Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science 141 (2005) 129–144
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Improving the Security of Downloadable Java Applications With Static Analysis

Pierre Crégut [1] Cuihtlauac Alvarado [1]

France Télécom - Recherche & Développement
2 avenue Pierre Marzin, 22300 Lannion, FRANCE

Abstract

Today, most middle-end mobile phones embed a Java runtime environment that can execute programs downloaded on the network by the user. This new functionality creates great opportunities for new services but also brings the full range of risks that existed on the personal computer to the phone.
Telecommunication operators are the last warrant of the quality of the software downloaded by their customers and might sign the applications they trust. Unfortunately they have little evidence to check the quality of the contents of the jammed bytecode they receive from developers. The traditional evaluation process relies mostly on the manual testing of the software on actual terminals. But this is not adapted for security properties.
MATOS (Midlet Analysis TOol Suite) is a static analysis tool that checks the possible values passed to some identified methods directly on the compiled application. It is used by the test teams of the mobile operator Orange to check what kind of connections are opened by MIDP applications. We will present the security requirements we want to check, how MATOS helps to ensure them and how the necessary analysis are performed using a combination of (rather) well-known analysis techniques.

Keywords: Static analysis, bytecode, mobiles.

1 Java downloadable applications

1.1 Execution environments for downloadable applications

Most modern mobile phones offer a mechanism to download applications from the network. The ability to download applications and the capacity of the

[1] Email: `firstname.lastname@francetelecom.com`

doi:10.1016/j.entcs.2005.02.033

handsets to exchange data with various protocols such as SMS, GPRS or Bluetooth extend the possible usages of the mobile phone (e-commerce, database front-ends, games, etc.) but they also create new security risks that are similar to the ones already present on personal computers.

During summer 2004, the first proof-of-concept worm on mobile phone named *cabir* was revealed. It targeted the *Symbian* operating system and it was later followed by similar viruses, developed for *Windows Mobile* operating system from Microsoft.

Java offers an interesting alternative to open operating systems because it provides a software "sandbox" that prevents the anarchic use of system resources: the information flow and the control flow are bounded by the Java type system and when an application tries to access a resource in a way forbidden by the security policy, a runtime security check will abort the execution of the program. Medium and high-end mobile handsets embed a Java runtime environment with a set of specialized libraries for small mobile devices (called a profile). MIDP, the profile defined within JCP process[2], and DoJa, the proprietary profile from NTT DoCoMo, are the most common environments.

A Java configuration defines the expressive power of the language. CLDC [8,7] is one of the lightest version: CLDC programs are real applications using dynamic allocation of objects or threads but they cannot use programmatic reflection[3] or interfaces to native libraries or the dynamic downloading of Java code.

A profile is a standardized set of libraries targeted to an application area. The MIDP [9,6] profile is used on top of the CLDC configuration and defines a small graphical user interface suitable for phone handsets, a generic framework to establish connections and a small database to manage persistent data. Some standardized extensions (eg. MMAPI or WMA) provide support for more complex multimedia data and network protocols.

MIDP also defines the life cycle of an application. An application (also called a MIDlet suite) is made of a descriptor file (also called the JAD file) and a standard Java archive file (called the JAR file) that contains the whole code of the application. The descriptor file contains one or possibly several entry points for the applications (the MIDlets). Each MIDlet is implemented by a class that inherits from an abstract class used as a template: `javax.microedition.midlet.MIDlet` and it must define some special callbacks for creating, launching and stopping the application.

[2] The standardization of Java led by SUN.

[3] except `Class.forname` that will require a special treatment but it can only refer to a class already on the phone.

1.2 Java MIDP security mechanisms and their limits

The underlying principle of MIDP security is that an application can perform any action a user could do, but if an action may hit the security of the user (examples of such functionalities are sending SMS, registering an application to react later on a given event, taking a picture, geo-localizing the handset), then the user should explicitly give his permission prior to the action. Depending on the level of control, this acknowledgement may be given once for all, once per session or each time the functionality is used.

