those parameters are optional arguments of the URL used to create the connection (authenticate=true for authentication, authorize=false to request a proper verification of the trust level of the host and encrypt=true to request encryption). They can conflict with the device settings/capabilities. A further check that goes beyond classical URL verification is to make sure that the exception BluetoothConnectionException is properly handled. There are two main cases:

- The URL is hard-written in the code: the check must just ensure that the URL is determined and check the presence of the desired argument.
- The URL is computed by the library as a response to a query of services. There are several objects and methods used.
  - if the URL is built by a call to ServiceRecord.getConnectionURL, then the ServiceRecord object must be obtained from a method of the library. Usually it is an argument to DiscoveryListener.serviceDiscovered callback which has been used with DiscoveryAgent.searchServices. The security parameters should be specified as a first parameter in the call to the getConnectionURL method.
  - it can also be obtained directly with a call to DiscoveryAgent.selectServices.

3.9 Platform-dependent requests

The MIDP API includes a specific API for making platform-dependent requests, called MIDLet.platformRequest. This API can be used in particular to trigger external programs such as the phone's default browser or the phone dialer.

One of the issues with this API is that, even if some of its uses (for instance, the establishment of phone calls) are well standardized, there may be other proprietary uses, and it can be used as a hidden communication channel.

3.9.1 General properties

General rules

G-57	The application does not use platform-dependent requests.

Rationale: MIDLet.platformRequest can be used for many device specific functionalities that raise unexpected security issues.
Implementation: Forbid calls to MIDlet.platformRequest

G-58	The application only uses platform-dependent requests for phone calls.

Rationale: This is the only standard use of platformRequest (with WAP browser usage that could also be added).
Implementation: Check the shape of URLs used with MIDlet.platformRequest

3.9.2 Phone calls

General rules

G-59	The application initiates only calls to determined numbers.

Rationale: Ensures that the phone number used are statically known
Implementation: Check the URL used with platformRequest

G-60	The application does not initiate calls to international numbers.

Rationale: To avoid unexpected costs for the end-user
Implementation: Check the shape of the phone number used. If + appears it must be followed by the right country (issues when abroad).

G-61	The application initiates only calls to premium numbers.

Rationale: For service front-ends. This usually implies further restrictions on the kind of numbers.
Implementation: Check the phone number part of the URL

G-62	The application does not initiate calls to premium numbers.

Rationale: To forbid over-charged calls. Depends on the applicative field of the application.
Implementation: Syntactic check of the phone number part of the URL.

Custom rules

C-17	The application only initiates calls to a given number.

Rationale: Specialisation of previous rules to a given number defined in the contract with the MIDlet provider.
Implementation: Syntactic check of the phone number part of the URL.

3.10 Persistent storage

MIDlets are standard Java applications, and all their objects are allocated and stored solely in RAM. By default, they do not have a persistent state. MIDlets therefore have the possibility to store data persistently, in order to manage a persistent state across sessions.

MIDP defines a specific mechanism, record stores. A record store is similar to a file, but its objective is to be simpler than a full-fledged file system, in order to be available over all categories of phones, including the simplest ones. On phones that include a standard file system, MIDlets may also have the opportunity to access the file system.

On high-end phones and PDA with a standard high-level operating system (e.g. Linux, Symbian or Windows Mobile), there is a regular file system. The FileConnection API makes it available to Java applications. This API is part of the General Connection Framework (GCF) of CLDC and a FileConnection object is created with Connector.open("file:¡path¿"). Protections and rights on those files are out of the scope of MIDP.

Some persistent storage mechanisms are also available with some API's. For instance, landmark stores are used by the location API. These API-specific mechanisms are covered in the appropriate API's.

3.10.1 Global properties

General rules

G-63	The application only uses record stores for persistent storage.

Rationale: This is the only kind of persistent storage tightly tied to the application (it will be removed with the application) and with a well understood scope (with potentially no sharing).
Implementation: FileConnections and LandmarkStores are not allowed. This is a check of the URL used for the first and on the class used for the second.

3.10.2 Record stores

General rules

G-64	The MIDlet uses at most a number of record stores at any time.

Rationale: Control the amount of active resources in use (more a portability issue).
Implementation: Resource usage counting on openRecordStore and closeRecordStore.

G-65	The MIDlet stores at most a number kbytes of information in a record store.

Rationale: Control the amount of memory used for each store. Probably a necessary check for the next rule.
Implementation: An easy best way is to enforce this is to impose a dynamic check before each modification.

if (n + recStore.getSize() ¡ LIMIT) recStore.addRecord(record,offset,n) else throw (new RecordStoreFullException());

More relaxed rules, implying more in-depth, reasoning could be imposed.
An obvious property to certify by means of PCC then is the absence of RecordStoreFullExceptions, or the absence of such exceptions under certain conditions. This would remove the need for the dynamic check, or allow it to be replaced with a check of some condition at an earlier program point. If this is possible, it is of course be preferable to a dynamic check: the user would rather know beforehand if an operation is going to succeed instead of having a dynamic check on memory usage fail at some stage during the operation.

G-66	The MIDlet stores at most a number kbytes of information in all their own record stores.

Rationale: Control the amount of persistent memory used by the application.
Implementation: Quite complex to implement. A proof can be obtained by adding individual constraints on each record store.

G-67	The MIDlet only uses private record stores.

Rationale: This ensures that the MIDlet suite will not share data with other applications (for example to spy the habits of the user and draw a consumer profile)
Implementation: The MIDlet creates only private record stores (check the boolean given as authentication mode to RecordStore.openRecordStore) and only access local record stores (it does not use the method RecordStore.openRecordStore(String name, String vendorName, String midletSuite).

G-68	The MIDlet only accesses record stores from their own vendor.

Rationale: This is a relaxed version of the previous rule where a MIDlet vendor may share information between his own different MIDlet suites.
Implementation: The vendor is an attribute from the JAD (and manifest) file. Its contents must be checked. Then the MIDlet must only use RecordStore.openRecordStore(String name,String vendorName,String midletSuite) with a constant string as vendor name or using the result of a call to midlet.getAppProperty with midlet as the current MIDlet and the correct attribute name (MIDlet-Vendor).

3.10.3 FileConnection API

General rules

G-69	The application does not use the FileConnection.delete() function.

Rationale: To preserve the integrity of the File system
Implementation: Check the use of FileConnection.delete()

G-70	The application does not use the Fileconnection.create() function.

Rationale: To preserve the integrity of the File system.
Implementation: Same as above

G-71	The application does not modify data.

Rationale: To preserve the integrity of files.
Implementation: Forbid the use of FileConnection.createOutputStream()

G-72	The files used by the application are in total at most a number kilobytes.

Rationale: To impose a resource usage policy and a fair sharing of the persistent storage.
Implementation: This requires that a specified design pattern is used. Aspect-oriented programming that automatically modifies writes to files and static analysis to limit the number of runtime checks can be an interesting alternative.

Custom rules

C-18	The application only uses files whose name matches a given pattern.

Rationale: This can be used to ensure that files are specific to the application and that the MIDlet does not jeopardise the integrity of files owned by other applications.
Implementation: This is a derived URL rule with a constraint on the target. The rule may require that a given attribute (the MIDlet name) is used in the file name. The checks are made difficult by the FileConnection.setFileConnection, which allows a MIDlet to open a file specified by a relative file name.

3.11 Personal Information Management (PIM) API

Mobile phones are often used as personal information managers, storing detailed contact information, as well as calendar appointments and to-do lists. The PIM API makes this information available to the MIDlets that need to use it.

Note that a MIDlet does not need to access the user's phone book in order to let a user choose the destination of a SMS or phone call. One of the options for data entry allows the user to return a phone number by selecting it from the phone book. The MIDlet then gets access to the phone number, but not to any other information.

Access to the user's personal information can be tightly controlled, because the PIM API requires the MIDlet to declare which type of information it accesses (contacts, events, to-do list), and to declare how it will use it (read only, write only, read/write). However, some security issues remain about the use of the PIM information, in particular related to the way in which the information may or not be disclosed to the outside.

3.11.1 Restricting access

General rules

G-73	The application does not publish PIM information.

Rationale: The PIM information is private, and it should not be disclosed (on Internet, through a SMS, or in any other way). The rule is rather strong, and it may be inappropriate for most applications.
Implementation: The objective would here be to follow the flow of PIM data. The main difficulty is to determine exactly what constitutes a disclosure. Then, the content used by the API's that may trigger this disclosure need to be monitored.

Custom rules

C-19	The application accesses only the contacts — agenda — to-do list.

Rationale: A specific kind of application should only access a given kind of PIM resource.
Implementation: First argument to every call to PIM.openPIMList is either CONTACT_LIST or EVENT_LIST or TODO_LIST depending on the version of the rule.

C-20	The application only accesses a list of fields.

Rationale: Restrict the accessed fields to what is needed by the application.
Implementation: The first argument of every call to (Contact—Event—ToDo).getXXX or (Contact—Event—ToDo).setXXX or PIMItem.removeValue must be a determined integer belonging to the authorized list (the list depends on the class used).

C-21	The application only discloses a list of fields.

Rationale: Restrict the fields to what the application needs to disclose to the outside world. For instance, a mapping tool may disclose an address, but not a name or a phone number.
Implementation: This is a combination of the previously described rules, in which the content obtained with (Contact—Event—ToDo).getXXX for some values of the first argument needs to be tracked throughout its usage.

3.11.2 Restricting operations

General rules

G-74	The application only reads personal information.

Rationale: Guarantees the integrity of the personal diary.
Implementation: Second argument to every call to PIM.openPIMList is READ_ONLY

G-75	The application does not modify existing records

Rationale: Guarantees that at least nothing will be destroyed in the personal diary (but it may be polluted).
Implementation: modification operations are only done on objects obtained with ContactList.createContact, EventList.createEvent or ToDoList.createToDo, or from a importContact or importEvent or importToDo. (Objects obtained from calls to PIM.fromSerialFormat should be imported in a list first.)

3.12 Multimedia connections

Another important use of MIDP is to be used in conjunction with multimedia players. The objective is of course not to develop players in Java, but to leverage the players that are already available on the phones. Most players are related to video or music, and the MIDlets that use them may be used to provide a user interface for the controls, and/or to implement DRM schemes.

Some players also offer the possibility to capture content. Typical capture methods include radio, photos, voice recording, and video recording. For privacy reasons, some operators may need to restrict the way in which MIDlets capture multimedia content.

Finally, note that media content is located through URL's. This means that the rules that apply to URL's, and in particular the network access rules, also apply to these URL's.

3.12.1 Media player locators

General rules

G-76	The application only creates players with local resources.

Rationale: Avoid network usage abuse with unexpectedly big resources.
Implementation: The inputStream used by the player is created by Class.getResourceAsStream or an URL with resource scheme (not fully compliant).

G-77	The application only creates players with determined resources of known types.

Rationale: To let the validator check that the contents are appropriate.
Implementation: Check of the URL used. It may require to follow the path between the media player creation method and the input stream creation method.

G-78	The application only loads images with determined resources of known types.

Rationale: To let the validator check that the contents are appropriate.
Implementation: For each image creation from an inputStream, the checker must find the corresponding method calls that create the stream and the URL argument that was used. This argument must be determined.

G-79	All images are loaded either from local resources or from HTTP/HTTPS connections with determined URL's and the file suffixes must be explicit.

Rationale: This is a stricter but more practical version of the previous rule.
Implementation: Same as above.

G-80	The application may only use dedicated record store to store media resources. There is at least one resource store per resource type.

Rationale: To let the validator check that the contents are appropriate and further control the use of the resources in the MIDlet.
Implementation: Mix checks of recordstore objects and multimedia resources to build up a more complex property.

G-81	The application only loads each image from the network once.

Rationale: This is to avoid network usage abuse.
Implementation: This is a resource usage checking problem and it can be quite complex. It relies on the fact that image names are determined (previous rule).

Custom rules

C-22	The application creates a player with HTTP scheme to a specific domain — host.

Rationale: Contractual rule with the developer to isolate the media server.
Implementation: Syntactic check of the URL's.

3.12.2 Capture

General rules

G-82	The application does not use capture.

Rationale: Captured sound/images may be used to steal private information of the end-user.
Implementation: Syntactic check of the URL's

G-83	The application uses only capture URLs for radio.

Rationale: This is the only kind of capture URL that has no privacy impact (except knowing who listens to what and when).
Implementation: Syntactic check of the URL's with a more complex pattern

3.13 Location API

The location API allows a MIDlet to get location information about the phone on which the application runs, and to manage a list of landmarks, i.e., of points of particular interest. The location API is independent of the method used to obtain the location information, which may be a coarse method based on a Cell ID, or a fine GPS solution.

This API causes obvious security issues, because it could allow a MIDlet to track its user and to forward this information to other users. Another security issue is the management of landmark stores (special files in which the description of landmarks are stored), which cannot be private to an application. Finally, there is a possible issue with the fact that some location methods are associated to a cost borne by the end-user.

General rules

G-84	The application does not allow location costs.

Rationale: To avoid unexpected costs for the end-user.
Implementation: A location provider is accessed through a call to the static method LocationProvider.getInstance(Criteria). There must be no control path from the Criteria object constructor to a getInstance call that does not contain a call to setCostAllowed(false) on this object (to change the default setting). Restrictions on thread use are necessary.

G-85	The application does not use a LandmarkStore.

Rationale: Security problems:

- The LandmarkSore is shared with all Java ME applications.
- The Landmarkstore is also not destroyed (erased) when the application (that has created this) is deleted from the terminal.

Implementation: The LandmarkStore class is not referenced.

3.14 Other frameworks

3.14.1 Introduction

The work on MOBIUS focuses on the MIDP framework, because it is the most widely available framework on mobile devices, and it exhibits properties that makes it a suitable candidate for advanced analysis techniques. However, there are many other frameworks that can be of interest for mobile systems. In particular, the following frameworks need to be mentioned:

Java Card.

Web services.

3.14.2 Java Card

The Java Card framework has been the focus of many research efforts in the recent years, for two main reasons: first, since smart cards are security tokens, the applications they host have to satisfy stringent security constraints; then, Java Card is a simple framework, with relatively small applications.

Because of these unusual characteristics, the Java Card framework presents two major advantages for research purposes: the simplicity of the framework and its applications makes it an interesting target for the most complex static analysis techniques (which often don't scale well); and it represents one of the most demanding environments in terms of security, often representing the future of the requirements for other environments.

In terms of simplicity, it has allowed researchers to make simplifying assumptions in the implementation of their analyses, and in some cases to be able to analyze realistic applications, even in a research environment:

- All applications run on a single thread, removing all needs for managing synchronisation.
- Object allocation is limited, so applications can be considered to manage their objects in a static manner (all objects are allocated at the creation of the application, or in its early lifecycle phases).
- Applications are small and simple. The depth of the execution stack is limited, and the overhead related to method declaration must also be limited. As a consequence, recursion is never used, and call trees are limited in size.

- The framework itself is small, with under 100 classes in Java Card 2.1, allowing researchers to model the entire API in their research.

This simplicity is slowly disappearing, as Java Card becomes more complex, and it will completely disappear with the release of the next major revision of the specification, which is due by the end of 2007. On the other hand, Java Card will keep its interest as a high-security framework, as its requirements will remain unusually high when compared to other frameworks.

We can here consider one particular example, which is the defence against physical attacks. As of today, smart cards are about the only computing systems that are regularly submitted to such physical attacks. However, it is quite likely that, as the value of assets stored in various kinds of devices increases, more devices will be subject to similar attacks in the future.

Physical attacks on smart cards can be grouped into two main categories:

- Side-channel attacks.
  These attacks consist in exploiting a signal measured on a side channel in order to gather some information about the running application, and in particular about its keys. The most usual such side channels are timing, power consumption, and electromagnetic radiation. Side-channel attacks have been widely used for many years, and in particular since the discovery of differential power analysis [KJJ99].

- Perturbation attacks.
  These attacks consist in perturbing the smart card chip through a wide variety of means, with objective of making the chip behave in an unexpected way that constitutes an attack. The most usual means used for perturbation attacks are power cut-off, power glitching, and laser attacks. These attacks are also called fault induction attacks; they are an important concern for the smart card community, in particular since optical fault induction has been improved [SA03].

The security infrastructure around smart cards is quite specific. There is a small number of applications, produced by a small number of developers, who are usually experienced. Before being deployed, applications often need to be certified by going through comprehensive and expensive security evaluations. Nevertheless, the study of Java Card and of smart card attacks can be relevant for MOBIUS, at least in two ways. First, countermeasures are difficult to design and to implement; static analysis algorithms may be used for the identification of potential vulnerabilities. Then, the problems experienced by Java Card applications today could become more mainstream as secure implementations of larger devices (such as mobile phones) become available in the future. In particular, the availability of more secure chips could make mobile devices interesting targets for physical attacks in the near future.

