Runtime monitoring for next generation Java ME platform

Gabriele Costa, Fabio Martinelli, Paolo Mori, Christian Schaefer, Thomas Walter

Runtime Monitoring for Next Generation Java ME Platform

Gabriele Costa[a], Fabio Martinelli[a], Paolo Mori[a], Christian Schaefer[b], Thomas Walter[b]

[a] *Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, Pisa, Italy*
[b] *DOCOMO Euro-Labs, Munich, Germany*

Abstract

Many modern mobile devices, such as mobile phones or Personal Digital Assistants (PDAs), are able to run Java applications, such as games, Internet browsers, chat tools and so on. These applications perform some operations on the mobile device, that are critical from the security point of view, such as connecting to the Internet, sending and receiving SMS messages, connecting to other devices through the Bluetooth interface, browsing the user's contact list, and so on. Hence, an adequate security support is required to protect the device from malicious applications.

This paper proposes an enhanced security support for next generation Java Micro Edition platform. This support performs a runtime monitoring of the operations performed by the Java applications, and enforces a security policy that defines which operations applications are allowed to perform. Two possible design approaches for the security support are presented and compared.

1. Introduction

The increased availability of mobile broadband connections enables the expansion of software downloads to mobile phones. This leads to a greater number of available services and a better utilisation of the computational power of mobile phones. A light version of the Java platform for mobile devices, Java ME, is available on most of the mobile phones currently available on the market. Moreover, in the last years, a lot of Java applications for mobile devices, named MIDlets, have become available for download on the Internet, such as games,

[☆] The work of G. Costa, F. Martinelli and P. Mori was partially supported by the European projects EU-FET-IP Emergent Connectors for Eternal Software Intensive Networked Systems (CONNECT) and EU-ICT-STREP Context-aware Information Sharing (Consequence).

Email addresses: gabriele.costa@iit.cnr.it (Gabriele Costa), fabio.martinelli@iit.cnr.it (Fabio Martinelli), paolo.mori@iit.cnr.it (Paolo Mori), schaefer@docomolab-euro.com (Christian Schaefer), walter@docomolab-euro.com (Thomas Walter)

Internet browsers, chat tools. The downside of this increased software availability is an increase in the possible attack vectors; e.g. the misuse of resources. As an example, a malicious MIDlet could establish a network connection to a remote server and transfer to this server all the contacts stored in the mobile device phone list. Thus, techniques need to be in place that prevent downloaded MIDlets from misusing resources on the mobile phone.

This paper presents a framework to prevent such misuse of resources using a runtime monitor that performs execution-time checks to monitor if the application is behaving correctly or not. This runtime monitor enhances the flexibility of the Java security model for mobile phones, allowing to enforce security policies without having to rely only on signatures of the downloaded applications.

Hence, our framework allows the secure execution of MIDlets developed by third companies, signed by untrusted principals or even unsigned.

This paper extends the security solution for mobile devices described in our previous work, that is based on a security enhanced version of the Java ME platform [1] [2], by proposing an alternative framework architecture, based on the bytecode in-lining technique. Moreover, this paper provides performance figures and a detailed comparison between the two approaches.

The paper is structured as follows. Section 2 describes the Java ME platform and discusses some previous attempts to enhance its security. Section 3 describes our approach to improve the security of the Java ME platform, and Section 3.1 describes the architectures of our framework exploiting the two distinct approaches. While Section 4 describes the in-lining technique, Section 5 describes the security policy language and the security policy evaluation. Section 6 presents the implementation of two prototypes, one for each approach, running on real mobile devices, and Section 7 makes a comparison between the two approaches. Section 8 describes the demonstration scenario in which our security support could be adopted. Finally, Section 9 draws the conclusion.

2. Related Work

Java Micro Edition (Java ME) is a version of the Java platform for mobile and embedded devices, such as mobile phones, personal digital assistants (PDAs), TV set-top boxes, printers, and so on. The Java ME architecture mainly consists of two components [3]:

- Configuration: provides the most basic set of libraries and virtual machine capabilities for a broad range of devices,
- Profile: set of APIs that support a specific set of devices.