Furthermore, MIDP2.0 provides a second level of protection: digital signatures. The operator or the handset manufacturer can sign an application to express they trust it. The application will then belong to a given "protection domain". Priviledges of a MIDlet suite are raised up to the level attached to a protection domain (Operator, Manufacturer or Trusted 3rd Party) by checking the signature of the MIDlet suite with the digital certificates stored in the mobile equipment (system formed of the mobile handset and a SIM card). This will enable the access to more functionalities on the handset or reduce the number of security queries asked to the user.

Even if the sandbox mechanism provides a reasonable level of security, there are good reasons to evaluate applications before their deployment:

- The security policies embedded on the terminal are too coarse-grained. In litigious cases, the system relies on the user but the user may not have all the necessary information (for example, in some countries, it is difficult to know the cost of an overcharged SMS message from its number);
- security checks are done at runtime when the action is going to be performed. Some psychological effects may impair the judgement of the user:
  - too many warning screens may drive the user to acknowledge an action without reading the screen;
  - if the user has performed many complex actions to reach that point, he may answer positively to an action he would have not performed otherwise.
  - furthermore the user doesn't know how many charged actions are still necessary to complete the process.
- some operations could affect the network (denial of service attacks: sending a message from hundreds of phones at a given date, for example), the operator should have a way to check and prevent those attacks.
- when a midlet is signed, the signature issuer is de facto responsible of the behavior of applications carrying it's signature. Owners of digital certificates enabling priviledge elevation in MIDP 2.0 are responsible of the means they use to avoid signing malicious MIDlets. On a signed MIDlet, some security warnings for the user are suppressed. To safely sign the MIDlet, the operator

should know beforehand the contents and the number of the confirmation screens that will disappear after he will have signed the application.

The operator could develop his applications or commission a trusted entity to develop them. But most applications are provided by independent contents providers and they only supply the compiled application to the operator. So the validation must be performed directly on the bytecode without even the help of a specification. The current practice is the following:

- the operator uses contracts as a deterrent with the developers to ensure that security rules are enforced;
- the operator tests the application (or rather has it tested). Sun with a group of handset manufacturers and operators has created a consortium to define a common grid for evaluating applications: the “Unified Testing Criteria”. The current version of the criteria targets the ergonomics and the functionality of the application rather than the security which is more difficult to evaluate through testing. The amount of time devoted to testing an application is necessarily very limited because of budget constraints. Moreover testers have only access to the object code (black-box testing) and not the source code or at least the specifications.
- the operator uses automatic tools to control the contents of the bytecode. This is the path that will be explored in the remainder of this paper.

1.3 A minimal security profile for MIDP applications

Usually, security evaluation means checking the application against its specification and checking the consistency of this specification. Unfortunately applications are closed binaries for the operator and in any case, it is impossible to check completely an application with the economic constraints put on the evaluation process (a cost per application well below 1000 euros). The goal of the evaluation process will rather be to extract an approximate sub-specification of the application behaviour from its code and check that it implies the harmlessness of the application for its user, which means:

- that the use of charged resources is controlled,
- that it does not destroy or steal user assets,
- that it does not block the terminal.

A last important issue for the operator is the compatibility of the MIDlet with a given handset (most MIDlets are charged and it is not acceptable to propose a MIDlet not compatible with the customer handset). Although MIDP is standardized, there are proprietary extensions either because there is a specific capability that may be not available on the handset (Bluetooth connectivity, video capture) or for which no standard API has been defined (SMS in MIDP1.0, geo-localization until recently), because the amount of a given resource cannot be fixed (size of the screen) or because there are too many possible values (various video or audio formats, network settings).

A classification of the properties to check exhibit the following five categories ordered by 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 and the network (eventually also with other devices like the SIM),
- controlling more accurately the temporal behaviour of the MIDlet.