3.14.3 Web services

Web services, in their various instances, are a dominant framework in today's server software. These services provide an interesting security challenge, in the sense that they are widely accessible from Internet, which makes them relatively vulnerable to attacks. In addition, standard tools, and in particular server-side libraries, are widely distributed and reused, so the identification of a vulnerability on a Web site may be reproduced on many other sites if the error is in one of these libraries.

Because the economical threat is large, there has been a lot of research work about the threats encountered on Web servers in general [?]. In addition, the software that runs on Web servers is usually tightly managed by administrators, who are knowledgeable about security. Despite these various advantages, security vulnerabilities in Web services remain widespread.

Since server software is under control of the server's owner, static analysis can be used in favourable conditions (as the developer and provider can be expected to openly collaborate with the security evaluators). Some companies have built programs for analyzing Web applications, and even for automatically applying the appropriate countermeasures to identified vulnerabilities [Sof06]. Such applications of static analysis are relevant to MOBIUS, in particular in the context of Web services, for several reasons:

- Web applications are often built from aggregating several basic Web services, making the use of individual certificates for each Web service interesting.
- Since the software is under the user's tight control, it is possible to enforce strong organisational security policies, which make it easier to apply static analysis in a systematic way.

On the other hand, Web services should not be the primary focus of MOBIUS, because of a few differences with the software that runs on mobile devices. First, the vulnerabilities on server-side software are radically different from those on client-side software, so the results of MOBIUS can't directly be applied. Then, server software, even when written in Java, relies on different frameworks, such as Java EE (Enterprise Edition), which are far more complex than the dialects used on mobile devices, and therefore have different requirements in terms of API and framework modeling.

3.15 Implementing checks

3.15.1 Rule patterns

In the rest of the section, we have identified a number of common rule patterns. These rules patterns are instantiated many times above, and in some cases, they are mixed in a single property.

File content restriction

This pattern covers the simplest kind of rules, which are simple syntactic checks on the content of the binary files. Many rules are defined on the application's metadata, and in particular on the application's manifest file.

The typical rules are those that concern the content of the manifest file, such as the permissions and the application properties.

API usage

API usage rules are about forbidding the use of an API (interface, class, or method) altogether. The verification of these rules requires a parsing of the binary code, but it remains a purely syntactical check, without any requirement for complex analyses.

The analysis consists in determining whether or not the API is used. Since the rules are usually restrictions, any reference identified is considered a violation of the rule.

API usage restrictions (parameter analysis)

API usage restrictions are the most common rules. These rules are restrictions on the use of an API, in which only some values are allowed for some parameters. The most commonly mentioned method is the Connector.open method, which is used to open a new connection (network connection, SMS connection, file connection, ...).

The parameter to this method is a string that represents a URL. The various rules that apply on this parameter restrict the protocol part, the host part, and even the file name in some cases (in particular to check the file extension).

Sample rule Let's consider a simple rule, which mandates that all network connections used by a MIDlet be HTTP or HTTPS connections. The rule in fact applies to many API methods with URL arguments. For instance, the rule applies to:

- Connector.open(String name)
- Connector.open(String name, int mode)
- Connector.open(String name, int mode, boolean timeouts)
- Connector.openDataInputStream(String name)
- Connector.openDataOutputStream(String name)
- Connector.openInputStream(String name)
- Connector.openOutputStream(String name)
- Manager.createPlayer(String locator)

If we consider only the first case, there are many ways to define the rule. One of them is as follows:

- If the name argument starts with http:// or https://, then the property is satisfied.
- If the name argument starts with a non-network protocol, then the property is satisfied.
- If the name argument starts with socket:// or datagram://, then the property is violated.
- If the name argument starts with an unknown protocol, then the property is violated.
- If the protocol part of the name argument cannot be determined, then the property is undetermined.

Checking the rule These rules are of course more difficult to check than the previous ones, because they cannot simply rely on a syntactic verification. Instead, the runtime behavior of the application needs to be analyzed. Whereas this is a problem for dedicated static analyses that check specific rules, the logic-based verification techniques that underpin the MOBIUS PCC infrastructure are capable of such a detailed analysis of the runtime behavior.

In the various rules, several kinds of parameters are restricted. In most cases, the restricted data are of integral or string types. We will here focus on the string type, which is more complex than integral types.

In its easiest form, the parameter is specified as a constant, like in the following example.

conn = Connector.open("http://www.somehost.com") ;

Here, the generated code is quite trivial, and it is relatively easy to figure out what actually happens, and to follow the reference into the method invocation:

```
11 ldc #11 ¡String "http://www.somehost.com"¿
13 invokestatic #12 ¡Method javax.microedition.io.Connector.open(String)¿
16 astore˙1
```

Things get more complicated as soon as concatenation starts being used. The following example is semantically equivalent to the previous one, but it is in fact quite different:

```
host = "www.somehost.com" ;
conn = Connector.open("http://" + host) ;
```

This is due to the way in which concatenation is compiled in Java. Since the String type is immutable, it is necessary to compute another string, and a string buffer is used to perform the concatenation itself. The generated code is as follows:

```
11 ldc #11 ¡String "www.somehost.com"¿
13 astore˙1
14 new #12 ¡Class java.lang.StringBuffer¿
17 dup
18 ldc #13 ¡String "http://"¿
20 invokespecial ¡Method java.lang.Stringbuffer(String)¿
23 aload˙1
24 invokevirtual #14 ¡Method java.lang.Stringbuffer.append(String)¿
27 invokevirtual #14 ¡Method java.lang.Stringbuffer.toString()¿
30 invokestatic #12 ¡Method javax.microedition.io.Connector.open(String)¿
33 astore˙2
```

This sequence of bytecode is roughly equivalent to the following Java source code (a local variable sb has been added for clarity):

```
host = "www.somehost.com" ;
sb = new StringBuffer("http://") ;
conn = Connector.open(sb.append(host).toString()) ;
```

The main difficulty in terms of analysis comes from the introduction of a mutable object. This means that the analysis must take into consideration the possible issues of mutable objects, in particular related to multithreading. One way to address the issue is to identify the fact that the object is locally created, and that it is only stored in local variables, and used as argument to well-known API methods which may not 'give away' the reference to another thread.

Other difficulties are related to the locality of the control flow. The Java bytecode has been designed in order to facilitate the analysis of code locally to a method, such as the analysis performed by the Java bytecode verifier. However, in many cases, the analysis required for the analysis of parameter values needs to be performed across methods, like shown in the example below:

```
host = openHttp("www.somehost.com") ;

...

Connection openHttp(String host)
–
  return Connector.open("http://" + host) ;
″
```

In such cases, the algorithm needs to be adapted in order to find out whether or not the value is known at each invocation of the method.

Actually, in this particular example, and for our particular rule (which only checks the protocol used), it is possible to prove that all invocations in the method satisfy the rule without analyzing the parameter to the openHttp method, because the protocol is hard-coded in the method.

This situation is in fact quite common, even in cases where the entire URL is built in a single method. For instance, in the example below, part of the URL is taken from some user input:

```
dest = userInput + "/data.jsp" ;
conn = Connector.open("http://" + dest) ;
```

In such cases, the analysis algorithm must be able to deal with partially known values. The analysis domain must support abstract values like "'a string that start with xxx"'.

In other cases, the way in which the URL is built may bring no information, for instance when the URL originates from Web content. Even then, there are ways to determine that the URL satisfies the rule, for instance if the use of the restricted API is conditional, and that the condition matches the required restrictions, like in the example below:

```
if (url.startsWith("http://"))
  conn = Connector.open(url) ;
```

In such cases, the analysis needs to include more complex control flow analysis, and in many cases some kind of data flow analysis (in order to track the objects on which the various operations are performed). Such cases are quite similar to the checks on sequences (described below).

Mandatory sequences of operations

In some cases, a property may require some checks to be performed. For instance, the properties on the maximum size of a record store may be expressed in a way that forces every modification of the record store that may increase its size to be protected by an appropriate check:

```
if (n + recStore.getSize() ¡ LIMIT)
  recStore.addRecord(record,offset,n)
else
  throw (new RecordStoreFullException());
```

Such rules are rather complex, for several reasons:

- The condition, like in this case, is not necessarily completely trivial, and it may be difficult to assess that it is appropriate.
- The operation must only be performed if the condition is satisfied, at any time (and not necessarily just after the condition check).
- Some action must be taken when the condition is not met.
- Information flow must also be monitored, because the operations monitored must be performed on specific objects.

Information flow restrictions

The last category concerns the restrictions of the flow of information, in which information originating from a given source must not be made available with some other destinations.

Typically, the information that are protected are sensitive, but not confidential. For instance, the contacts of a user are often considered as sensitive. These contacts must be protected against direct disclosure, but they may not need to be protected against more indirect disclosures.

3.15.2 Challenges for the analysis

In the description of the main patterns of properties, we have already identified a few challenges for traditional analysis tools. We here detail some of the challenges that we clearly identify. This part is inspired by Deliverable 1.1, which it complements on some points.

The various challenges need to be addressed in the MOBIUS project, but all solutions do not need to be based on static analysis. For instance, we have already hinted for some properties that it would be preferable to define stricter properties that force developers to use a specific programming pattern, in order to simplify the analysis. Although MOBIUS aims at being as transparent as possible for the developer, we are aware that the proof of properties sometimes requires some changes in developer behavior. We may also consider making recommendations to the companies in charge of standardising the MIDP API's, in order to recommend changes that would significantly help the static enforcement of security policies.

Note that the PCC infrastructure of MOBIUS, especially when it relies on logic-based verification techniques developed in WP3, is much more powerful that standard static analysis tools, as it relies of the full power of program verification. This means it can cope much more easily with the challenges mentioned below than typical static analysis tools. The interesting question is then not so much whether it is possible to cope which these challenges, but whether it is feasible for realistic applications and realistic security properties.

Locality of control flow

The Java Virtual Machine is designed for bytecode verification, but the Java bytecode verifier performs all of its checks locally to every method, as typing information is entirely defined in the method templates used in the declarations.

For many of the security properties listed, the verifications are not all local to a method, and some checks must be global (or at least, they must span across several methods).

Many possibilities are available to analysis designers to cope with this, from method inlining to global static analysis. One of the challenges is to select the most appropriate methods, i.e., the method that cover most of the actual requirements, while keeping an acceptable level of performance.

For the logic-based verification approach developed in Work Package 3, pre- and postconditions for methods provide a natural way to turn global properties into local properties of individual methods. So for the logic-based verification approach, and hence for the MOBIUS PCC infrastructure that uses this, such global properties do not pose as much of a challenge as for more traditional static analysis.

Modeling of the API

MIDP is a framework for client devices. Apart from offline games, many MIDlets mostly consist of a presentation layer for some content that is actually available on the Web. As a consequence, most MIDlets include very few computations, and they rely heavily on the various APIs.

In addition, the semantics of the API is often central for the analysis, at least on two points:

- In some cases, the analysis domains are strongly related to API classes. For instance, the string domain relies heavily on the String and StringBuffer classes. Similarly, container classes like Vector and Hashtable need to be handled specifically.
- In some cases, some properties are strongly related to the behavior of some API methods. For instance, the check on the size of a record store depends directly on the definition of some API methods.

The MIDP API, especially when combined with the various optional APIs available for Java ME devices, are quite large, and modeling them is difficult and time-consuming. When new analysis domains and algorithms are designed, the difficulty of modeling the API needs to be taken into consideration.

Multithreading

The presence of multiple threads has often been mentioned as a challenge in the context of the MOBIUS project. Although these issues are indeed real, the use of threads in MIDlets needs to be studied carefully.

The main use of threads in MIDlets is intended to avoid making blocking calls in the main user interface thread, so the application remains responsive at all times. Very few applications use threads in order to perform multiple operations on data. However, since the user interface, network connection, and computation threads are often separated, there are some interactions between the various threads, as some will consume the data produced by the others.

In the context of MOBIUS, we need to investigate further this issue in order to identify the potential risks of multithreading, as well as the possible ways deal with them, for instance by integrating some additional conditions in the most important properties. This will be addressed in Task 3.3 on multi-threading.

Use of generic containers

One of the challenges, in particular for the typing-based analyses, is the use of generic containers, which is very common in MIDlets. There are mostly two different issues there:

Collection types. The most commonly used collection type in MIDP is Hashtable, using strings as keys. It is used in particular to store preferences and other global MIDlet information. Quite often, these tables are allocated as singletons and accessed through static methods, hence making them accessible from any point in the program. Some applications also use the Vector type or simply arrays, but this is far less commonly used to store security-related data such as URL's.

Persistent state. Most applications available today use record stores to manage their persistent states, but standard file systems may become more widely used in the future, as they become more widely available. Record stores, if they are not shared, are supposed to be accessible only from the application that created them; it is therefore possible to reason about their content, and the issues that they raise are somewhat similar to the issues raised by collection types. On the other hand, shared record stores and files can be accessed by other applications, and it is therefore impossible to make any assumption about their content, unless of course this content is protected against modifications (which is difficult and will not be considered here).

Of course, the main issue with generic containers is that, although their content is structured, it is difficult during the static analysis to use this structure in order to gather interesting information. Most of the information about data is therefore lost when such containers are used, making it difficult to track their content.

The issue is more visible when record stores are used, because all records in a record store are arrays of bytes. This means that any data structures needs to be serialised into an array of byte before to be stored in a record store. In this process, it is very difficult to keep some information using static analysis. For instance, it may be possible to keep track of the content of a Hashtable object as long as it is in the object itself. However, once this content is serialised (using an application-specific mechanism), keeping track of the content becomes much more difficult.

Combination of analyses

The last issue is that a typical security policy contains many properties, and an application must satisfy all of them. Since these properties are likely to be of different kinds, the various analyses need to be combined.

The main worry is here about complexity, since the combination of the different analyses may not always be simple. In particular, if some kind of abstract execution of the application is required, different analyses may require different kinds of abstract executions, leading to potential poor performance.

On the other hand, the combination of analyses can have positive results, since the results obtained on several analyses can refine each other.

As a result, it is very important to think about the analysis as a whole, and to consider the various analyses together.

Chapter 4

Application-specific Security Requirements

This chapter discusses specific security requirements for some representative applications taken from different application domains, namely

- a 'Pocket Protector' application for storing sensitive information, esp. password management, in Section 4.1;
- an instant messenger application 'Mobber' in Section 4.2;
- an application for remote electronic voting in Section 4.3;
- an SSH application in Section 4.4;
- a bytecode verifier in Section 4.5;
- the access controller that is part of the MIDP framework in Section 4.6.

We use these sample applications as the basis for a concrete discussion of security requirements. However, this list is by no means exhaustive and we analyzed other applications with interesting security requirements. For instance, we have omitted from this list the 'Momental Joker' (https://www.dtek.chalmers.se/~tox/d3proj/), a mobile game developed at Chalmers University of Technology, which has very non-trivial security properties as cheating detection, coalition detection, and confidentiality of strategies, as well as sensitive information to protect (e.g. credit card numbers).

For the applications above one may need to certify aspects of their functional behaviour, and not just the absence of unwanted behaviour or some very coarse restrictions on the behaviour as expressed through the 'generic' requirements listed in the previous chapter.

Another consideration in the selection of the applications above was access to code of concrete implementations, so that these can be used as case studies in the other work packages. For the first four applications there are open-source implementations available on the web. A Java implementation of a bytecode verifier has been developed by INRIA, and for the upcoming release of MIDP 3.0 there will be an open-source reference implementation that included the access controller.

The last two applications in the list are platform components, whereas the first four are MIDlets, i.e. applications that run on the platform. However, the SSH MIDlet is similar – as far as both security requirements and implementation are concerned – to a platform component, namely the SSL implementation providing as part of the javax.microedition.io

package of the MIDP API. We decided to use SSH rather than SSL as case study of a secure communication mechanism because it has similar security requirements, it uses similar technology in its implementation, but is smaller in size and there are (several) open-source implementations available.

Secure communication mechanisms such SSH, the bytecode verifier and access controller are crucial parts of the trust infrastructure of the MIDP platform. These are components that need to be certified to the highest possible degree. Moreover, any future global computing platform can be expected to include similar components, as it will require secure communication mechanisms (for instance for secure downloading of mobile code), some form of language-level security by means of typing, and some form of platform-level security by means of sandboxing or other access control.

In the context of MIDP it is not sufficiently realistic (yet?) to consider the possibility of downloading certified platform components.[1] So investigating if the MOBIUS PCC infrastructure for expressing detailed security requirements is not really relevant for MIDP. However, one can expect it to be important for the computing platform of the future.

The applications considered here also provide useful case studies that can be used as running examples to try out some of the ideas and tools developed in the course of the project. In particular, the program verification environment developed in Work Package 3 will be used to check fully annotated versions of some of these applications (maybe not all, as there seems to be some overlap between the security requirements involved). In fact, substantial parts of the applications have already been formally specified as part of the requirement engineering work in this task.

4.1 Pocket Protector

Pocket Protector is an open source MIDP application for secure storage of information. The idea is that it can be used to securely store bank PIN, usernames and passwords, credit card numbers, or other access codes one might want to remember. The applications uses the MIDP Record Management System (RMS) for persistent storage of information. Access to the data in the secure storage is protected by a master password, and the data is stored in encrypted form. The application is available from http://www.eaves.org/jon/j2me/pocket.shtml.