Different Configurations and Profiles are provided for distinct classes of devices. In particular, Java ME for mobile phone consists of: the *Connection Limited Devices Configuration* (CLDC) [4], and the *Mobile Information Device Profile* (MIDP) [5]. Other profiles and configurations exist for more powerful devices, such as smart-phones and TV set top-boxes.

The Java ME platform provides a basic security support. The security support provided by CLDC concerns the low level and the application level security. The low level security guarantees that MIDlets do not harm the device while running. The application level security, instead, deals with security relevant operations performed by MIDlets, such as accesses to libraries or resources. To execute MIDlets, CLDC adopts a sandbox that ensures that: the MIDlet must be pre-verified; the MIDlet cannot bypass or alter standard class loading mechanisms of the KVM; only a predefined set of APIs is available to the MIDlet; the MIDlet can only load classes from the archive it comes from; and, finally, the classes of the system packages cannot be overridden or modified.

The security support provided by MIDP [6] defines four protection domains: *Untrusted, Trusted, Minimum, and Maximum*. Each MIDlet is bound to one protection domain depending on its provider, and this determines the value of its permissions. Permissions refer to the actions that the MIDlet can perform during its execution and their value can be either *allowed* or *user*. For example, the `javax.microedition.io.Connector.http` permission refers to HTTP connections. If the value of this permission is *allowed*, then the permission is granted, otherwise, if the value is *user*, a user interaction to explicitly grant the right is required every time that the MIDlet tries to establish an HTTP connection. When asked by the MIDP security support, the user can deny the right to execute the action, or can allow it by choosing among three possible values: *oneshot*, *session*, *blanket*. If the user chooses *oneshot*, the right to execute the current action is granted, but the user will be asked when the MIDlet will try to perform the same action again. If the user chooses *session*, the right to execute the current action is granted to the MIDlet until it terminates. Instead, if the user chooses *blanket*, the MIDlet will be allowed to perform the action until it is uninstalled or the permission is explicitly changed by the user. In other words, this disables any further control on this action.

A security study of Java ME has been presented by Kolsi and Virtanen in [7], where they described the possible threats and the security needs in a mobile environment. In particular, they described how MIDP 2.0 solved some security issues of MIDP 1.1, but they concluded that some problems are still present. A security analysis of Java ME has been presented also by Debbabi et al. in [8], [9] and [10]. In these papers, they detail the MIDP and CLDC security architecture, and they identify a set of vulnerabilities of this architecture. Moreover, they also test some attack scenarios on actual mobile phones. However, the previous papers do not propose any improvement to the Java ME security support to solve the security issues they described.

An attempt of extending the Java ME architecture with an enhanced security support is shown in [11]. This paper proposes a runtime monitor architecture that consists of a *Runtime Monitor*, a *Policy Manager* and a *History Keeper*. The Runtime Monitor is in charge of making resource access decisions, and relies on the Policy Manager to identify the relevant application-specific policy. Once the policy is identified, the Runtime Monitor evaluates its conditions in conjunction with resource usage history information of the system and MIDlet, as obtained from the History Keeper. This architecture enforces policies written

in the *Security Policy Language* (SPL) [12]. SPL is a constraint based security policy language that allows to express simultaneously several types of authorization policies, hence allowing the definition of complex access control models (e.g. RBAC, DAC, TRBAC).

In [13] the authors present a framework for the run time monitoring of applications running on the .NET platform exploiting the in-lining technique. Their model is symmetrical to the one we have presented but for the technical differences deriving from the diverse environment. Recently, they also presented in [14] a complete system composed by an instrumentator for modifying the .NET Intermediate Language and a PDP for enforcing ConSpec policies. Actually, the proposed framework implements the PDP as a *Dynamic Link Library* that is invoked during the execution. Instead, our model exploits a separate process for monitoring the running applications. The necessity for this mechanism arises from the different execution environment. Indeed the Java MIDlets do not share the execution context and they can not access public, external libraries.