The first generation of automatic tools answered the first issue. We will concentrate on the second category as it corresponds to the most urgent needs (knowing the kind of connections established by a MIDlet). The third is the necessary complement to the second to get an exact picture of the warnings suppressed by signing a MIDlet. The fourth category of properties should be used for semi-critical MIDlets handling data such as credit card number or personal information. The last category is useful to check that the application works [4] and that it does not try to exploit a handset bug [5]

2 Automatic verifications performed by MATOS

MATOS (Midlet Analysis TOol Suite) is the tool developed by France Telecom to automatically validate MIDlets. MATOS is built as a framework that takes a MIDlet suite and a policy file as arguments and performs a set of independent verification phases depending on the contents of the policy file. Phases can be either global for a given MIDlet suite or launched on each MIDlet:

A JAD conformity analysis it is a textual analysis that checks that the descriptions of the MIDlet, in the descriptor file and in the manifest file of the java archive, are consistent.

[4] There are strict rules to follow to design a GUI, display or sound, that works properly on several phones.

[5] As an example, on some phones, a developer can hide the confirmation screens presented to the user when an SMS is sent if he calls the sending of the SMS and the displaying of the screens in an unexpected sequence.

A class and method control that checks that the MIDlet only uses regular MIDP APIs and registered proprietary extensions.

An argument analysis that checks whether the arguments used in the calls of critical methods fulfil the requirements expressed in the development charter.

The core of MATOS is the argument analysis but the results are valid only if the set of APIs used by the MIDlet suite are restricted to the subset for which security rules have been designed. Checking that this hypothesis is valid is the main goal of the class and method control.

3 Static analysis of method arguments

The main achievement of MATOS is to identify the calls to MIDP methods considered as dangerous, to provide an approximation of the set of the possible values of their arguments and to check that they are acceptable according to a given security policy.

3.1 Rationale for the choice of the analysis

The arguments we want to control are mainly the ones that should be presented to the user when he accepts a security sensitive operation according to MIDP 2.0 security policy[6]. Most of them are of type `String`: URLs of opened connections, names of MIDlets delayed or names of the record databases opened by the MIDlet. Names are usually constant strings defined in the program. URLs may have a computed component but there is usually a fixed prefix containing the connection scheme and the destination address[7].

Consequently we have decided to concentrate on the analysis of object references and strings in particular. In Java, objects of type `String` are not mutable. Complex strings can be built with the help of a `StringBuffer` object that accumulates elementary components and transforms the whole in a result string.

We will use the following assumption on the way critical strings are built: concatenation operations are done with local `StringBuffer` objects (objects only assigned to a local variable). This assumption has been validated by experience on a collection of free MIDlets (see section 4).

[6] In fact a lot of implementations do not bother to explicitly show the URL of the connection to the user.

[7] URLs follow the general framework defined by IETF standards [2]:
`scheme://[user[:password]@]host[:port]/localpath?arguments`.

3.2 Principle of the analysis

The analysis must be performed on the bytecode because it is the only form of the application available to the test teams. Designing and implementing a new static analysis for a complete language like the Java bytecode is time consuming and error-prone. Classpath resolution, native methods handling, basic handling of `Class.forName` are some examples of the technical points that must be handled to level a theoretically sound analysis to a practical tool. Therefore we have chosen to rely on existing libraries and to combine them to build a solution. MATOS uses components from SOOT, a complete framework [16] for the optimisation of bytecode developed at Mc Gill University in Montreal. Like most optimising compilers, it provides :

- basic local dataflow analysis (forward or backward) [12],
- advanced devirtualisation analysis: we will rely on its points-to analysis,
- a simpler intermediate language called Jimple which is a stack-less version of the Java bytecode.

3.2.1 Syntax for the results

We will present the analysis for strings. Results are coded as regular expressions (or automata) that model an upper approximation of all the possible values of the strings. A result is a couple (v, R) where v is an abstract variable in a set V and R is a set of simple definitions where the syntax of a definition is defined by the following grammar:

```
<definition> ::= <var> = <expr>
<expr> ::= <expr> + <expr>         // String concatenation
        |  <expr> | <expr>         // Alternatives
        |  *                       // bottom value (anything)
        |  "<string constant>"     // A string constant
        |  <var>                   // a variable
```

3.2.2 Points-to analysis

The goal of a points-to analysis is to provide an approximation of the points of definition (points of allocation for an object) of the possible contents of a storage cell (in Java, a local variable, a static or an instance field).