From the perspective of the operator, the application need not be trusted, in that it is not meant to use the network. (In fact, any network usage by the application would be highly suspicious and might indicate we are a dealing with a malicious application that is leaking confidential information.) From the perspective of the user, the application has to be trusted, as it handles confidential data that is of high value to the user.

Because the application uses the MIDP Record Management System (RMS), it can be used to illustrate some of the typical concerns with applications using RMS, already discussed in Section 3.10.2. One typical concern is resource usage: does the application not use too much of the scarce persistent storage? Another typical concern is access control: because the access control mechanism provided by RMS is very basic, one will often want to certify that an application obeys a more specific access control policy than can be enforced by RMS. More on this in the section below.

[1] The MIDP architecture only allows additional APIs to be downloaded as part of an application that uses them.

Background: RMS The Record Management System is the only way to store persistent information, i.e. information that is kept between different runs of a MIDlet. Effectively, it is a very simple file system. The MIDP terminology for a file in this file system is a record store.

Access control to record stores is very basic, using so-called authentication modes. A record store can be AUTHMODE_PRIVATE, which restricts access to the record store to the MIDlet suite that owns it. A record store can be AUTHMODE_ANY, which allows any MIDlet to access it; this provides a way for different applications to share information. Access control can be refined further for record stores with mode AUTHMODE_ANY:: on creation of a record store one can specify if other MIDlet suites can only read the record store, or if they are also allowed to write to it.

Clearly, AUTHMODE_ANY mode should be used very carefully, as confidentiality or possibly also integrity can not be guaranteed for record stores that have this mode.

Threat analysis Clearly all the data stored by the Pocket Protector should be treated as highly confidential. A malicious – or extremely badly written – implementation might store sensitive information in a record store that could be read by other MIDlets, or leak information to such a record store. A malicious implementation might try to send the confidential information over the network. Assuming the implementation is not signed by the operator, it would have to request permission, but hostile application might confuse the use, for instance by frequently asking for confirmation of various tasks, so that a typical user will ignore these messages and simply give confirmation without checking the precise request.

Threats:

- other MIDlets accessing the RMS of the Pocket Protector,
- other executing entities accessing the memory (volatile and persistent) released by the MIDlet,
- a malicious implementation of the Pocket Protector which leaks information or has a backdoor,
- shoulder surfing,
- attacker getting access to the phone (e.g. stealing it) trying brute force attack, possibly dismantling the phone to bypass access control on the RMS.

Security requirements Below are possible security requirements for the Pocket Protector that a user might want to be certified, followed by some discussion and motivation.

We deliberately split this up into many small security requirements, some of which overlap or are subsumed by others, because we want to consider different ways of expressing requirements – as this might allow different ways of certifying them – and also because some of these individual security requirements might be reusable for different applications.

The security requirements range from very generic security requirements, (for example, PP-1), that one might want to certify for larger classes of MIDlets, to very detailed functional properties of the application (for example, PP-13), which one could argue are less suitable or less interesting to be certified.

PP-1 The application must not write to any record stores with mode AUTHMODE_ANY.

PP-2 The application must only create record stores with mode AUTHMODE_PRIVATE.

PP-3 The application does not access any record stores owned by other MIDlet suites.

PP-4 Any data written to the record store must be encrypted.

PP-5 The application creates at most one (alternatively, exactly one) record store (throughout its lifetime, as opposed to every time it is activated).

PP-6 Any persistent memory released in the record store must be zeroed-out.

PP-7 Any volatile memory released (and may be recycled by the garbage collector) the volatile memory must be zeroed-out.

PP-8 Any records from the pockets store displayed on the screen must only appear on screen for x seconds.

PP-9 The application must not use the network.

PP-10 Any access to the record store must be preceded by the user entering the correct master password.

PP-11 After n failed login attempts the application must lock and the content of the record store must be deleted.

PP-12 The master password must never be displayed on the screen.

PP-13 As part of any procedure to change the master password, the user must be asked to enter the correct old master password.

These requirements are concerned with the prevention of leaking of the confidential data, either via the RMS (PP-1...PP-5), via persistent memory (PP-6), via volatile memory (PP-7), via the network (PP-9), or via the telephone GUI (PP-8, PP-12), as explained below. PP-11...PP-13 are standard security requirements for systems that use passwords.

- PP-1 is needed to prevent the MIDlet from leaking sensitive information to other MIDlets via publicly accessible record stores, i.e. record stores with mode AUTHMODE_ANY.

  In fact, this requirement is subsumed by the framework-specific rule rule G-68 on page 38. Note that rule G-68 talks about using only private record stores, whereas PP-1 talks about writing only to private record stores, and so rule G-68 subsumes PP-1. This just goes to show the subtle variations that are possible when specifying these requirements.

- PP-2 ensures that the MIDlet cannot create publicly accessible record stores through which it might exchange information with other MIDlets; PP-2 is redundant if PP-1 is guaranteed, but it seems a sensible restriction. Like PP-1, PP-2 is also subsumed by rule rule G-68 listed in Chapter 3.

- PP-3 is also redundant if PP-1 is guaranteed, since a MIDlet can only access record stores owned by other MIDlets if these have mode AUTHMODE_ANY. PP-3 is identical to rule G-68 listed on page 38.

- PP-4 is redundant if we trust the access control provided by the RMS. Still, it is a sensible defence in depth to use access control provided by the RMS, i.e. making the record store AUTHMODE_PRIVATE, and encrypting the contents, with the master passphrase. After all, the access control of the RMS can be by-passed by for instance physical tampering with a phone, which would be possible for an attacker after stealing the mobile phone or finding it after the owner lost it.

- PP-5 is a sensible requirement concerning resource usage, the resource being persistent store. However, it is also relevant for information flow, because a hostile application could leak information by creating record stores: even if these have mode AUTHMODE_PRIVATE, the existence of a record store may leak information, even if its contents is unreadable. For instance, a hostile MIDlet might create a record store named FirstDigitOfPinCodeIsSeven, and MIDlets in other suites can detect the existence of this record store, even though they no not have access to its contents.[2]

- PP-6 and PP-7 are (rather paranoid) requirements that prevent information leakage through some attacks on the platform architecture. The platform may choose not to zero-out memory that is allocated to an application. In this case, another non-Java application could exploit this by allocating of a big chunk of memory and inspecting if there is anything interesting there. The requirement PP-6 may be redundant in case the requirement PP-4 is used. Unfortunately, this does not hold for the requirement PP-7, as the volatile memory is used to hold the unencrypted record of the confidential data.

  Requirement PP-7 could be enforced with the use of the finalize methods. However, these methods are excluded from the CLDC platfrom we are interested in. Thus the storage must be zeroed explicitly and the control flow analysis techniques must be used to guarantee that before a reference is lost the content of the object is zeroed. Similarly, requirement PP-6 can be enforced by the explicit zeroing of the RMS contents and similar control flow analysis techniques.

- PP-8 is a simple requirement to reduce the risk of shoulder surfing attacks.

- PP-9 is redundant if the MIDlet is untrusted and does not have network access anyway. If the MIDlet is untrusted and only has access to the network after asking user permission, PP-9 is not redundant, as we do not want to rely on the vigilance of the user to prevent the MIDlet from sending say an SMS text message.

- PP-10 could be strengthened by requiring that the user enter the master password directly prior to any confidential data being displayed, i.e. so that rather than unlocking the application once, this has to be repeated for every record of confidential data the user wants to see. This would prevent the possibility of an attacker getting access to the phone while the Pocket Protector is unlocked.

[2] One could argue this is a design flaw in the design of the RMS interface, and that the same error message should be given (or rather, the same exception should be thrown) when one tries to open a non-existent record store or a record store to which one does not have access. This situation is analogous to login procedures, where the error message of a failed login attempt should not reveal whether the password was incorrect or the login name did not exist, e.g. see [HlV05].

- PP-11 would protect against brute force attacks by someone with physical access to the phone. One might argue that this is overkill: a manual brute force attack using the keypad of the phone will be a lot of work for the attacker. A brute force attack is only a real concern if the attacker does not have to enter the password manually, and this will require a very determined attacker, who'll have to take apart the phone and replace the keypad input with an automated device.

- PP-12 is to prevent an attacker reading the master password over the shoulder of the user. One might even go as far as not displaying asterisks, as is usually done when entering passwords, so as not to reveal the length of the master password. This requirement is in fact a standard testing requirement for MIDP application specified in the Unified Testing Criteria (UTC2.1) [Ini06].

- PP-13 is a standard security requirement for password based systems.

- PP-4, PP-10, and PP-11 are different from the other requirements in that they rely on the internals of the Pocket Protector application: whereas for instance PP-1 considers the MIDlet as a black box and only imposes a restriction of the interaction of this black box with its environment (in this case, the RMS API), for PP-4 and PP-10 one has to look at operations internal to the black box, namely the encryption procedure or the master password check.

  PP-4 is an information flow property, but one where the encryption procedure inside the Pocket Protector is considered as a declassification procedure.

Note that the Instant Messenger in Section 4.2 includes some identical or very similar security requirements.

4.2 Instant Messenger

Mobber is a mobile communicator based on a Jabber/XMPP protocol. It allows users to exchange text messages using GPRS in a manner identical to standard PC instant messengers (IM). It is an open source MIDP application, available to download from http://mobber.gryf.info/. Mobber supports standard messenger features like grouping contacts, file transfer or email sending (through gateways). Apart from that it is very crude (lacking i.e. sophisticated GUI), thus provide an effective case study of instant messengers security.

Instant messengers are usually relatively simple applications, mainly because of straightforwardness of communication protocols. Consequently, there are quite a lot of client implementations. Some of them are published by small programming groups, and their quality may be hard to determine. Still, many people prefer to use such charge-free and ad-free implementation over those offered by big companies. On the other hand, users are often very concerned about the confidentiality of their private communication. This creates a need for some way to verify security properties of such clients. Nowadays the only way to do this is by recommendation from other users.

All of the above applies to PC and mobile instant messengers, but the mobile case has some additional properties interesting from the security point of view. There are also some restrictions specific for mobile case. Here we would like to analyse some application-specific requirements that may in principle be verified and certified for such applications.

Security of messengers Most security issues related to instant messaging are solved by the server-side implementation or by the protocol itself. Each client has to be authenticated with a password, and servers ensure consistency of each communication stream. Thus users are well defined, and conducted talks are protected against intrusions. In order to maintain higher levels of privacy, basic cryptographic public-key procedures are supported, allowing encryption of messages. This basically assures users that it is not important what program their communicate with - as long as their own communicator works properly.

The application itself may violate security requirements on many levels, some of which we will discuss below. They may come from malicious implementation, interactions with other MIDlets, or even from cross-scripting from received messages.

Threats From the user's point of view, the instant messenger should provide a security level similar to other communication supported by the device. The contact list of IM (roster) is of the same importance as the standard address book, the communication history is of the same importance as the SMS history. Therefore we may consider encrypting this data in some sense superfluous - stealing a phone and PIN reveals enough private information not to bother of this particular private information. This does not apply to private passwords, which should be additionally protected. Reasons for this we explain in detail in the next section.

The main threats that we may be concerned with are:

- Malicious implementation of the MIDlet that gives attackers access to some phone functionality or stored data.
- Bad implementation, that is vulnerable to some sort of attacks.
- Possibility of identity theft when user's password is not protected well enough.

Security Requirements Below we present a set of possible security requirements that might be checked and certified, followed by a discussion about them. We do not focus only on necessary requirements, but also on some useful ones. Implementations that do not meet some of them still may be very secure when used properly.

IM1 Each communication stream opened by the application that is not obligatory in the protocol must be directly accepted by the user (i.e. the user has to confirm that she accepts opening the communication).

IM2 The application must not input to communication streams any unintended data.

IM3 Data inserted or received in one communication stream must not influence any other communication streams.

IM4 Received data must not be interpreted by the application.

IM5 The application must not require access to any data stored by other MIDlets on the device.

IM6 The application may only connect to servers from defined list.

IM7 The application must force every communication stream to be encrypted.

IM8 The user password may never appear directly on the screen.

IM9 The user password may not be stored as a plain text in device memory.

Discussion

- IM1 is motivated by the possibility that the application may maliciously open connections to some unknown destination, and send some private information (e.g. content of concurrent communication) there. Since there are connections required by protocol, it would be impractical to ask each time for permission from user to do this. Consequently some 'white list' of necessary connection should be stored (visible for user), and only connections not belonging to this list should be verified.

- IM2 addresses creation of content of messages, and must be analysed in more details. What precisely may we consider as unintended information? It is clear that protocol embeds each message in some XML document, whose precise content is not of any concern to user. Nevertheless, some information (like passwords) certainly may not be included in this document. A simple requirement to embed messages in minimal form defined by the protocol is too restrictive - it excludes features like JEP HTTP Binding or file transfer. It suggests that some sort of 'black-list' paradigm should be used to define what data cannot be added by application to message content. This list may overlap with data considered in requirements IM3 and IM5.

- IM3 is a sensible requirement of streams independence. It should be verified with information flow analysis, that data transferred in each stream do not influence creation of concurrent streams.

- IM4 refers to possible vulnerability of instant messenger to cross-site scripting (XSS). One possible case is that a malicious implementation is created deliberately to wait for precise commands from some remote host. A second, and more probable case, is that a flawed XML parser allows to insert some executable code inside messages. The content of messages (after possible decryption) should not need any further interpretation (maybe apart from displaying emotions).

- IM5 is a standard security requirement for network mobile application, and indeed is mentioned in Section 3.10. Any such attempt would raise suspicions if data acquired is not send to some unknown destination.

- IM6 is related to some more sophisticated attack strategies that include implementation of malicious servers. Since most users do not know and do not care what servers they are using, it may be hard to provide any safeguard on application level. One may consider also verifying the list of allowed servers by some trusted party.

- IM7 differs from above requirements in a sense that it is often considered superfluous. In fact many widely-used instant messengers do not force such encryption, and it is not considered as a serious security flaw. Moreover, abundance of messengers without any support for this encryption would seriously decrease usability of such application. Still, this requirement may be necessary in some cases.

  We do not mention here issues related to privacy in public-key infrastructure, which is a whole different topic of research.

- IM8 and IM9 address the security of private user passwords. As mentioned above, for most data stored by instant messenger protection by PIN on the card should be sufficient. This is not the case with account passwords. Even a slight look on phone screen that reveals them, may lead to permanent identity theft. Since there are practically no public services to deal with such type of issues, this may cause even more serious problems that stealing the phone itself. This suggest a need for additional security protection.

Some security requirements address issues for which framework-specific security requirements have been discussed in Chapter 3. For instance, IM5 is related to standard security requirements for record stores discussed in Section 3.10.2, and IM6 to requirements on destination numbers and ports discussed in Section 3.5.1. Also, note that the Pocket Protector in Section 4.1 includes some very similar or even identical security requirements.

4.3 Remote Voting

Overview The Kiezen op Afstand (KOA) Remote Voting System is a Free Software (GPL2) voting system developed for the Dutch government in 2003–2004. "Kiezen op Afstand" is literally translated from Dutch as "Remote Voting." The system was used in the European Parliamentary election of June 2004 and subsequently released under the GNU General Public License, version 2. The system has since been extended with various components including an independent tally system developed with applied formal methods and an adaption to the Irish voting system[?, CK05].

An important extension of the KOA system is the development of a MIDP applet[3] and supporting framework for enabling remote voting from a mobile phone. The domain of voting provides an excellent case study for mobility and security as a relatively small number of concrete security properties are embodied in such a system and voting is a critical technology domain.

The generic security properties we have identified are:

Identification and Authentication: It is necessary to prevent impersonation, voting in an incorrect constituency, multiple votes cast, etc. Any unidentified or unauthenticated access must be prohibited by the system.

Accurate Registration of Voter Intent: The vote that the voter has confirmed to the system must be identical to the one stored in the database.

Anonymity and Confidentiality: There should be absolutely no way of connecting a voter to his/her vote once it is cast.

Data Integrity: Once a vote is cast, there must be no possible way of changing any aspect of it.

Reliability: The database of votes should remain consistent at all times.

System Integrity: Once certified, the code, initial parameters, and configuration information must remain static, and no system change should be allowed during the active stages of election.

[3]Source code for this application can be accessed from svn+ssh://topos.ucd.ie/home/topos/subversion/ucdkoa/. Access is restricted but can be obtained from the authors.

Verifiability and Auditability: It should be possible to verify that all votes have been correctly accounted for in the final tally. The testing procedures should be public. There should also be provably authentic election records.

Non-Coercability and Receipt Freenes: There should be no means for a voter to prove that s/he has voted in a particular way. This precludes vote coercion, vote buying and vote selling.

Unfortunately, non-coercability is impossible to guarantee completely in a remote voting system, since it is always possible for a coercer to physically witness a vote being cast if the voter agrees. (However, partial solutions that work under additional assumptions exist.) Since the complete lack of vote coercion is impossible to guarantee, the emphasis in the KOA system is on limiting the ease of conducting vote coercion.