Proof carrying code (PCC) enables safe execution of code from untrusted sources by requiring a producer to furnish a proof regarding the safety of mobile code. Then the code consumer uses a proof validator to check, with certainty, that the proof is valid (i.e. it checks the correctness of this proof) and hence the foreign code is safe to execute. Proofs are automatically generated by a certifying compiler by means of a static analysis of the producer code. The PCC approach is problematic for two main reasons. A practical difficulty is that automatic proof generation for complex properties is still a daunting problem, making the PCC approach not suitable for real mobile applications. A more fundamental difficulty is that the approach is based on a very strong assumption: since the producer sends the safety proof together with the mobile code, the code producer should know all the security policies that are of interest to consumers. This appears an impractical assumption since security may vary considerably across different consumers and their operating environments.

3. Runtime Monitoring

The execution of MIDlets on a mobile device is a threat for the device's security, especially if these MIDlets have been downloaded from third parties, i.e. unknown (and hence untrusted) providers. As a matter of fact these MIDlets could execute actions that could damage the device, could violate the privacy or could consume the phone credit. As an example, a malicious MIDlet could send offensive SMS messages or e-mails to the contacts in the device phone book; this MIDlet violates both the privacy of the user, because it uses the contacts in the user's phone book, and could also be expensive for the user, because the user is charged for the SMS messages sent by the MIDlet and her/his reputation is harmed. Moreover, a user could also be interested in controlling the MIDlets that come from trusted parties, such as the mobile device manufacturer or the telco operator, to avoid to be charged for some (legal) operations automatically executed by these MIDlets, such as network connections to check for software updates.

The Java ME platform provides basic security support, as described in Section 2. However, this support is not adequate to allow the secure execution of MIDlets, because the only factor that is taken into account to decide whether a MIDlet is allowed to execute a given action or not is the provider of the MIDlet. Hence, if the MIDlet provider is trusted, i.e. the MIDlet is signed by a principal that is included in the list of trusted principals stored on the mobile device, the action can be executed. Otherwise, the action is denied or an explicit authorization by the user is required before the MIDlet continues to execute the action.

This paper proposes a solution to improve the security support of the Java ME platform. This solution is not based on the level of trust of MIDlet providers, but on the monitoring of MIDlets during their execution and on the enforcement of a security policy. In particular, each security relevant action that is performed by a MIDlet is intercepted by our security support before it is actually executed on the mobile device, and it is evaluated against the security policy to determine whether the MIDlet has the right to execute it or not. Moreover, after the execution of the action, the security policy is evaluated again, to check the results and determine whether the execution of the MIDlet can be continued or not. If the security policy does not allow the execution of the action, an error (Java exception) is returned to the MIDlet as result of the skipped action. If the MIDlet is instrumented to manage this error, the execution will continue, otherwise the MIDlet terminates.

The security policy takes into account various factors in the decision process, such as the parameters of the action that the MIDlet wants to perform, the status of the policy (that has been influenced by the actions that have been performed by the MIDlets previously executed on the device), and the current status of the mobile device, such as the battery charge level, or the telecom operator signal strength. The right of a MIDlet to execute an action is not static, because it depends on factors that can change over time. Hence, the right of executing a given action that was granted to the MIDlet in a previous interaction, could be denied to the same MIDlet in a further interaction. An action represents an interaction of the MIDlet with the mobile device, and it is defined as security relevant when its execution can alter the state of the mobile device, or when the user can be charged for this action. As an example, sending an SMS message is a security relevant action because the user is charged by the mobile telecom operator, and because the message can contain user data that should be kept confidential. Since we want to monitor Java ME applications, we chose as security relevant actions a proper subset of the API methods of MIDP and CLDC core classes.