It is a very classical analysis first designed for the C language [1,14] and then adapted to Java [17,11,13,10]. Points-to analysis can be used for precise class-analysis (also known as devirtualization) [10] as the exact class of an object is known at its allocation point. It is also used in escape analysis [17], and synchronization removal [4].

A points-to analysis can usually be modelled as two related phases:

- first, a model of the program is built where allocation nodes and variables are abstracted and linked together to form a flow graph. Building this graph usually requires that an approximation of class analysis has already been done. How this support analysis and the points-to analysis cooperate is a place for variations between different methods (see [10] for various options),
- Then the information carried on the allocation nodes is propagated through the graph. The information can be annotated with contextual information like the contents of the call-stack when the allocation was performed.

We will write $\mathcal{P}(v)$ for the result of the points-to analysis of v.

3.2.3 Def/Use analysis

The def/use analysis is a simple forward dataflow analysis that links the statements that define the contents of a (local) variable with the statements that use it. The analysis is local to a given method.

Def/use analysis can be used to build the set of possible definitions of a value at any point in the program. We represent each set of values by an abstract variable and a set of simple equations where an elementary equation has the following syntax:

```
<definition> ::= <var> = <expr>
<expr> ::= <expr> | <expr>        // Alternatives
        |  "<string constant>"    // A string constant
        |  <var>                  // a variable
        |  %i                     // i-eth parameter
        |  o.m(<expr>,...<expr>)  // method invocation
        |  o.f                    // field reference
```

Each variable in an equation represents an occurrence of a variable at a point of use. An elementary expression on the right hand side represents the instruction at the possible point of definition. This syntax is very close to the syntax of results.

3.2.4 Combining analysis

The main problem of points-to analysis is that it is a control-flow insensitive analysis. Therefore it is not possible to follow operations with side effects like concatenation with `StringBuffer`. So we combine it with a def/use analysis to handle concatenations when they are limited to a method. This is the most common case in Java as the classical idiom `s1 + s2` is translated into the allocation of a hidden buffer to which the two strings are appended and its transformation back to a non-mutable string.

The algorithm proceeds as follow. A goal to solve can be either:

- $< C, m, i >$ - evaluate the possible values of the i-eth argument of a call to a method m of class C,

- $< C, f >$ - evaluate the possible values of a field f of class C,
- $< C, m >$ - evaluate the possible values returned by a method m of class C.

The algorithm uses the following sets of values:

- a working set W of pairs $(v, g) \in V \times G$,
- a result set R that contains equations,
- a finite map $S : G \to V$ from solved goals to the variables used as entry points in R for their solution.

Initially, $S = R = \emptyset$ and W contains the initial goals defined by the security policy with identified variables to use to label the result.

The main argument for the termination of the procedure is that terms describing goals are built on a finite alphabet and each goal is solved in a bounded time. The algorithm iterates the process presented below on the goal set W until it is empty.

(i) A goal (v, g) of W is selected and $g \mapsto v$ is added to S. There is a support variable v_g in the program whose set of possible values correspond to a solution of g:
   - if $g =< C, m, i >$ then v_g is the i-eth parameter,
   - if $g =< C, f >$ then v_g is the field,
   - if $g =< C, m >$ then v_g is the pseudo-variable introduced for the return statement.

(ii) We look at the contents of $\mathcal{P}(v_g)$:
   - either all the values that may be stored in the variable are string constants and the goal is solved. If $\mathcal{P}(v_g) = s_1 \dots s_n$ are , then we add $v = x_1 \| ... \| x_n$ to R and launch a new iteration on the next goal (back to step 1),
   - or there are computed values and we proceed to the next step.

(iii) For each goal there is a set of instructions and corresponding variables that define the possible values for the goal:
   - if $g =< C, m, i, v >$ the instructions are the invocation of the method m (finding them requires a class analysis, we use the result of the points-to analysis) and the variable the i-th argument of each invocation,
   - if $g =< C, f, v >$, the instructions are the ones that store a new value in the field f (here again a class analysis is required) and the variable is the contents,
   - if $g =< C, m, v >$, the instructions are the return instructions in the code of the method m and the variable is the pseudo return variable.