Threat Analysis There are a number of threats possible in the process of remote voting that could invalidate the electoral count. In the KOA system, during the voting process, the two most important pieces of information are:

- The PIN codes that are used to identify and authenticate a voter with the vote server.
- The details of the vote cast.

These two pieces of information must be protected and communicated only using secure, authorised mechanisms.

Some possible threats are:

- other MIDlets accessing the PIN codes or the choices of a voter,
- any interference with the vote between mobile client and vote server,
- communication of information to any external party other than the vote server,
- any MIDlet accessing personal information from the phone, such as from the phone book, that could connect a voter to his/her vote,
- a malicious implementation of the e-voting MIDlet that displays the correct choices to the voter during confirmation but which sends a different vote to the vote server,
- consistent blocking of a connection between a mobile client and the vote server that impedes a voter's rights to vote,
- vote buying and vote coercion.

Security Requirements This section provides a list of possible security requirements for the e-voting MIDlet that a user would want to be certified, followed by some discussion and motivation. In many ways, the e-voting MIDlet can be considered to have some of the strictest security requirements due to the important nature of elections. Naturally, there are some obvious or trivial security requirements for such an application such as "the application does not initiate phone calls". These will not be enumerated here as this discussion restricts itself to remote voting-specific security requirements.

KOA-1 The application only opens or accepts connections from a constant domain URL.

KOA-2 The application only establishes secure connections.

KOA-3 The application does not use any persistent storage.

KOA-4 The application does not receive messages.

KOA-5 The application does not have any access to personal information stored on the phone (e.g., the phone book, calendar, etc.)

KOA-6 Once a vote is confirmed, it is immutable.

KOA-7 The application must maintain a valid connection to the server at all times.

KOA-8 A voter must confirm the receipt of a "transaction code" before a vote is irrevocably registered in the election database.

KOA-9 The application must be a correct implementation of a well-defined protocol for a voting MIDlet.

- KOA-1 and KOA-2 ensure that the e-voting client only communicates with the specified vote server and that this communication is by HTTPS only. KOA-1 corresponds to framework security requirement G19. KOA-2 corresponds to framework security requirement G38. No other protocols (HTTP, FTP, SMS, etc.) may be used, nor may any other method of communication (Bluetooth, IrDA, etc.).

  These restrictions help maintain the data integrity by ensuring that votes are sent directly and securely to the vote server. Otherwise, any connection apart from that to the voting server could compromise accurate registration of voter intent by sending data to another server first and then altering it and then sending it on to the vote server.

  These restrictions also help maintain confidentiality of the voting system by preventing the communication of sensitive information (such as a voter's PIN codes) except to the proper vote server. This requirement could be extended with a requirement similar to C13, ensuring that certificates used by HTTPS are checked.

- KOA-3 ensures that confidential information, such as the PIN codes of a voter or the details of a ballot, are not stored on the mobile phone. KOA-3 corresponds to framework security requirements G64, G65 and G66 where the number of record stores allowed or the number of kilobytes allowed is '0'. This restriction helps prevent problems of impersonation, ballot leaking and/or anonymity violation that might occur in the case where access is obtained to the persistent storage of the e-voting MIDlet.

- KOA-4 is important, although possibly appearing either trivial or overly conservative. This requirement is necessary to preclude voters being directly canvassed (via SMS, etc.) during the casting of a vote. This security requirement corresponds to the framework property G28.

- KOA-5 is necessary to maintain the anonymity of the e-voting system. Since there must be no connection between vote and voter, any personal information residing on the mobile phone must not be accessed by the application. This includes the phone number of the mobile phone from which the vote is sent. It may not be possible to prohibit access to this phone number and, in this case, it is necessary to ensure that it is not sent over the connection to the vote server.

- KOA-6 is required to prevent any changes to a ballot once the voter has confirmed his/her choice. This maintains the data integrity of the system and prevents the threat of an external party accessing the vote between the client and the server in an attempt to change the contents of the vote.

- KOA-7 is an obvious requirement in a remote voting system. The process of casting a vote should be considered to be an atomic transaction with no possibility of interruptions at any stage.

- KOA-8 is a method of ensuring auditability and verifiability in a remote voting system. Given that a Voter Verified Paper Trail is not possible in such a system, the solution used in the desktop based version of the KOA system is to provide a "transaction code" to the voter. This code can be later checked against a list of votes for a particular candidate or list after the election.

  Although the transaction code can be considered as a receipt, and thus proof of the contents of a particular vote, it is not connected to the vote until after the results have been announced, which limits the ease of possible vote coercion to some extent.

- KOA-9 ensures that the application follows the protocol and does nothing else. This means that it is "functionally correct" in some respects. An example element of this protocol would be that the user interface should correctly reflect the vote of the user.

4.4 Secure Communication Mechanisms: SSH

Secure communication channels, established using cryptographic protocols such as SSL [DA99, FKK96] or SSH [Ylo06], play a crucial role in many applications. Typically these mechanisms are provided as part of the platform. Clearly, a platform component that provides a secure communication channel has to be certified to a highest degree, as the security of any application that uses this, and possibly of the platform itself, depends on it.

SSH (Secure Shell) provides secure communication over an untrusted network. It was originally intended to replace telnet and FTP, but can be used for other purposes. Secure communication is an important building block in many situations, and one that is crucial for security. Any component that provides secure communications is one that one would like to certify, and hence SSH is an interesting application to study.

An alternative case study would be SSL (Secure Sockets Layer), which is now part of the MIDP API. SSL and SSH are different protocols, but have similar security goals - notably providing confidentiality of communication over an untrusted network - and use similar mechanisms. Indeed, some implementations of SSH will use SSL libraries.

We selected SSH as example application MidpSSH[4] for which to consider security requirements rather than SSL, because the application is smaller, because the protocol it implements is simpler, and because there are several open source SSH implementations for J2ME available, which provide interesting case studies. Sun does not distribute the source code of its reference implementation of SSL, and its license forbids any reverse engineering of the binary. We also were unable to get contact with the developers of the existing SSL mobile implementations.

[4]The MidpSSH code is available from http://www.xk72.com/midpssh/

An implementation of SSH (or some other security protocol, for that matter) can be insecure because of flaws at different levels, namely

1. the level of cryptographic primitives,
2. the level of the protocol,
3. the level of the implementation,
4. the level of deployment of the application, and
5. the level of maintenance of the application.

If one of the cryptographic primitives used in the protocol is not cryptographically secure, then the implementation will not be secure either. If the cryptographic primitives are secure, the security protocol may still be flawed. Security protocols are notoriously difficult to design. Indeed, SSH version 1 had such a security flaw (see e.g. [KYW02]).

If the security protocol does meet its security goals and the cryptographic primitives it uses are cryptographically secure, then the implementation may still be flawed. Even if we assume that the implementation is developed with all due care, it may happen that the particular deployment lacks sufficient regard (e.g. the password is set to some default value, the operating system's random number generator is weak etc). At last, the security of a computer system may be compromised because of the insufficient efforts during the lifetime of the application (e.g. new bugs may be introduced, a strong password may be changed to a weak one etc).

We are only interested in the flaws which are pertinent to the implementation of the application. So we assume that the security protocol is correct and meets its security goals, and are then ultimately interested in certifying that the implementation is correct or at least does not include certain kinds of errors. However, we try to predict certain problems that are related to the other levels and that have their impact on the design of the software.

Note that in practice many security flaws on the level of implementation of security protocols do occur. For instance, in the years 2001-2003 CERT published 6 security alerts for security in vulnerabilities in the implementation of SSH, all of which were implementation flaws. The study of cryptographic primitives is a well-established branch of mathematics with a long tradition, namely cryptology. The study of security protocols has quickly developed into an active area of research in the field of theoretical computer science over the last decade. By comparison, the study of implementation flaws is still in its infancy.

Threat analysis The assets protected by the SSH MIDlet are

- the confidentiality and integrity of the data communicated between the user and the remote machine;
- the integrity of public keys of remote sites maintained by the MIDlet;
- the integrity and confidentiality of the own private keys.

The application should also be constructed so that it does not introduce additional availability restrictions to the services offered by the remote machine (e.g. it should not deadlock even though the remote machine works properly). The additional asset which is strictly connected

with the mobile phones platform is the cost of the connection — the application should use the network only when necessary.

The particular kinds of threats to an SSH application are:

- the input channels of the application can carry data which impacts the application so that it leads to asset compromise,
- the application can sent data to output channels so that assets of the application are compromised,
- a neglectful initialisation procedure can introduce the SSH implementation to an instable state which can lead to an asset exposure,
- an incorrect control flow after an exception have been thrown can introduce the SSH implementation to an instable state and disclose assets,
- improper handling of the well formed SSH protocol messages and not well formed SSH messages can endanger the assets,
- poorly designed interaction with humans can ease the social attacks for the assets,
- bugs in application can bring it in an unstable state and can hazard the assets.

Security requirements The mentioned above threats lead to the following possible security requirements that an SSH application designers may want to verify.

SSH-1 The MIDlet JAR file is signed.

SSH-2 The record stores used by the application are private.

SSH-3 The configuration data supplied by the user or downloaded from the network is checked to make sure it is in the proper format.

SSH-4 The security-sensitive data such as passwords or keys should not be stored in the record store unencrypted.

SSH-5 It should not be possible to set up an unencrypted SSH connection (or at least the user should be explicitly warned when such a connection is to be established).

SSH-6 The control characters displayed to the user should be escaped.

SSH-7 The strings which serve to initiate HTTP connections should be checked that they start with http://[5] and the strings which serve to initiate SSH connections should be checked that they start with socket:// corresponding to the intents of the user. No other connections should be attempted.

SSH-8 Exceptions can be divided into three different kinds depending on the way they should be handled. Runtime exceptions (e.g. ClassCastException) should not be thrown, runtime errors should not be caught in the application (e.g. NoClassDefFoundError or OutOfMemoryError) while the exceptions connected with the control flow of the application should be carefully caught and served.

[5]The MidpSSH application allows the user to download some configuration information from the web.

SSH-9 The only allowed input-output operations in the exception handling blocks are the fixed messages displayed to the user.

SSH-10 The conformance to the SSH protocol should be strictly followed, i.e. the MIDlet should be a functionally correct implementation of SSH and nothing but SSH, as defined in [Ylo06, Leh06].

SSH-11 The security-sensitive data such as the default password values, the default login names, the default computer addresses to connect to, the default cryptographic keys or initialisation vectors should not be initialised as constants.

SSH-12 Each time a configuration choice made by user can lead to an asset compromise, the user should be explicitly warned about that.

SSH-13 The class which performs the SSH operation should be carefully separated from GUI and the classes that do the input-output operations so that the errors in the latter does not affect the former.

SSH-14 The data in the input communication thread should not be visible from the data in the output communication thread except from what is allowed by the SSH protocol class.

SSH-15 The application should use a cryptographically secure pseudo-random number generator.

The requirements SSH-1-SSH-3 address the threats connected with the input channels. The threats concerning the output channels are taken up in the requirements SSH-4-SSH-6. The initialisation of data is mentioned in requirement SSH-7. The constraints SSH-8-SSH-9 apply to the management of the exceptional flow while the requirement SSH-10 to the flow of the SSH protocol. The requirements SSH-11 and SSH-12 involve the proper arrangement of the user processes and the requirements SSH-13 and SSH-14 concern the design of the application.

In principle, the distinction between the particular categories is not always rigid, so some of the requirements may be also considered to be in groups where they are not explicitly mentioned.

- SSH-1 protects the application from malicious modification on its way between the code producer and the code consumer. This practically guarantees that the code was authorised by the party who signed it. It also prevents from upgrading the application with some other code (possibly with malicious content). It is especially important in the case of an SSH application as it is very easy to modify its source code or bytecode so that it can send passwords or session transcripts to unintended third parties.

  The use of this requirement can be lifted in presence of a PCC architecture where the trustworthiness of security policies may give the users confidence when using the application in the case they do not trust the provider of the code.

- The requirement SSH-2 protects the application from modification of its behaviour by means of redefinition of its configuration files or files with certificates or keys. This requirement is similar to the framework requirement rule G-68 on page 38, where additional rationale for use such restrictions in the context of security-sensitive applications is given.

This requirement can be strengthened to not just protect the data in record stores from other Java MIDlets running on the platform (which can try to access persistent data via the platform APIs), but to also protect the data from native code on a mobile device (that may by-pass the platform APIs to access persistent storage). To to this, it should be required that all data stored in the record stores by the SSH MIDlet is encrypted (e.g. by a master password that it provided by the user on start-up so it does not need to be stored on the device).

- SSH-3 protects the application from reconfiguration or from sending improper SSH or terminal commands which may affect the availability or even expose the passwords or the communication content. A rule of this kind is quite common in general security-sensitive applications as it effectively protects them against SQL injections or improper configuration settings which can expose passwords or other protected data.

- SSH-4 is a general rule to protect the secret data from exposing to other (non-Java) applications on the phone or any kind of access that circumvents the Java environment. The data should be protected with a master password which should not be stored anywhere. The general goal of this rule is similar to the one in SSH-2. However, we assume here that the potential attacker is stronger.

- SSH-5 protects the user from exposing her/his password and the content of the communication. In general, the SSH protocol admits the encryption scheme where the data is transmitted clear text [?, section 4.11.1]. This feature is convenient when the implementation is debugged, however, it is insecure when present in a working program. In this light, the feature should be omitted from the implementation. In case it is impossible, the user should be warned that she/he tries to launch a connection which can expose the assets. That is why the requirement can also be treated as a requirement where the proper human interaction is proposed. This requirement is also expressed in the SSH architecture document [?, section 9.3.1]. The goal of the restriction is similar to the one in SSH-4.

- SSH-6 is one of the rules mentioned in the security analysis of the SSH protocol [?, Section 9.2]. It prevents attacs which exploit the special meaning ot control characters which allows to hide some warnings or show some commands that can lead the user to unintended password exposure.

- SSH-7 assists in making sure that the user connects to hosts which correspond to her/his expectations. The application uses HTTP connections to fetch the configuration data (in particular the addresses of remote SSH servers) and the socket connections to do the SSH communication. As these are the only kinds of connections that are made, the application should not attempt any other connections and should not mix these two communication modes. This requirement is an interesting variation on the framework requirement rule G-21, where the connection string should start with determined form only, and rule G-37, rule G-38, rule G-39, where the range of available connection kinds is restricted.

- SSH-8 is a kind of rule that ensures careful handling of exceptions in the application. The way the requirement can be verified depends strongly on the particular way the exceptions are used in the application. If an exception, when thrown, transfers control

through several objects, it may be the case that the intermediate objects are left in an inconsistent state. This is especially important in the context of applications that handle a specific network protocol like SSH where the internal state of the protocol should be carefully preserved and where some of the transitions in the internal state of the protocol are by custom realised throug exceptions. Carelessness in handling of exceptions may lead to asset exposure in case the internal state of the protocol is left inconsistent.

- SSH-9 restricts the catch blocks so that it is impossible for attackers to read anything from output channels in case an exception is handled. The only input-output operations that can take place here is displaying fixed strings. This protects the application from leaking the assets in this part of the source code. This can improve the security as the exceptional flow is designed less carefully than other parts of Java applications.

- SSH-10 is the essential requirement for security protocols. One of possible source of holes in networked applications is that they admit protocol messages which are not fully conforming to the original protocol which sometimes leads to security problems. This restriction means that wrong SSH messages should be discovered as soon as possible and suitable actions should be taken (either the messages should be ignored or the communication over a connection should be stopped).

- SSH-11 serves to make sure that the user of the application adheres to proper password and login creation rules as well as key management practices. It prevents from setting a password, login or key to default, commonly known values.

- SSH-12 is another rule which sets up the communication between the user and the application. Each time the application goes into a state which is predicted in its design, but may cause the asset compromise, the user should be explicitly warned. Otherwise, the user might unintentionally or carelessly choose an option which could expose her/his password or communication with the remote site. This requirement is explicitely mentioned in the SSH architecture document [?, Section 9.3].

- SSH-13 is a rule to ensure robustness of the application. It follows the rule that the data should be only available where it is really needed. In particular, the passwords and keys should not be readable from the GUI. Similarly, the state of SSH protocol should not be readable from within the objects which perform the input/output operations.

- SSH-14 is a rule similar to the one above. It ensures the separation of the input information flow from the output information flow. It is important as for instance unrestricted access may lead to printing out on the local console the password which is sent out to the remote site.

- SSH-15 is motivated by the fact that weak random number generation is a standard security weakness Indeed, Sun's MIDP Reference Implementation of SSL have been shown to be vulnerable to an attack exploiting the weakness of the pseudo-random number generation (PRNG) used [SMH05]. Moreover, it is one of the major security issues mentioned in the security considerations for SSH [?, Section 9.1]. Checking the strength of the random number generation is beyond what we can hope to certify by means of our PCC infrastructure. However, we should check that the application uses

the pseudo-random number generation algorithms provided by the platfrom, rather than implement its own. (Similarly, one could require that the application uses platform-provided implementation of encryption algorithms such as DES and RSA, rather than implement its own crypto.)