(iv) We use the def/use analysis to get the local expression that defines each value. The result of the analysis is a pair (v_0, D) where v_0 is a variable and D is a set of equations that represents the possible definitions that we must translate into a set of equations describing possible string values.

We add $v = v_0$ to R and we iterate through each equation $v_d = e_d$ of D. The interesting cases of the translation are the following:

- the value to follow is the result of a special String or StringBuffer operation (string concatenation, transformation of a string buffer in a string or vice versa): the equation is translated in the corresponding operation on strings (conversion operations are silently ignored) and added to R;
- the value to follow is a string result of a regular method invocation $o.m$ where o can be of classes $\{C_i\}_{i \in I}$ after class analysis. We add the goal $\{< C_i, m > / i \in I\}$,
- the value to follow is the i-eth parameter of the current method $C.m$: we add the goal $< C, m, i >$,
- the value to follow is a field $o.f$. If $o.f$ is of type `String` and the possible classes of o are $\{C_i\}_{i \in I}$ then we add the goals $\{< C_i, f > / i \in I\}$
- otherwise the equation cannot be interpreted and the result $v = *$ is added to R.

To add a goal g means that if g is not in the domain of S then (v_d, g) is added to W otherwise $v_d = S(g)$ is added to R.

The program variables that are the focus of the analysis may be of type `StringBuffer`. We must be careful that those variables are *local* and we prohibit the analysis of fields of type `StringBuffer` because the points-to analysis and the iteration process ignore the control-flow of the program.

3.3 Validation of the arguments

The properties to check contain the following information:

- what arguments should be checked,
- what are the acceptable values,
- how the result of the analysis should be presented to the tester.

Each property is defined by a rule that identifies the data to analyse (using the method presented in 3.2.4) and a filter that describes what is expected and how to present the results. Most filters are built from two components:

- the first one checks that there is enough information to give back a meaningful verdict (for example every value returned for a given argument must contain a fixed prefix with enough letters)
- the second checks the actual conformity of the different values found with

```
 TextField phoneField;
 ...
 sendOne("sms://" + phoneField.getString());
 ...
 private void sendOne(String phone) {
    MessageConnection co=(MessageConnection) Connector.open(phone);
    TextMessage msg = ...
    co.send(msg); ... }
```

Fig. 1. Relevant fragments of the test program.

```
<rule name="connector-1">
  <args class="javax.microedition.io.Connector"
     signature=
        "javax.microedition.io.Connection open(java.lang.String)"
     report="connector-report">
   <argument position="1"/> </args>
  <description> This rule checks ...  </description>
</rule>

<report name="connector-report">
 <description> This report ... </description>
 <pseudoString>
  <filter pattern="\x22([^:]*):([^\x22]*)\x22">
    <b> An identified network connection with the following
        parameters: </b>
    <ul> <li> Method called: %C.%M </li>
         <li> Method hosting the call: %c.%m </li>
         <li> URL used: %r </li>
         <li> Computable prefix: %1:%2 </li>
         <li> Scheme used: <font color="green"> %1 </font>
         </li></ul></filter>
    ...
    <filter pattern=".*"> ... </filter>
  </pseudoString>
</report>
```

Fig. 2. Relevant fragments of the rule profile used

the security policy.

Each elementary filter is made of a sequence of pairs made of a regular expression and a text to print. For each argument the possible values are enumerated, loops are eliminated by considering that the looping part is an unsolved string. The tool selects the first filter that matches the value analysed and prints the corresponding text. The text may refer to parts of the matche d expression.

The profile is written with an XML syntax and contains its own documentation. Modifying the rules and filters can be performed by an engineer with a good understanding of security policies and of Java but it is not necessary that he has an in-depth understanding of how the tool works.

3.4 Example

As an illustration, figure 1 presents some fragments of a archetypal program to analyze with a profile that contains the rules given in figure 2. The result

```
An identified network connection with the following parameters:
- Method called: javax.microedition.io.Connector.open
- Method hosting the call: Test1.sendOne
- URL used: "sms://" + javax.microedition.lcdui.TextField.getString(*)
- Computable prefix: sms://
- Scheme used: sms
```

Fig. 3. Result of the analysis

of the analysis is given in figure 3. The evaluator can see that the application may send an SMS to a phonenumber obtained from a dialog with the user.