4.5 Byte Code Verifier

The byte code verifier (BCV) is supposed to be run on every application, before it will be accepted on the device. It guarantees that during execution there will be no type errors or stack overflows. The BCV is defined in the Java Language Specification [GJSB05a], while Leroy's survey [Ler03] presents detailed hints and justifications. Notice that we consider here an implementation of the BCV for the full Java language, including subroutines. Correct functioning of the BCV is crucial, because all applications that are downloaded on the device have to be accepted by the BCV. In fact, in the past, security holes in the BCV for MIDP/CLDC have been found (and exploited).

Secure component loading A typical scenario for Global Computing is the downloading of system components over the network, for example to replace an existing component with one that has more functionality, or that corrects a bug or security hole. Allowing to download system components also imposes extra demands on the security requirements that are to be guaranteed. For a normal application, one needs to show that it does not violate the overall security requirements of the system, that it does not leak or corrupt confidential information etc. However, to be able to securely download a system component, one also needs to know that it functions correctly, otherwise it could impact the overall well-functioning of the system. For example, one only wants to dynamically install a new access controller or byte code verifier on the device, if one can be sure that the component will actually do the job that it is supposed to do. Thus, for such system components the security requirements will also consist of functional specifications, describing the expected behaviour of the component.

A quick description of the BCV algorithm The BCV is based on a data-flow analysis with an abstract virtual machine that only uses data types. It uses the so-called Kildall algorithm. With each instruction, a type state is associated, that corresponds to the memory state where data are replaced by their types. The BCV iterates on a sequence of instructions. For each instruction, it calculates the type state after its execution and then unifies this with the type state of the successors of the instruction. Unification is possible if one of the following conditions is verified:

- one of the two states is undefined. In this case the resulting state equals the already defined value;
- the two states have the same stack height, the same types for their respective variables or their types can be unified. When two types are unified, they are replaced by their lowest common supertype. There is a type lattice that can determine if two types can be unified (see Leroy's paper for a detailed description).

When the type states are not modified anymore, we have reached a fixpoint, the algorithm stops, and the byte code has been verified. If the next type state of an instruction cannot

be computed, either because of a stack underflow, or because the type of the variables is wrong, byte code verification fails. Also if unification between two states is not possible, the verification fails. Notice that this process is called monomorphic byte code verification (to avoid the subroutine problem, one should use a polymorphic byte code verifier, see e.g. Guillaume Dufay's formalisation in Coq [Duf03]).

Threat analysis A malicious or incorrect implementation of the BCV can block the device in several ways. First of all, it could accept malicious applications, that are not type-safe. Second, it could not accept any applications at all, or it could not terminate, thus blocking the whole device. Finally, we want to ensure that the BCV does not change the applications it analyses in anyway (and in particular, that it does not insert malicious code into them).

Security requirements Below we list possible security requirements that a user might want to be certified for a BCV implementation. We distinguish several types of properties: first the ones related to the implementation of the byte code verifier, then the properties that should be guaranteed through the abstract execution over the type states, and finally the properties that ensure the correctness of the abstract execution.

BCV-1 The application must not throw any run-time exception.

BCV-2 The byte code verifier must be visibly pure.

BCV-3 The byte code verifier must terminate.

BCV-4 If the application is accepted by the BCV, its instructions will not produce stack overflow or stack underflow.

BCV-5 If the application is accepted by the BCV, its instructions will not produce any type error while accessing data in the stack.

BCV-6 If the application is accepted by the BCV, its instructions will not produce any type error while accessing registers.

BCV-7 If the application is accepted by the BCV, the stack size and the types stored in type states of the application should be consistent for the different execution paths.

BCV-8 If the application is accepted the registers and the register types stored in type states should be consistent for the different execution paths.

BCV-9 The abstract execution of an instruction simulated by the BCV algorithm must correspond to an execution of this instruction in the concrete semantics.

BCV-10 After the execution of an instruction, the obtained state can be abstracted to the abstract state obtained after the abstract execution of an instruction.

Properties 1 to 3 are implementation-related issues. In contrast, properties 4 to 8 describe the properties that should be satisfied by the analysed application (and thus by all its instructions) after it has been accepted by the BCV. These security requirements are functional specifications, describing the required behaviour of the BCV. Properties 7 and 8 are properties that should be guaranteed for instructions that can be reached through several execution

paths. Finally, properties 9 and 10 are related to the correctness of the interpretation of the instructions over a type state. Notice that these properties will help to prove the correctness of properties 4 to 8.

- BCV-1 - This property says that the byte code verifier should not throw any unexpected exceptions, i.e. no run-time exceptions, and throw only those exceptions that are declared explicitly (and thus have a meaning for the program). For example, to signal that an application is rejected the BCV can throw an exception, but it should not throw any NullPointerException.
- BCV-2 - The byte code verifier being only a verifier, it should not visibly modify its environment, and change data related to any other program; it can only modify the global memory state by creating new objects. In particular the instructions given to the BCV should not be changed.
- BCV-3 - The byte code verifier should terminate properly. With each iteration of the loop, the type states are more refined or remain the same. In fact the algorithm stops when the type states are the most refined as they can be.
- BCV-4 to 8 - The byte code verifier verifies these properties over the code via the abstract execution of the byte code. These properties are valid only once the algorithm has terminated and if the code is not rejected.
- BCV-4 - The execution of the code does not result in stack overflow/underflow, otherwise the code should have been rejected by the algorithm.
- BCV-5 - The instructions that pop data from the stack are executed when data with the right types is on the stack; for example, when i2s is executed, there will be an integer on the top of the stack and not a reference.
- BCV-6 - The same as BCV-5 but for load and store instructions.
- BCV-7 and 8 - At each join of two or more execution paths, the stack length must be the same for each execution path and stack types must be compatible, that is the same for the registers.
- BCV 9 and 10 - These properties are used to show the correctness of the byte code verifier with respect to the JVM semantics. To formally define or to prove these properties, we need a formal definition of a JVM in order to show that there is a bijection between the executions:

$$\begin{array}{ccc} St_i & \xrightarrow{\text{exec}} & St_{\text{succ}(i)} \\ \uparrow & & \uparrow \\ S_i & \xrightarrow[eval]{} & S_{\text{succ}(i)} \end{array}$$

  where exec is a step of the abstract execution in the byte code verifier and eval is a step of the execution in a JVM.
- BCV-9 - First we must show that for each program state which can be abstracted to a byte code verifier 'type'-state, we can execute the associated instruction of the given JVM, without any errors.

- BCV-10 - Each calculated memory state is an abstraction of the resulting memory state that a JVM will generate by running the instructions.

Some remarks on implementation, specification and verification of the BCV To understand how the requirements above can be specified (and verified) formally, we are experimenting with a Java-implementation of a BCV, annotated with JML.

Most 'modern' implementations are based on visitor design patterns with an instruction graph (see the Java implementation mustang.dev.java.net/, ovm/j (http://www.ovmj.org), the verifier in BCEL (http://jakarta.apache.org/bcel/), etc.). It is problematic to specify and verify this kind of implementations with JML, in particular to show termination, as this would require us to express much stronger properties on the instruction graph before the beginning of the algorithm (notice that this problem is due to the lack of expressiveness for properties over graphs in JML). Therefore, to be able to prove termination relatively easy, we have chosen to use a home made BCV implementation that iterates through an array containing the instruction sequence.

To specify the correspondence between an abstract execution of the BCV and an execution of the JVM, we use the JVM semantics as specified in Bicolano[6]. To connect the JML annotations with the Bicolano specification in Coq, we use the JML native construct [Cha06].

Also to prove termination, we use the JML native construct. In JML the termination witness (the decreases clause) only allows to use integer expressions. We encode this by defining a mapping of memory states to numbers, for which we can show that a maximum exists. This is the case, since any given JVM machine has a finite number of type states. It is not easy to do this in JML, while it is much easier to do this on the target prover's level. The native construct then allows to link this integer expression at the target prover's level to an integer in the JML specification.

4.6 Access controller

Security of the MIDP framework is ensured by an access controller that should check if access to protected API has been granted by the user under the restriction of the current security policy of the terminal. This component of the Application Management System (AMS) can be either implemented in Java or natively. We will consider the verification of a Java version of a MIDP Access Controller.

4.6.1 MIDP 2.0 access controller

The MIDP 2.0 access controller should load the handset security policy (usually stored in a file using the notations proposed in chapter 3 of MIDP 2.0 specification). It must then check the signature of the application to settle the rights granted to the MIDlet suite. It must also take user settings into account. Those settings can make the security policy more stringent than the default. The result of this initialisation phase is usually coded as an internal permission object used by the MIDlet.

Each time a MIDP protected API is called, the access controller must determine the permission name and checks if the user grants the access. The value associated to the permission may be:

[6]See http://www-sop.inria.fr/everest/personnel/David.Pichardie/bicolano/main.html.

Forbidden the check fails.

Allowed the check succeeds.

User - blanket the user is asked only once, the first time the MIDlet is used. The check succeeds only if the user authorises the use of the API.

User - session the user is asked once each time the MIDlet is used.

User - one shot the user is asked each time the API is called.

There are several properties that should be verified by the access controller:

- The permissions granted on the terminal are granted once and for all and are correct:
  - The permissions read by the AMS from the system policy file are the basis for the computation of the permission object
  - the level of trust of an application depends of its signature and this verification cannot be by-passed (verifying the correctness of the signature algorithm is out of the scope of the project)
  - the user can modify permissions but only within the range allowed by that policy and the Recommended Security Policy for GSM/UMTS section of the MIDP specification. An application cannot modify those settings.
- It is not possible to call a protected API without performing a successful check (with the correct arguments) of the policy enforced. [7]
- A check is correctly implemented :
  - Session and once for all permissions are correctly handled.
  - Security screens cannot be overridden by the system [8].

4.6.2 Extended access controller

There are several reasons why the current access controller does not provide a satisfactory user experience:

- each confirmation is about an atomic action. There may be many atomic actions during the execution of a MIDlet. Too many screens defeat the principle of user-validated security policy as the user ends-up clicking the Ok button without looking at the security screen.
- the behaviour of a MIDlet can often be viewed as a sequence of transactions. A given transaction should not be cancelled in the middle because the user was not ready to grant the rights for its execution: those rights should have been requested before the transaction started (otherwise the user may have already paid for something he will cancel).

[7] Even such simple checks can have flaws in its implementation. Laurent Gottely (France Télécom R&D) had developed a technique for analysing the implementation of similar security checks in J2SE and found a bug with sockets.

[8] There is a known published bug on an old Java MIDP 1.0 phone where a call to setCurrent(mask) followed by a call to the method sending an SMS would result in the screen mask overriding the security screen for the SMS.

A solution to this problem is to split the request for obtaining the rights from the consumption of those rights and to provide a way for requesting several rights at a time.

F. Besson, G Dufay and T. Jensen [BDJ06] have proposed such a framework where the rights granted are global and consumed by whatever method call requiring them. They have developed a static analysis to check that the number of calls does not exceed the number granted in the authorisation.

C. Alvarado has proposed to introduce permission objects. A permission object would first accumulate requests for performing some critical actions. Then it would be validated. Finally permissions would be consumed by the calls to protected APIs (that would take a new permission argument).

4.6.3 Verifying the access controller

The availability of the reference implementation of MIDP 3.0 (next version of the specification) as open source software (as announced by Motorola at JavaOne 2006 conference) will provide a complete test-bed to experiment implementations of a modified access controller. Moreover, the reference implementation itself can be checked against the requirements drawn in the first subsection

MIDP 3.0 access control policy may be an enhancement of MIDP 2.0 policy. A public requirement for MIDP 3.0 is the ability to have several MIDlets running at the same time which imposes new isolation properties on the different permission objects in use.

Chapter 5

Formalisation of security requirements

In this chapter, we discuss how to express security requirements in a formal specification language. We focus on properties that are relevant to the MIDP framework and MIDP applications. Because formal proofs require formal specification, formal specification is an obvious prerequisite for proof-carrying code. Formal specification in a uniform language is also useful for comparing and categorising different security requirements.

This chapter focuses on expressing security requirements as security contracts [WPSJ05] expressed in the Java Modeling Language (JML) [LPC+05] or its bytecode-level cousin BML [BP06], which is developed in WP3.1 as the core specification language for the MOBIUS project[1]. The work reported in this chapter highlights some additional notions that are needed to make certification of security contracts in JML/BML possible or feasible. We also show how certain high-level specification formalisms can be translated to core JML. (For instance, finite state machines or temporal logic formulas for specifying legal orderings of method invocations can be translated to core JML using JML's so-called ghost variables [TH02].) We discuss why we will need certain extensions of core JML, e.g. JML's recent concurrency extension to better support multithreaded programs [?].

Note that it was already foreseen that a generic specification language like JML is not be the most convenient formalism to specify certain kinds of security requirements, especially those concerned with information flow or resource usage. Deliverable D1.1 already focused on resource and information flow requirements, for which dedicated type-based and logic-based formalisms will be investigated in tasks 2.1-2.4 and 3.2.

For access control we also investigated a dedicated formal model, as opposed to the generic specification formalism of JML. This investigation was sparked off by the access controller discussed in Section 4.6. This work resulted in a formal access control model for application frameworks where access control can involve user interaction to obtain permissions. This work is reported in the ESORICS publication 'A Formal Model of Access Control for Mobile Interactive Devices' [BDJ06], and we have chosen not to duplicate this material here.

Some of the simpler security requirements, especially some of the framework-specific requirements listed in Chapter 3, are well-suited for enforcement by basic (trusted) static analyses techniques, as well as proof-carrying code. Still, our PCC infrastructure should be capable of expressing even such relatively basic properties. Moreover, we want to be able to compare and categorise these requirements along with the more challenging ones.

[1] As far as expressivity is concerned JML and BML are the same, except for less stable and more experimental features that are included in JML but not in BML.

We have structured this chapter as follows: In the first five sections, we formalise requirements that are directly security-related: API usage requirements in Section 5.1, requirements on exceptional control flow in Section 5.2, resource usage requirements in Section 5.3, (explicit) information flow requirements in Section 5.4. API usage and exceptional control flow requirements are particularly important: they are often easy to specify, sometimes easy to verify, and often sufficient (at least intuitively) to guarantee certain high-level security properties. API usage requirements come up particularly often in the requirement list from chapter 3, which was provided by the industrial partners.

The final four subsections of this chapter address additional security-relevant properties of Java programs that we plan to work on. Section 5.6 considers concurrency-related properties, Section 5.5 object immutability.

5.1 API usage

Many of the security requirements in chapter 3 restrict API usage. Some restrictions concern single method calls. For instance, some providers may want to completely forbid certain security critical methods or may want to impose tight restrictions on method parameters. Other restrictions concern the order or the number of certain method calls. For instance, a provider may want to impose a format for games where the number of calls to an SMS-sending method never exceeds the number of user requests to start a new game. Such a restriction would protect users from unexpected costs for SMSs.

A first classification of restrictions on API was proposed in [CA05], which suggests the following categories, in order of increasing complexity:

- controlling the use of a given class or method,
- controlling the possible values of the arguments of a given method,
- counting the number (or at least a bound) of calls of a given method,
- controlling the use of data inside a MIDlet: transfer of information between the user interface, the store, the networks, and eventually also with other devices like the SIM,
- controlling more accurately the temporal behaviour of the MIDlet.

The subsections below refine this classification bit further, and discuss how each category can be formalised. We start with the following broad classifications

- Restrictions on single method calls. These can
  - simply forbid the use of some method call, or
  - restrict the values that can be used as an argument, e.g. arguments that are 'constant' or 'determined', or have a constant prefix.
- Temporal requirements on method calls, i.e. constraints of sequences of method calls. These can
  - constrain the number of methods calls (e.g. method m can only be call 10 times),

– constrain the order of method calls; e.g. static method m cannot be called before static method n, or instance method m cannot be called before instance method n on the same object, etc.; or
– constrain the data flow of arguments and results of methods, e.g. static method m is only called with arguments that are the result of call to method n.

Of course, many combinations of such specification patterns are possible.

5.1.1 Restrictions on single method calls

Forbidden methods

Some providers may want to switch off some security critical functionality on their phones. For instance, they may want to disallow to use the push registry. The push registry enables MIDlets to set themselves up to be launched automatically by inbound network connections.
To prohibit dynamic push registration, providers may want to forbid the method PushRegistry.registerConnect
Technically, we can specify this in JML by requiring the precondition false:

```
package javax.microedition.io;
public class PushRegistry –
  ...
  //@ requires false;
  public static void registerConnection(String connection,
                                        String midlet,
                                        String filter)
                                 throws ClassNotFoundException,
                                        IOException –...″
  ...
″
```

The “forbidden method” security requirements from chapter 3 are meant to forbid client calls to API methods but not internal API calls to these methods. Therefore, the JML “requires false”-specification faithfully expresses the security requirement only under the additional proviso that API specifications are treated as abstract interfaces for API clients. Those abstract interfaces express the properties that API clients may assume.