4 Results and limitations

4.1 Benchmarks

MATOS has been tested on a collection of 415 free MIDlets from the site `www.midlet.org`. We have analysed the opening of a connection (functions in the class `javax.microedition.io.Connector`). The MIDlets retrieved have the following characteristics:

- they were MIDP 1.0 MIDlets (MIDP 2.0 was just appearing),
- some of them use proprietary extensions: for example Siemens r Nokia packages to send SMS messages,
- a quarter (127) of them create connections of any kind and require a real analysis (156 connection openings). Most of them are HTTP connections.

The results are the following:

- a simple points-to analysis for constants can guess the target of 33 invocation of the openings of a connection (26%)
- the complete analysis can solve 80 % (126) of the questions.
- time for the analysis is less than 5s per MIDlet[8] and is almost constant.

Because connections are HTTP connections, most of the URL contain a variable part corresponding to the arguments of the query, so a pure points-to analysis is not sufficient.

Because the MIDP library is huge compared to the code of the MIDlet, the time required for the analysis is almost constant (in fact, we use a stripped down version of MIDP whose semantics only respect the possible control paths and data paths between elements). The cost of the analysis is still too high to be performed on the handset.

The main reasons for the last 20% of unresolved URLs are the following:

- too much computation of the URL (substrings)

[8] Times taken on a 1.6 Ghz centrino laptop with 512 Mb memory.

- use of non local StringBuffer objects (rare)
- use of heap structures shared with other portion of the code (Hashtables, vectors). A more precise contextual analysis could solve these cases.
- URL coming from external resources (record databases, value fetched from the network, etc).

The main limitation of the tool is that it can only control object references and not values of a primitive type because the points-to analysis only consider objects. In fact MATOS can follow constant integers when they are used locally because of the local def/use analysis. This is enough to check integers used as pseudo enumerated constants because developers refer to the symbolic value which is inlined by the compiler (because it is a final static field).

4.2 A development charter

As shown above, as any static analysis tool, MATOS only builds an approximation of the actual values of the arguments and may not be able to decide whether a security property is respected or not. To be able to guarantee to the developer that MATOS will have enough information to successfully check a security property when it is respected, we ask the developers to follow a "development charter" that requires explicitly that the URL arguments are built following the definition of a "pseudo-constants". Those rules express the fact that strings are almost constants and that concatenation is used wisely and are in fact more strict than necessary but they are easier to understand than a precise characterisation of the capacity of analysis of the tool.

4.3 Comparison with other works

The first generation of automatic tools, directly integrated on commercial provisioning platforms [5], checked the use of a given class in a MIDlet and compared it with the contents of a database to check the availability of the API on a given handset.

The closest tool is *Trusted Logic MIDP Validation Tool*, an extension of a tool originally designed for Java Card applets validation. The tool has similar objectives (verifying arguments) but is based on a probably cleaner abstract interpretation model. There is no result available on its performances especially on its handling of computed values.

The Java String Analyzer (JSA) developed at BRICS [3] computes a finite state automaton that provides an upper approximation of the values of string expressions. It is used to control strings appearing in web services, XML transformation or SQL queries.

4.4 Towards a standardization of security requirements for MIDP

Recently, Sun and a group of device manufacturers launched a program called "Java Verified"; all the devices from these manufacturers will embed the Java Verified certificate. Applications signed by "Java Verified" will be granted the privileges of "Trusted 3rd Party" applications. Java Verified relies on a set of MIDlet test criteria called "Unified Test Criteria" (UTC) [15]. We claim that automatic argument approximation offers a viable technological option in order to check new UTC requirements such as: "the MIDlet does not make network connections out of ones described by the developer" and such a security assurance requirement would increase the confidence level of UTC compliant MIDlet suites.