As pointed out on page 18, the absence of calls to forbidden methods can be checked by a relatively simple static check. In fact, such checks might be incorporated into the bytecode verifier.

Here is a table of forbidden methods that are mentioned in the security requirements from chapter 3:

security requirement	forbidden method
G-4	java.lang.Class.forName()
G-8	all protected methods
G-44, G-46	javax.microedition.io.PushRegistry.registerConnection()
G-45	javax.microedition.io.PushRegistry.registerAlarm()
G-57	javax.microedition.midlet.MIDlet.platformRequest()
G-67	javax.microedition.rms.RecordStore.openRecordStore()
G-69	javax.microedition.io.file.FileConnection.delete()
G-70	javax.microedition.io.file.FileConnection.create()
G-71	javax.microedition.io.file.FileConnection.createOutputStream()

Discussion 5.1.1 (Direct vs. indirect calls). An informal security requirement that 'some (untrusted) application A is not allowed to invoke some method m' can be interpreted in two ways, in a setting where several applications may be active at the same time. To understand this, suppose that we have another (trusted) application B this is allowed to call m. Is it now allowed for A to call a method of B which in turn calls m? Here there are clearly two options, which depend on whether A's security policy also applies to B when it is carrying out requests of A. (The stackwalking mechanism of Java's code-based access control can enforce both policies.)

In general we have to be aware of this issue, as soon as we have different security policies for different parts of the code base and there can be transfer of control between these parts. Note that the two possible interpretations of the informal security requirement then have a direct impact on the way code is certified: If B is always allowed to call m, even when acting on behalf of A, then we can certify A without looking at the code of B. If B is not always allowed to call m when acting on behalf of A, then when certifying A we have to know the code of B, and (re)certify that B also meets the stricter security policy imposed on A. Note that this makes certification highly non-modular.

In the MIDP setting this is not so much of an issue, as only one MIDlet is active at the same time, and MIDlets cannot call each other directly. So the scenario above is only possible if A is a MIDlet and B is the platform. Indeed, all the requirements on API usage in Chapter 3 are only concerned with direct calls to the API, not with indirect calls via third-party code, as the only third-party code around is the API code itself. □

Restrictions on the value of method parameters

Even more common than forbidden methods are restrictions on method parameters. For instance, the name-parameter for Connector.open(String name) may be required to begin with the prefix "http://". As a result of this requirement, the MIDlet would only be allowed to open http-connections. This can be expressed as a simple precondition:

```
package javax.microedition.io;
public class Connector –
  ...
  //@ requires name.startsWith("http://");
  public static Connection open(String name) throws IOException –..."
  ...
"
```

Chapter 3 suggests that such preconditions can often be guaranteed by enforcing coding conventions that can easily be checked statically. For instance, a particularly simple convention requires that the actual name-parameter is a string constant with an "http://"-prefix:

```
...
Connector.open("http://somehost");
...
```

A more liberal convention would require that a given static analysis algorithm is able to determine that the actual name-parameter has an "http://"-prefix:

```
...
```

```
String protocol = "http://";
...
String host = "somehost";
...
Connector.open(protocol + host);
...
```

Coding conventions of this kind depend, of course, on the static analysis algorithm. It may be, for instance, considerably harder to detect that the name-parameter begins with "http://", if for efficiency it is built using a mutable string buffer and then converted to a string. Still, it seems very reasonable to enforce certain coding conventions that make common restrictions on method parameters statically checkable. For instance, disallowing that name-parameters for Connector.open() are built from string buffers does not seem to impose a serious performance penalty, because typical MIDlets do not open a large number of network connections during a single run. It would be nice if, in addition to being amenable for static analysis, such coding conventions would be designed to be clearly explainable to developers as well.

Many of the security requirements given in Chapter 3 that involve restrictions on method arguments talk about arguments being 'constant' or 'determined'. These concepts are explained in Section 3.3. Below we discuss the formalisation of these concepts:

Discussion 5.1.2 (Formalisation of 'constant' and 'determined'). A subtle issue that affects the formalisation of many of the requirements that give restrictions on the value of method parameters is the formalisation of the notions of arguments being constant or determined. The concepts, introduced in Section 3.3, are used in many security requirements. One can argue that they are not semantic properties, and that semantically we do not care about the distinction between them.

The meaning of an argument being a constant is clear. The Java Language specification ([GJSB05b], §15.28) defines a notion of (compile-time) constant expressions. Constant expressions can be values of primitive types or references of type String. However, the notion of being a constant expression cannot easily be expressed in JML/BML.

One option might be to use a notion of a so-called strong invariant, i.e. an invariant that never be broken, unlike a normal JML invariant, which can be temporarily broken during a method as long as it re-established at the end. Such a notion is not part of JML or BML currently, but it has been proposed [?]. This notion may be needed in a multi-threaded setting, as will be investigated in Task 3.3.

One can express a more general property, namely that a given expression is equal to some constant string at some program point, using an assert statement. For instance, by an assert directly prior to some method call, we could express that a given variable is guaranteed to have a given constant value at some program point.

```
//@ assert url.equals("http://www.somehost.com");
Connector.open(url);
```

However, this is a weaker property than url being a constant expression, because even if url is not a constant expression, and url has another value at other program points, the assertion might still be true. However, from the perspective of security the property that url is guaranteed to have a particular value at a given program point is just as good.

In fact, the assert above essentially expresses the property that the argument url is 'determined'. However, whereas the definition of 'determined' in Section 3.3 is defined wrt. a

given analysis, the assert above expresses that the argument url is determined in the sense that for every execution the argument will have the same value, so this notion of being determined is independent of any analysis.

In a similar way one can of course express a requirement that for instance the protocol is 'determined'

```
//@ assert url.startsWith("http://");
Connector.open(url);
```

So, we can for instance formalise the requirements listed in Table 3.1 on page 24 in JML. However, note we cannot distinguish between the first requirement in this table, of something being a constant, and the second requirement, of something being determined. Still, considering the motivation behind these requirements, it is not a serious problem that we cannot discriminate between these properties. Indeed, for most of the security requirements about some method argument being constant, there is a twin security requirement about that method being determined.

A drawback of the approach described above is that security requirements of an application are expressed (as assert statements) in the application code. This is a drawback in the PCC setting, as for certification one not only has to prove that these assertions are true, but one also has to check that all calls to restricted API calls are preceded by an assert statement of the correct shape. After all, one should not trust an untrusted application provider to do this. Then, for certification by means of PCC it is preferable to express security requirements separate from the application code.

A more elegant way of formalising the property of arguments being constant/determined is with a precondition, also known as requires clause, on the method that is called, rather than with an assert at the call site. But note that the notion of a argument being constant or determined implicitly talks about a particular call site. E.g. the code of some application could contain two calls to Connector.open, each with a constant (or determined) argument, but a different constant (or determined) argument for the different call sites. If we want to express a restriction that the argument of a method is constant or determined with a precondition of that method, this precondition has to consider a set of constant/determined values.

A way to do this would be to declare some set of strings, and specify a contract for the API method that the argument must come from that set. This scheme is illustrated below. This avoids the need of assert's in the application code at every call site.

```
public class Connector–
 //@ public final static StringSet DETERMINED˙URLS = new StringSet(...);

 //@ requires DETERMINED˙URLS.contains(url);
 public void open(String url)
```

Of course, we could write this as a disjunction but the set notation is more convenient. Also, it is quite natural for the certification of an application to require the developer to provide a list of arguments that the application will use.

To conveniently express and reason about such specifications, it would be useful to have suitable notions such as (immutable) sets or lists of data, e.g. the class StringSet with the method contains used in the security contract above. Such notions could be made available

in JML's API of model classes,or, better still, as native classes (cf. the notion developed in the work on the bytecode verifier reported in Section 4.5, see also [Cha06].) □

Here is a table of methods with parameter restrictions that are mentioned in the security requirements from chapter 3:

security requirement	restricted method
G-3	methods for image/sound/text processing
G-5,6	java.lang.Class.forName()
G-12–14, C-3	java.lang.System.getProperty()
G-7,G-19–21, G-28, G-37–39, G-41–43, G-56,G-37–39, G-60–62, C-8,12,17,18	javax.microedition.io.Connector.open()
G-22–26,33,34, C-7	javax.wireless.messaging.MessageConnection.newMessage()
G-36	javax.wireless.messaging.MessagePart()
G-51–55	javax.microedition.io.PushRegistry.registerConnection()
G-58	javax.microedition.midlet.MIDlet.platformRequest()
G-19,74	javax.microedition.pim.PIM.openPIMList();
C-20	several PIM methods
G-76–78, C-22	javax.microedition.media.Manager.createPlayer()

5.1.2 Temporal requirements on method calls

The restrictions above concern the value of method parameters at the time of a method call. Other common security requirements are concerned with restricting the possible executions over time.

The most basic example of such a temporal security requirement is one that restricts the history of method parameters, i.e., it describes which operations must have been applied to a method parameter before the method call (for example: the well-formedness of an input must have been validated before it is used, a confidential input to a database must have been encrypted before it is written to the database, or the address parameter for a network connection must not have a substring that has been supplied by the user).

In a more general form, temporal requirements describe sequences of states or events that are (not) supposed to happen, e.g. a sensitive method should not be called between two calls to System.currentTimeMillis (rule G-15), calls to a protected API are done following a given automaton (rule C-6) or the application sends at most or at least a number messages in each session (rule C-9, rule C-11). We can distinguish two groups of requirements:

- requirements that eventually a certain event should happen (or a certain state should be reached), so-called liveness properties; and
- requirements describing which seqeunce of events or states are allowed to happen, so-called safety properties.

Most of the properties mentioned in this document that give temporal requirements on method calls to the API can be considered safety properties. For example, all requirements on the history of input to an API call are safety properties. In addition, rule G-15, 16, 18, 29, 30, 48, 81, rule C-6, 9 and 13 in Chapter 3 are all safety properties on sequences of method calls. Many different existing formalisms that allow to describe such safety properties, e.g.

- security automata [Sch00],
- finite state machines (also known as automata),
- temporal safety properties [Eme90].

Specifications in these formalisms can easily be embedded into JML specifications, making appropriate use of so-called ghost fields. Ghost fields can be declared and assigned to just like ordinary fields, but using special ghost field syntax. Conceptually, one can understand ghost fields by thinking of an imaginary interpreter that treats them just like all other fields. However, ghost fields are specification-only and are ignored at runtime. Ghost fields can be used to "store" parts of the history of objects or executions.

Below we describe with an example how ghost fields can be used to encode safety properties. In general, properties that are described using finite state machines, security automata or temporal safety properties can all be embedded into a JML specification in a similar way, using correspondences between these different specification formalisms. Such translations can be automated, given a precise input format. For example, [HO03] describes a translation of UML state diagrams to JML.

Formalising and verifying liveness properties (such as rule G-11) is more difficult. We refer the interested reader to [BGH+04] for some of our ideas how liveness properties can be specified and verified using JML. However, given that only very few liveness properties have come up in our gathering of security requirements, we have decided not to pursue the formalisation of liveness properties in MOBIUS, and we will not discuss them further here.

Graphical representations of temporal requirements on method calls are often most convenient for the human reader. Indeed, as mentioned in Chapter 3, the Unified Testing Criteria (UTC), version 2.1, [Ini06] now requires a graphical representation – a so-called 'flow diagram' – of the behaviour of an applet, as an aid to the (human) tester. However, the notion of flow diagrams is completely informal and graphical[2]. [Cré06] describes a more formal notion of 'midlet navigation graph': a graph whose nodes abstract the potential screens displayed by the application linked by directed edges labelled with the potential user event that trigger the transition (soft buttons called commands in MIDP, key pressed or pointer events (if available)). The NAVMID tool by France Télécom provides support in drawing such graphs. By means of ghost fields such constrains expressed in graph-form can be expressed in JML, as explained in detail below. At a later stage in the project we might decide to extend existing tools that generate JML, for instance the AutoJML tool developed in Nijmegen [HO03], to cope with the output of the NAVMID tool by the definition of a suitable XML DTD to complete automate the translation from 'flow diagrams' to JML specification.

Below some examples of how ghost fields can be used to encode various safety properties that express constraints on the order in which methods can be called.

Example 5.1.1 (Using a ghost field to track input validation).
In order to store whether a database input has been validated, we can use a ghost field:

```
public class Input –
  ...
  //@ ghost public boolean valid; // a boolean ghost field
  //@ initially valid == false;   // constructors must initialise this
```

[2]In fact, UTC 2.1 requires the flow diagram to be supplied as JPG or GIF.

```
                                // to false

  /*@ ensures valid == true; @*/
  public void validate () throws InvalidException –
    ...
    /*@ set valid = true; @*/          // assign true just before return
  ″
  ...
  // all mutators ensure valid == false
  // the object state is properly encapsulated
  ...
″
```

This specification expresses that after application of validate() an input object is considered valid. To ensure correctness of the specification of method validate(), we have to insert appropriate set-annotations at every possible normal (i.e. exception-free) exit point of the method. Note that here validity means merely that validate() has been applied. The specification says nothing about the functionality of validate(), in particular, it does not say that validate() implements input validation correctly. Still, this specification is useful if the Input class is trusted or its functional correctness has been verified by other means. We can now restrict a non-validating database class to only accept valid input:

```
public class DataBase –
  ...
  /*@ requires in.valid @*/
  public void enter (Input in) –...″
  ...
″
```

Note the comments at the bottom of the Input class: In addition to validating input before database entry, it is just as important to ensure that the input does not mutate after validation. Therefore, mutating methods must reset the ghost field valid to false. Moreover, it is important that Input objects cannot be mutated through aliases that bypass the API, that is, the state of Input objects must be properly encapsulated. In order to express what this means, JML has recently been extended by the Universe ownership type system [DM05]; alias control systems will be investigated in tasks 2.5 and 3.4. The example becomes simpler if the Input class is immutable. We plan to extend JML by immutability specifications and a static rule system for checking object immutability, see section 5.5.

The ghost field Input.valid is subject to the following additional restriction:

- Clients of the Input API must not set the ghost field Input.valid directly.

Without this restriction, hostile client applications could fool a prover by setting in.valid to true right before calling Database.enter(in). The restriction can easily be checked statically. Note that this is a subtle loop-hole that could be used to invalidate any certification; the issue is discussed in Section 5.8. □

Existing APIs often do not package input validation into a single method. To give evidence that the ghost field specification technique is also applicable for more complex requirements,

we now sketch how one can express the requirement PP-4 from the pocket protector example in section 4.1.

Example 5.1.2 (Formalisation of PP-4).
Requirement PP-4 says that all data written to the record store must be encrypted.

```
public class RecordStore –
  ...
  /*@ requires data.encrypted && data.freeze!=null; @*/
  public int addRecord (byte[] data, int offset, int numBytes)
    throws RecordStoreNotOpenException, RecordStoreException,
      RecordStoreFullException –..."
  ...
"
```

The precondition for addRecord makes use of a ghost field data.encrypted, which has a similar purpose as the ghost field in.valid from the previous example.

We require that the entire data array must be encrypted, which is a bit more than needed for prohibiting unencrypted data from entering the record store—it would be enough to require data[offset...offset+numBytes] to be encrypted. We could relax the precondition accordingly, but do not want to complicate the example.

Whereas Input objects from the previous example encapsulate their state, byte arrays are open and can be mutated directly. This is the reason why we need the additional ghost field data.freeze. It gets set to an appropriate non-null reference at the end of the encryption procedure. An invariant on byte arrays ensures that the data array does not mutate unless data.freeze is null or data.freeze gets reset. Because we require that applications do not reset the freeze ghost field directly, a postcondition data.freeze!=null on an API method ensures that data does not get mutated between a call to this method and a succeeding API call.

There is no byte array class that could hold the declarations for the ghost fields encrypted and freeze, so we have to declare them in the Object class:

```
public class Object –
  ...
  /*@ ghost public boolean encrypted;
    @ initially encrypted==false; @*/

  /*@ ghost public String freeze;
    @ initially freeze==null;
    @ invariant !(this instanceof byte[])
    @         —— freeze==null                  // this may be mutated
    @         —— new String(this)==freeze; @*/  // this is frozen
  ...
"
```

The invariant requires that a non-null freeze string is equal to the string representation of this. Because we disallow applications to reset freeze, it is impossible for applications to mutate this without breaking the invariant when freeze!=null.

The pocket protector example uses the cipher engine PaddedBufferedBlockCipher from the Bouncy Castle cryptography API [otBC]. Encryption is achieved in a piecewise fashion using

the methods processBytes() and doFinal(). The former buffers and/or encrypts the given input bytes, the latter is intended to be applied at the very end of the encryption protocol in order to encrypt the last remaining bits left in the buffer and add the final padding. This API architecture allows clients to efficiently encrypt the concatenation of several input components. For instance, the pocket protector encrypts the concatenation of three components: a nonce, a message digest, and the actual plaintext record. It achieves this by calling processBytes() three times and then doFinal() once.