5 Conclusion and further works

5.1 Achievements

We have shown why operators have to analyse the bytecode of downloadable applications to ensure a correct level of security for their customers and that it was possible to develop a useful and robust automatic tool for validating MIDlets by assembling existing analysis bricks developed for other purposes. The tool is used by the test teams of the mobile operator Orange to check the applications it proposes. A web version of the tool may be made available to developers on the Orange developer portal.

We think that the results obtained are accurate in the most useful cases. Advocating the need for clean programming in the developer community and the usefulness of automatic security evaluation for maintaining the confidence of the customers should help to make the last issues disappear. Providing a way to control the connections opened by a MIDlet is now a requirement of the Unified Testing Criteria [15].

5.2 Assumptions on the virtual machine

All those analyses rely on the assumption that the terminal is safe: well-typed applications should have a completely bounded control-flow. Bugs in implementations of the MIDP library or in the core of the virtual machine, attacks on the hardware at low level can ruin the effort put on application validation. More effort should be devoted to the isolation and proofing of a real trusted computing base on the handset.

Another assumption made by the analysis on the virtual machine is that all the code of the application is known (no dynamic loading), no reflective property of the language was used (there is none in J2ME except `Class.forName`) and that there is no native code extensions to application (no JNI support). All those assumptions are verified by the J2ME CLDC configuration but not by J2ME CDC or J2SE. It is not clear whether it would be possible to extend the analysis to those environments.

5.3 Further works and open issue

We are planning to add new phases to MATOS that would give the test team a better understanding of how MIDlets are built before they run it on a real handset: what are the different screens and how does the information flow through the MIDlet. Those additions would solve most of the remaining security validation requirements.

On the other hand, checking the compatibility and dependability of a MIDlet for a given handset is a completely open issue: first there are a lot of informal ergonomic requirements (use of the full screen, choice of the buttons, choice of colours), second there are various compatibility issues specific to each terminal that are difficult to model (behaviour of audio players, issues in memory layout, control on the lifecycle of the MIDlet when an external event occurs, etc.).

References

[1] Andersen, L. O., "Program Analysis and Specialization for the C Programming Language," Ph.D. thesis, DIKU, Department of Computer Science, University of Copenhagen (1994), available as DIKU report 94/19.

[2] Berners-Lee, T., L. Masinter and M. McCahill, *Uniform resource locators (URL)*, RFC 1738, IETF (1994).

[3] Christensen, A. S., A. Møller and M. I. Schwartzbach, *Precise analysis of string expressions*, in: *Proc. 10th International Static Analysis Symposium, SAS '03*, LNCS **2694** (2003), pp. 1–18. URL http://www.brics.dk/JSA/

[4] Corbett, J. C., *Constructing compact models of concurrent Java programs*, in: *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis* (1998), pp. 1–10.

[5] Elata, *Elata sense commercial documentation* (2004). URL http://www.elata.com/

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

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

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

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

[10] Lhoták, O. and L. Hendren, *Scaling Java points-to analysis using Spark*, in: G. Hedin, editor, *Compiler Construction, 12th International Conference*, LNCS **2622** (2003), pp. 153–169.

[11] Liang, D., M. Pennings and M. J. Harrold, *Extending and evaluating flow-insenstitive and context-insensitive points-to analyses for Java*, in: *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (2001), pp. 73–79.

[12] Nielson, F., H. R. Nielson and C. L. Hankin, "Principles of Program Analysis," Springer, 1999.

[13] Rountev, A., A. Milanova and B. G. Ryder, *Points-to analysis for Java using annotated constraints*, in: *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (2001), pp. 43–55.

[14] Steensgaard, B., *Points-to analysis in almost linear time*, in: *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1996), pp. 32–41.

[15] Unified Testing Initiative, *Unified testing criteria for Java(tm) technology-based applications for mobile devices* (2004), version 1.4.
URL `http://www.javaverified.com`

[16] Vallée-Rai, R., L. Hendren, V. Sundaresan, P. Lam, E. Gagnon and P. Co, *Soot - a Java optimization framework*, in: *CASCON conference*, 1999, pp. 125–135.
URL `http://www.sable.mcgill.ca/publications`

[17] Whaley, J. and M. Rinard, *Compositional pointer and escape analysis for Java programs*, in: *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (1999), pp. 187–206.