Here are all methods from the PaddedBufferedBlockCipher engine that the pocket protector uses:

- void init (boolean forEncryption, CipherParameters params)
  - Initialises the cipher engine.
  - forEncryption - if true the cipher is initialised for encryption, if false for decryption.
- int getOutputSize (int len)
  - Returns the minimum size of the output buffer required for processing (including the doFinal()) an input of len bytes.
- int processBytes (byte[] in, int inOff, int len, byte[] out, int outOff)
  - Processes len input bytes, producing output if necessary.
  - Returns the number of output bytes copied to out.
- int doFinal (byte[] out, int outOff)
  - Processes the last block in the buffer.
  - Returns the number of output bytes copied to out.

A typical usage protocol for this cipher engine goes like this: after initialisation, first compute the total length of the plaintext to be encrypted/decrypted and then use getOutputSize() to compute the required length for the output array. Next, create an output array of this length. Fill this array by repeatedly calling processBytes(). Finally, call doFinal() when all input has been processed. The following state diagram depicts the sequence of method calls for this protocol:

The pocket protector uses exactly this protocol. There are some other methods in the PaddedBufferedBlockCipher, which the pocket protector does not use. For this example, let us assume that these other methods are forbidden. In reality, it would probably be better to extend the protocol, allowing the additional methods in appropriate ways.

The state diagram alone is not a complete specification of the usage protocol. It only specifies legal orderings of methods calls. In addition, it is also important that the methods are called with compatible parameters. For instance, it is important that two calls to processBytes() have the same out parameter, unless doFinal() has been called in between. Or, it is important that the offset parameters outOff are incremented correctly. We use ghost fields in order to make the usage protocol precise.

public class PaddedBufferedBlockCipher –

```
// CIPHER MODE CONSTANTS:
/*@ ghost public static final int ENC;     // encryption mode
  @ ghost public static final int DEC;     // decryption mode
  @ ghost public static final int UN;      // uninitialized
  @ invariant ENC!=DEC && ENC!=UN && DEC!=UN; @*/

// THE GHOST STATE:
/*@ ghost public int mode,remInLen,fullOutLen,curOutOff;
  @ ghost public byte[] curOut; @*/

// THE INITIAL GHOST STATE:
/*@ initially mode==UN && remInLen==0
  @        && curOut==null && fullOutLen==0 && curOutOff==0; @*/
...
"
```

The ghost field curOut stores the output array that is currently processed. The ghost fields remInLen, fullOutLen and curOutOff store the remaining input length, full output length and current output offset, respectively. Each state in the state diagram is associated with an invariant on the ghost state:

state	invariant
s_0	mode==UN && curOut==null && remInLen==0
s_1	mode!=UN && curOut==null && remInLen==0
s_2	mode!=UN && curOut==null && remInLen¿0
s_3	mode!=UN && curOut!=null

Note that the invariants for the different states do not overlap, i.e., at most one of these is satisfied. We now use the following recipe to derive the pre- and postconditions for the cipher methods:

- Preconditions: Use the state invariants to ensure that a method is never called from a bad state.
- Preconditions: Use parameter restrictions to ensure that a method is always called with correct parameters (for instance, with the correct offset parameter or with the correct output array).
- Postconditions: Describe how the ghost state is transformed.
- Postconditions: Set out.freeze to non-null in the postconditions of processBytes() and doFinal().
- Postconditions: If mode==ENC, set out.encrypted to true in the postcondition of doFinal().

Figure 5.1 shows the specifications for the cipher methods. The ghost fields are subject to the following additional restriction:

- Client applications must not set ghost fields from the PaddedBufferedBlockCipher class, nor the ghost fields Object.freeze and Object.encrypted.

The restriction can easily be checked statically. This completes the formal specification of security requirement PP-4 for the pocket protector. □

```
/*@ requires
  @    curOut==null && remInLen==0;
  @ ensures
  @    mode==forEncryption?ENC:DEC && remInLen==“old(remInLen) &&
  @    curOut==“old(curOut) && fullOutLen==“old(fulOutLen) &&
  @    curOutOff==“old(curOutOff); @*/
public void init (boolean forEncryption, CipherParameters params)
  throws IllegalArgumentException –...″

/*@ requires
  @    mode!=UN && curOut==null && remInLen==0 && len¿0;
  @ ensures
  @    mode==“old(mode) && remInLen==len &&
  @    curOut==“old(curOut) && fullOutLen==“result &&
  @    curOutOff==“old(curOutOff); @*/
public int getOutputSize(int len) –...″

/*@ requires
  @    mode!=UN && curOut==null && remInLen¿0 && outOff==curOutOff &&
  @    out.length==fullOutLen;
  @ ensures
  @    mode==“old(mode) && remInLen==“old(remInLen)-len &&
  @    curOut==out && fullOutLen==“old(fullOutLen) &&
  @    curOutOff==“old(curOutOff)+outOff && curOut.freeze!=null;
  @
  @ also
  @
  @ requires
  @    mode!=UN && curOut!=null && remInLen¿0 && outOff==curOutOff &&
  @    out==curOut;
  @ ensures
  @    mode==“old(mode) && remInLen==“old(remInLen)-len &&
  @    curOut==“old(curOutOff) && fullOutLen==“old(fullOutLen) &&
  @    curOutOff==“old(curOutOff)+outOff && curOut.freeze!=null; @*/
public int processBytes (byte[] in, int inOff, int len, byte[] out, int outOff)
  throws DataLengthException, IllegalStateException –...″

/*@ requires
  @    mode!=UN && curOut!=null && remInLen==0 && outOff==curOutOff &&
  @    out==curOut;
  @ ensures
  @    mode==“old(mode) && remInLen==0 && curOut==null && fullOutLen==0 &&
  @    curOutOff==0 && out.freeze!=null && (mode==DEC —— out.encrypted); @*/
public int doFinal(byte[] out, int outOff)
  throws DataLengthException, IllegalStateException,
    InvalidCipherTextException –...″
```

Figure 5.1: The usage protocol for PaddedBufferedBlockCipher discussed in Example 5.1.2

The two examples above show how data flow properties can be expressed in JML by using a ghost field that for some object (an input object and a byte array, resp.) describe the 'origin' of the information it contains. An alternative way for formalising data flow properties is by declaring some global sets of objects that keep track of their origin.

Example 5.1.3 (Alternative formalisation of PP-4).
For the formalisation of PP-4 we want to distinguish byte arrays that have been encrypted from those that have not been. The specification below provides a way to do this:

```
public class PaddedBufferedBlockCipher –
 //@ final static ghost SetOfStrings ENCRYPTED;

 //@ requires ...
 //@ ensures  ENCRYPTED.contains(new String(out))
 public int doFinal(byte[] out, int outOff)

"
```

We use Strings, because these are immutable, as opposed to array contents, which makes them more convenient to work with.[3] The specification above declares a set of encrypted strings ENCRYPTED, and the postcondition of doFinal says that the contents of the byte array out, seen as a string (using the constructor String(byte[] bytes)), is in this set.

Given this specification for the PaddedBufferedBlockCipher class, we can now give a succinct specification that only data produced form the encryption mechanism can be written in the record store.

```
public class RecordStore –
  ...
  //@ requires ENCRYPTED.contains(new String(data));
  public int addRecord (byte[] data, int offset, int numBytes)

"
```

□

5.2 Exceptional control flow

Failure to handle errors correctly, or, worse, failure to handle errors at all, is a notorious source of security flaws, as discussed for instance in [GvW03, HlV05]. This is a generic flaw, independent of the programming language used, but of course, the risks are different for languages with an exception handling mechanism, such as Java – that we study here – and those without, such as C, where one has to use a special return value to signal errors.

Failure to handle errors correctly can have different serious consequences, e.g. it may enable a denial-of-service attack, because an application could fail to release some resource it had claimed when the exception occurred, or it may leak confidential information, either

[3] Here we just assume some type SetOfStrings for immutable sets of Strings. The most convenient way to introduce and reason about such sets would be using the native keyword introduced in the course of the work on the bytecode verifier reported in Section 4.5, see also [Cha06].

because of the exceptional control flow or because it fails to complete an access control check upon the occurrence of the exception.

Since we are working in the context of a language with an exception handling mechanism, error handling boils down to the appropriate handling of exceptional control flow. Exceptional control flow is important as an auxiliary means to ensure cumulative requirements on method calls (as discussed above). Consider for example requirements rule C-9 (the application sends no more than a number messages in each session), and rule C-11 (the application sends at least a number messages in each session). Suppose that sending a message is done by a method that first makes some checks on the message format and throws an exception if these fail. In this case, the requirements would impose restrictions on the number of times the sending method may successfully execute, but an exceptional termination should not be considered as a successful send.

Requirements on the exceptional control flow can easily be specified in JML. Two explicit specification clauses are available for this: signals and signals˙only. A clause signals (E e) P specifies that if the precondition of the method is met and it terminates because of an exception that is a subclass of the exception type E, then the predicate P should hold (the variable e can be used to refer back to the thrown exception in the predicate P). Thus, this clause specifies under which conditions a certain exception may be thrown. A specification signals˙only E1 E2 ... En specifies that if a method meets it precondition, then it either terminates normally, or it terminates because of one of the exceptions listed (in fact, a signals˙only clause can be desugared into a signals clause). In addition, one can use in JML an exceptional behaviour specification, which states that the method must terminate because of an exception.

The most common security requirement regarding exceptional control flow is in fact the requirement that no exceptions arise at all. For example, requirements such as rule G-7 and rule G-10 can be guaranteed by ensuring that the application will not terminate because of any (unhandled) security exceptions. This can be specified by writing the following specification for the application:

```
//@ signals (SecurityException) false;
```

More generally, security properties that can be enforced by some dynamic checks can often be expressed as absence of runtime exceptions. An example is rule rule G-65, where a dynamic check on the amount of persistent memory used throws an exception of the maximum amount specified by some policy is exceeded. Note that this provides a general way to turn dynamic check at runtime into a static check that can be verified at compile-time – or rather, certification-time.

In Deliverable D1.1 (Section 3.3.3) the following rule is given as a means to avoid control flow misuse:

> 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.

A very simple approach to ensure this rule would be to specify that never any method should terminate with an exception. However, as it might not be always possible to directly

catch and handle exceptions, a more feasible approach would be to specify as precisely as possible which exceptions can be thrown (using the signals˙only clause, and for each exception under which conditions it might be thrown (using the signals clause). In this way, the static verification tools can show that each possible exception eventually will be caught and appropriately handled.

Finally, it should be mentioned that, using weakest precondition-like techniques, in many cases it is possible to compute automatically the necessary preconditions such that a method will not throw an exception. This is in particular useful to ensure the absence of runtime exceptions, such as throwing a NullPointerException or an ArrayIndexOutOfBoundsException. Generation of annotations to ensure the absence of runtime exceptions will be one of the topics of Task 3.7.

5.3 Resource usage

Security requirements on resource usage have already been extensively discussed in deliverable D1.1, especially in Chapter 5, with approaches to formalisation being discussed in Section 5.3.

In this deliverable, more concrete examples of security requirements on resource usage are given, in chapters 3 and 4, for example rule G-1, rule G-3, rule C-4, rule C-5, rule G-22 ... rule G-27, rule C-7, rule C-9 ... rule C-11 rule C-18, rule G-64-rule G-65 rule G-68, rule G-76, rule G-77 in Chapter 3, and PP1-PP5 in Chapter 4.

All of these requirements concern system resources that are used via special API calls, so these requirements fall under the category of API usage discussed in Section 5.1. All of these requirements can be expressed in BML/JML. Some can be expressed using simple preconditions. Others in addition require the standard specification pattern of introducing a so-called ghost field – a global auxiliary and specification-only variable – to record the usage of a given resource (e.g. number of threads started, numbers of SMS sent, etc.). More on this specification pattern in Section 5.1.2.

This technique has been tried out to specify and verify memory allocation, even though this is not a resource that is accessed via API calls, but one that is accessed via the Java language itself. This approach is described in detail in [BPS05], and progress on supporting the special JML keywords for this is discussed in [Atk06].

It was already foreseen that requirements about resource usage might be better suited to be expressed and ultimately certified through dedicated type systems, hence the work on type systems for resource usage in Task 2.3 and 2.4. It was also foreseen that enforcement via logic-based approached would require custom extension of the base BML logic developed in Task 3.1, hence the work on extension of the base logic to express properties about resource usage in Task 3.2.

5.4 Data flow

Somewhat to our surprise, it turned out that some data flow requirements – albeit of a restricted form, namely explicit information flow requirements – can be expressed in JML. We noticed this when investigating possible ways to formalise some of the framework-specific security requirements given in Chapter 3, more in particular the constraints on the history of arguments to API call discussed in Section 5.1.2.

Two ways to capture basic explicit information flow constraints in JML are

- by attaching a ghost field to data objects, as done in examples 5.1.1 and 5.1.2
- by declaring a ghost set of legal inputs, as done in Example 5.1.3

The possibilities and limitations of the latter approach are discussed in detail below. As demonstrated in Example 5.1.3, it is relatively easy to express requirements of the pattern

"All data passed to output channel OUT comes from input channel IN"

where data is passed to some output channel by feeding it as argument to some API call, and data is obtained from some input channel as a result of some API call.

To give a concrete example, suppose we have an integer input channel provided by some static methods IN.read() and a integer output channel provided by a static method OUT.write(int). Then the requirement above can be formalised as follows

```
class IN–
 //@ public ghost final static IntegerSet ALLOWED;

 //@ ensures ALLOWED.contains(“result);
 public static int read();

”


class OUT–
  //@ requires ALLOWED.contains(i);
  public static void write(int i);

”
```

In example 5.1.3 this pattern is used to express the security requirements that all data passed as input to some API method must have been passed through an encryption procedure. Another example of a security requirement that could be expressed would be that all data used as phone numbers, e.g. all data passed as telephone number to a method that dials a number or stores a number in a phone book, must either be user input or obtained from the phone book, thus capturing the security requirement that phone numbers should not be computed, mentioned in Deliverable 1.1.

In the example above we use a specification-only set ALLOWED to express the explicit information flow constraint. The same approach is used in Example 5.1.3. For explicit information flow constraints of objects, as opposed to primitive values, an alternative is to use a ghost field to track if an object is (not) allowed as input to some method, as described in examples 5.1.1 and 5.1.2. We have carried out a more substantial case study in using ghost fields to specify and verify explicit information flow constraints in [SC06].

Remark 5.4.1 (Explicit vs implicit information flow).
Note that the pattern above expresses a restriction on explicit information flow rather than implicit information flow: even if it prevents data from some input channel being passed on to some output channel (either directly or in some processed form), it does not prevent information about this data from leaking.

For example, suppose we have boolean input channels provided by static integer methods Public.read() and Confidential.read() and a integer output channel provided by a static method Out.write(integer). Then the program fragment

```
lo = Public.read();
hi = Confidential.read();
if (h)
    Out.write(lo); Out.write(lo);
 ″ else –
    Out.write(lo); Out.write(lo-1);
```

leaks information from channel Confidential to Out, even though it only passes on information from channel Public to OUT.

(Absence of) implicit information flow cannot be specified by simple program assertions as JML provides, or indeed Hoare logics. (In [War06] it is shown that absence of implicit information flow can be specified in JML, using the approach of [JL00], but this rapidly becomes extremely involved.) Instead, dedicated method for information flow using type-systems, as investigated in Tasks 2.1 and 2.2, or more powerful logics, notably dynamic logic, as investigated in Task 3.2, seem better-suited; [DHS05] shows implicit information flow can be expressed in dynamic logic. □

The pattern can be generalised, for instance, for multiple input and output channels, or to allow combinations of data, e.g. saying that strings from some input channel may/must be prefixed with some fixed string before being sent to some output channel, or saying that concatenations of strings obtained from some input channels may be sent to some output channel, etc. More generally,

> All data passed to output channel OUT comes from some set that inductively defined in terms of inputs that come given input channels. (IN-OUT-INDUCTIVE)

for any inductive construction that we can express.

Remark 5.4.2 (Positive vs negative data flow requirements).
A limitation is that we can express 'positive' requirements on data flow, i.e. requirements that say that data sent to some output channel must be constructed in a specified way from certain input channels, but not 'negative' requirements on data flow, i.e . properties say that some data sent to an output channel does not come from (or is not constructed from) certain input channels. In other words, we can specify some data flows that we allowed, but not data flows that we want to forbid.

It may be possible to express 'negative' requirements in a 'positive' way, by giving an inductive characterisation of the complement of the set we want to forbid as outputs. However, this may be very unpractical, when this would requires a very complicated inductively defined set. □

5.5 Immutability

The notion of an object being immutable comes up several times in the earlier discussion of security requirements. It is sometimes used to express security requirements. For instance, rule J-1 and rule J-3 in Chapter 2 mention (im)mutability of objects.

Even when security requirements do not explicitly mention (im)mutability, it is a useful property to exploit in verification. For example, as discussed in Section 3.15.1, immutability of Strings vs. mutability of StringBuffers make a big difference on ensuring some security requirements. Especially in a multi-threaded context, verification may not be feasible or even possible if we do not exploit information about objects being immutable.

For platform components, it will be crucial to certify that certain classes are immutable. Examples of immutable API classes include String and (subclasses of) Permission, Key, javax.microedition.pki.Certificate, and javax.microedition.io.SecurityInfo.
The native classes proposed in Section 4.5 and [Cha06] are also typically (always?) immutable.

It is currently not possible to specify object immutability in JML, although [Cok05] already discusses how it can be exploited in program verification. As part of the MOBIUS project, we will extend JML with an immutable class modifier. Defining the semantics of this modifier and verifying it is more difficult than it may seem, esp. in a multi-threaded context. Indeed, it is only thanks to the recent revision of the Java Memory Model [JSR04a, ?] that immutability of strings is guaranteed.

Work on proposing a definition of object immutability for JML, and tool support for enforcing and exploiting it is continuing in Task 3.3, and will be reported on in the deliverables for that task.

5.6 Concurrency

Several framework-specific security requirements are about concurrency aspects, for example rule G-16, rule G-17, rule G-18, rule C-4, rule C-5, on page 21. These properties are about certain API calls being in a single thread or 'properly synchronised', the absence of deadlock, thread-safety, and the number of threads an application uses.

Even when security requirements do not explicitly mention concurrency, concurrency often creates major complications for reasoning about programs and certifying security requirements. For instance, ensuring a particular order on calls to certain API methods (rule C-6) is complicated if several threads call these methods; indeed, rule G-18 aims to avoid this complication. Similarly, thread-safety rule G-17 is mentioned as a prerequisite for checking more complex requirements.

Many of the properties above require new primitives to be added to JML and BML. Some, for instance a keyword "thread˙safe, have recently been proposed in [?]. Work on defining a rigorous formal semantics of these notions and associated proof rules is continuing as part of Task 3.3, and will be reported on in the deliverables for that task.

Some of the specific requirements about threads can already be specified in JML or BML, using the API methods in java.lang.Thread. For instance, the requirement that certain API method apimethod is only done in a particular thread can be expressed by means of a precondition for that API call of the form

```
//@ requires java.lang.Thread.currentThread()==THREAD˙ONE;
public void apimethod(...)
```

where THREAD˙ONE is a constant reference to the single thread that may call the method[4], which could be declared a final static ghost field. However, even though specification of such

[4]Note that this is another example where we need API contract that are tailor-made for a specific application.

a property is possible in JML without further extensions, verification of such a property is not possible without extensions of the logical theories used by verification tools, in this case, with knowledge about the Thread class and the methods it provides.

5.7 Object visibility and alias control

The general rule in management of the security sensitive data is to make sure that the data is visible only in places where it is used. This ensures that confidential data is not leaked in a uncontrolled way by a far piece of code and that the integrity of data is not infringed by a far piece of code. In fact, many security holes are the result of improper management of data visibility or because it is possible to make certain data modifications in place where they should not happen. This leads to security requirements like rule J-1 for the Java programming language according to which all the fields that contain mutable data should be declared as private. Such restrictions reduce chances that bugs in our code or a malicious extension of the code can expose the assets protected by the application we develop. This requirement, however, is not enough as mutable objects can be shared by means of aliasing (e.g. a constructor can set a private field to one of its parameters and thus share field with another object). That is why we need in the context of security sensitive applications additional formalisms for alias control.

The proper visibility management is especially important in case of multithreaded applications where we can avoid data races by making some data manipulated in one thread invisible by another thread. Moreover, the multithreaded code potentially exposes data in the application to intruders in more fain-grained fashion, i.e. the intruders can use a side thread to manipulate or read data in the main thread.

The basic way to specify the visibility is to use the private Java keyword together with one of many alias control type systems, e.g. the universes type system [DM05] or the flexible alias protection system [NVP98]. These type systems will be developed during the course of the MOBIUS project in Task 2.5 Alias Control Types.

5.8 Towards certification using JML/BML

This section discusses some issues involved in using verification of security properties expressed formally in JML/BML as a basis for PCC. These are obvious, but should not be overlooked!

Some security properties for an application can be formalised as a contract for an external API that the application uses. Certifying then requires verification of the application with respect to such an API specification.

Note that a fundamental and unavoidable complication is that even a simple security requirement might be formalised as quite a complicated API specification. For instance, as discussed in page 46, a simple rule that say that all network connections must be HTTP or HTTPS connections translations into contracts for a few API methods. Writing such API contracts, or understanding what these contracts express, requires expertise and thorough knowledge about the APIs used. Even though in the PCC setting code consumers can check for themselves that code obeys certain requirements, code consumers cannot be expected to understand these formal requirements, and will have to trust some third parties that these requirements correctly formalise what they want them to mean.

In general we may want to impose different requirements on different applications, in which case we would need different API specifications for different applications. The certificate for an application should then contain information about the API specification.

Verification of an application will typically rely on additional properties of the application code, e.g. invariants that it maintains. These properties will also be needed for certification, probably as BML annotations in the bytecode.

As part of the certification procedure we may have to do some sanity check on such annotations in the application, to make sure these annotations do make assumptions that may not be true or may be invalidated. For example, an application could trivially meet any security requirement by imposing a precondition //@requires false; on the methods that constitute its interface to the outside world. We should check that the application does not make any assumptions on its usage, i.e. that it only assumes preconditions //@requires true; on externally accessible methods. Similarly, any invariants that the application maintains, and that it relies on to meet its contract, should be impossible to break by direct access to its data-structure, i.e. access not via its methods, but via direct access to its fields or exposed internal data-structures. Here the issues raised in Chapter 2 apply.

Some security properties for an application can be specified by means of specifications inside the applications, as opposed to specification on some external API. An example is the specification that some argument is 'determined', as discussed on page 79, by means of an assert statement of a very specific form directly prior to some method call. Here it is even more obvious that as part of the certification procedure we should check that the application is annotated in the right places with the right specifications.

For security properties whose formalisation involves ghost variables, our certification should rule out illegal (write) access to these ghost variables. At present JML/BML does not provide scoping/visibility mechanism to restrict access to a ghost variable to read-only for some component. Malicious code might try to set values of ghost fields used in other packages to express security requirements, making any subsequent guarantees void. One solution would be to add such a scoping/visibility mechanism to JML and BML. Alternatively, one might check the absence of illegal access separately, as part of the certification procedure, in the same way as the checks discussed above. Yet another solution would be to make the ghost field private, and to add model methods (i.e. specification-only methods) for getting and setting the ghost field with different visibilities. This is already possible in JML, but BML might not include model methods, at least not the initial version.

In all, the problem of preventing illegal write accesses to ghost fields can be solved. It is not a priority to fix a particular solution, or indeed develop support for the solution, e.g. in BML or any of the tools, as long as we are aware that the problem exists.

Appendix: List of acronyms

- AMS - Application Management Software
- BML - Bytecode Modeling Language
- BCV - Byte Code Verifier
- CLDC - Connected Limited Device Configuration
- FSM - Finite State Machine
- GUI - Graphical User Interface
- IrDA - Infrared Data Association
- JML - Java Modeling Language
- JTWI - Java Technology for the Wireless Industry
- J2ME - Java 2 Micro Edition
- MIDP - Mobile Information Device Profile
- PCC - Proof Carrying Code
- PIM - Personal Information Management
- RMS - Record Management Service
- SMS - Short Message Service

Bibliography

[Atk06] Robert Atkey. Specifying and verifying heap space allocation with JML and ESC/Java2. In Workshop on Formal Techniques for Java Programs, 2006.

[BDJ06] Fréderic Besson, Guillaume Dufay, and Thomas Jensen. A formal model of access control for mobile interactive devices. In Computer Security — ESORICS 2006, Proceedings of the 11th European Symposium on Research in Computer Security, number 4189 in Lecture Notes in Computer Science. Springer-Verlag, 2006.

[BGH+04] F. Bellegarde, J. Groslambert, M. Huisman, O. Kouchnarenko, and J. Julliand. Verification of liveness properties with JML. Technical Report RR-5331, INRIA, 2004.

[BP06] L. Burdy and M. Pavlova. Java bytecode specification and verification. In Symposium on Applied Computing, pages 1835–1839. ACM Press, 2006.

[BPS05] G. Barthe, M. Pavlova, and G. Schneider. Precise analysis of memory consumption using program logics. In Software Engineering and Formal Methods, pages 86–95. IEEE Press, 2005.

[CA05] P. Crégut and C. Alvarado. Improving the security of downloadable Java applications with static analysis. In Bytecode Semantics, Verification, Analysis and Transformation, volume 141 of Electronic Notes in Theoretical Computer Science. Elsevier, 2005.

[Cha06] J. Charles. Adding native specifications to JML. In Workshop on Formal Techniques for Java Programs, 2006.

[CK05] D. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using esc/java2 and a report on a case study involving the use of esc/java2 to verify portions of an internet voting tally system. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, Construction and Analysis of Safe, Secure and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004, volume 3362 of Lecture Notes in Computer Science, pages 108–128. Springer-Verlag, 2005.

[Cok05] D. R. Cok. Reasoning with specifications containing method calls in JML. Journal of Object Technology, 4(8):77–103, 2005.

[Cré06] Pierre Crégut. Midlet navigation graphs. Unpublished draft, 2006.

[DA99] T. Dierks and C. Allen. The TLS protocol: Version 1.0. RFC 2246, The Internet Society, January 1999.

[DHS05] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, Security in Pervasive Computing, volume 3450 of Lecture Notes in Computer Science, pages 193–209. Springer-Verlag, 2005.

[DM05] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. Journal of Object Technology, 4(8):5–32, October 2005.

[DSTZ06] M. Debbabi, M. Saleh, C. Talhi, and S. Zhioua. Security evaluation of J2ME CLDC embedded Java platform. Journal of Object Technology, 5(2):125–153, 2006.

[Duf03] G. Dufay. Vérification formelle de la plate-forme Java Card. PhD thesis, Université de Nice Sophia-Antipolis, December 2003.

[Eme90] E. A. Emerson. Temporal and Modal Logic, chapter 16. Elsevier, 1990.

[FKK96] A. Frier, P. Karlton, and P. Kocher. The SSL 3.0 protocol. Technical report, Netscape Communications Corp., November 1996.

[GJSB05a] J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification, third edition. The Java Series. Addison-Wesley, 2005.

[GJSB05b] J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification, Third Edition, chapter 17. Sun Microsystems, 2005. http://java.sun.com/docs/books/jls/.

[Gro03] Phenolite Hackers Group, 2003. http://www.phenolite.de.

[GvW03] Mark G. Graaf and K. R. van Wyk. Secure Coding: Principles and Practices. O'Reilly & Associates, Inc., 2003.

[HlV05] M. Howard, D. leBlanc, and J. Viega. 19 Deadly Sins of Software Security. McGraw-Hill, 2005.

[HO03] E. Hubbers and M. Oostdijk. Generating JML specifications from UML state diagrams. In Forum on specification & Design Languages, pages 263–273. University of Frankfurt, 2003. Proceedings appeared as CD-Rom with ISSN 1636-9874.

[Ini06] Unified Testing Initiative. Unified testing criteria for Java technology-based applications for mobile devices. Technical report, Sun Microsystems, Motorola, Nokia, Siemens, Sony Ericsson, May 2006. Version 2.1.

[JL00] R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. Science of Computer Programming, 37(1-3):113–138, 2000.

[JSR00a] JSR 30 Expert Group. Connected limited device configuration (CLDC), version 1.0. Java specification request, Java Community Process, 2000.

[JSR00b] JSR 37 Expert Group. Mobile information device profile (MIDP), version 1.0. Java specification request, Java Community Process, 2000.

[JSR02a] JSR 118 Expert Group. Mobile information device profile (MIDP), version 2.0. Java specification request, Java Community Process, November 2002.

[JSR02b] JSR 218 Expert Group. Connected limited device configuration (CLDC), version 1.1. Java specification request, Java Community Process, 2002.

[JSR03a] JSR 120 Expert Group. Wireless messaging API. Java specification request, Java Community Process, 2003.

[JSR03b] JSR 184 Expert Group. Mobile 3D graphics API for J2ME. Java specification request, Java Community Process, September 2003.

[JSR03c] JSR 185 Expert Group. Java technology for the wireless industry (JTWI), release 1. Java specification request, Java Community Process, 2003.

[JSR03d] JSR 205 Expert Group. Wireless messaging API (version 2.0). Java specification request, Java Community Process, June 2003.

[JSR04a] JSR 133: Java memory model and thread specification, 2004.

[JSR04b] JSR 172 Expert Group. J2ME web services specification, version 1.0. Java specification request, Java Community Process, March 2004.

[JSR04c] JSR 177 Expert Group. Security and trust services API for J2ME. Java specification request, Java Community Process, September 2004. Final release.

[JSR04d] JSR 75 Expert Group. PDA optional packages for the J2ME platform. Java specification request, Java Community Process, June 2004. Final release.

[JSR05a] JSR 211 Expert Group. Content handler API, version 1.0. Java specification request, Java Community Process, June 2005.

[JSR05b] JSR 226 Expert Group. Scalable 2D vector graphics for J2ME, version 1.0. Java specification request, Java Community Process, March 2005.

[JSR05c] JSR 248 Expert Group. Mobile service architecture (MSA). Java specification request, Java Community Process, 2005. Early draft.

[KJJ99] P. Kocher, J. J., and B. Jun. Differential power analysis. In Advances in Cryptology, volume 1666 of Lecture Notes in Computer Science, 1999.

[KYW02] C. K. Koc, S. S. Yerubandi, and W. Wanalertlak. SSH1 man in the middle attack. Technical report, Oregon State University, 2002. http://islab.oregonstate.edu/koc/ece478/02Report/YW.pdf.

[Leh06] S. Lehtinen. The secure shell (SSH) protocol assigned numbers. RFC 4250, The Internet Society, January 2006.

[Ler03] X. Leroy. Java bytecode verification: algorithms and formalizations. Journal of Automated Reasoning, 30(3-4):235–269, December 2003.

[LPC+05] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. JML Reference Manual, July 2005. In Progress. Department of Computer Science, Iowa State University. Available from http://www.jmlspecs.org.

[MF99] G. McGraw and E. Felten. Securing Java. Wiley, 1999. Available online at www.securingjava.org.

[MM01] R. Marlet and D. Le Metayer. Security properties and Java Card specificities to be studied in the SecSafe project. Technical Report SECSAFE-TL-006, Trusted Logic, August 2001. Available from http://www.doc.ic.ac.uk/~siveroni/secsafe/docs.html.

[NVP98] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In Eric Jul, editor, European Conference on Object-Oriented Programming, number 1445 in Lecture Notes in Computer Science, pages 158–185. Springer-Verlag, 1998.

[otBC] The Legion of the Bouncy Castle. Bouncy castle cryptography library 1.33. http://www.bouncycastle.org/docs/docs1.5/, date retrieved June 5, 2006.

[SA03] S. Skorobogatov and R. Anderson. Optical fault induction attacks. In Cryptographic Hardware and Embedded Systems, pages 2–12, London, UK, 2003. Springer-Verlag.

[SC06] A. Schubert and J. Chrzszcz. ESC/Java2 as a tool to ensure security in the source code of Java applications. In Software Engineering Techniques: Design for Quality, IFIP, Warsaw, 2006. Springer-Verlag.

[SCC+06] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java™ on the bare metal of wireless sensor devices: the squawk java virtual machine. In Virtual execution environments, pages 78–88, New York, NY, USA, 2006. ACM Press.

[Sch00] F. B. Schneider. Enforceable security policies. ACM Transactions on Information and System Security, 3:30–50, 2000.

[SMH05] K. I. Fagerland Simonsen, V. Moen, and K. Jargen Hole. Attack on sun's MIDP reference implementation of SSL. In Helger Lipmaa and Dieter Gollman, editors, Nordic Workshop on Secure IT Systems, 2005. Available at http://www.fi.muni.cz/~xvyborny/tartu/nordsec.

[Sof06] Fortify Software. Fortify application defense data sheet, 2006. Available from http://www.fortifysoftware.com/products/ad.jsp, login required.

[TH02] K. Trentelman and M. Huisman. Extending JML specifications with temporal logic. In Algebraic Methodology and Software Technology, volume 2422 of Lecture Notes in Computer Science, pages 334–348. Springer-Verlag, 2002.

[War06] M. Warnier. Language Based Security for Java and JML. PhD thesis, Radboud Universiteit Nijmegen, 2006. To appear.

[WPSJ05] B. De Win, F. Piessens, J. Smans, and W. Joosen. Towards a unifying view on security contracts. In Workshop on Software Engineering For Secure Systems, pages 1–7. ACM, 2005.

[Ylo06] T. Ylonen. The secure shell (SSH) protocol architecture. RFC 4251, The Internet Society, January 2006.