3.1. Runtime Monitoring Architecture

¿From an architectural point of view, the runtime monitor framework has been integrated with the Java ME architecture following the *Reference Monitor* model [15]. This model includes two main components: a Policy Enforcement Point (PEP) and a Policy Decision Point (PDP). The PEP is the component that intercepts the invocations to security relevant actions before they are actually

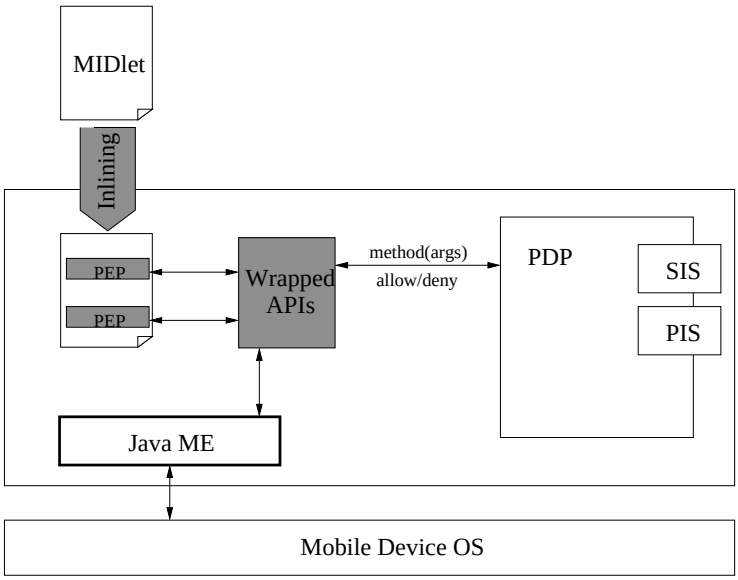


Figure 1: Runtime monitoring architecture: in-lining approach

executed, asks the PDP for the permission, and enforces the PDP decision. The PDP, on the other hand, is the component that reads the security policy and that performs the decision process.

We experimented with two distinct approaches for the integration of the PEP in the Java ME architecture. The first approach integrates the PEP directly in the MIDlet to be monitored. This solution is based on the modification of the bytecode of the MIDlet itself to insert the segment of code implementing the PEP before and after the code that invokes the security relevant action. The process of bytecode modification is known as *in-lining*, and the MIDlet obtained from the in-lining process is called *in-lined MIDlet*. This implies that, to allow the monitoring, every MIDlet should be in-lined before being executed on the mobile device. The in-lining process can be executed on the mobile device, before installing the MIDlet itself, or by a remote (and trusted) server. Hence the life cycle of the MIDlet is the following: *i)* the MIDlet is developed and distributed to the users by the MIDlet provider, *ii)* the MIDlet is downloaded onto the mobile device, *iii)* the MIDlet is in-lined on the mobile device by the MIDlet In-liner, *iv)* the in-lined MIDlet is installed on the mobile device, *v)* the in-lined MIDlet is executed several times, and v_{bis}*)* the execution is monitored by the proposed framework, *vi)* the in-lined MIDlet is possibly uninstalled from the mobile device. All the steps of the MIDlet lifetime, except for step *v)*, are executed only once. Steps *iii)* and v_{bis}*)* are due to our security framework.

The architecture of the framework adopting the in-lining technique is shown in Figure 1. The whole framework runs on the mobile device, and the components are:

- The *In-liner* takes as input the MIDlet executable code, i.e. the jar file including the MIDlet or the individual classes. In a policy file the security relevant API calls are specified. The in-liner searches in the bytecode of the MIDlet for these API calls and inserts the code that implements the Policy Enforcement Point in the MIDlet itself.

- The *Policy Enforcement Point* (PEP) is responsible for monitoring the MIDlet during its execution. Specifically, it intercepts all the security relevant actions that the MIDlet tries to perform on the underlying mobile phone, asks the Policy Decision Point to decide whether the action is allowed and enforces the decision by actually executing the action or by returning an error to the MIDlet. As previously explained, the PEP consists of a set of instructions that have been properly embedded in the in-lined MIDlet.

- The *Policy Decision Point* (PDP) is responsible for evaluating whether a given action is permitted in the current state by the policy on the mobile phone. It is invoked by the Policy Enforcement Point, and it exploits the Policy Information Service to get the policy and to manage the policy state, while it exploits the System Information Service to retrieve information about the mobile phone state.

- The *Policy Information Service* (PIS) is responsible for managing the policy states. In particular, it stores the value of policy variables, that are necessary for a decision on the execution in a later run of the application. These variables can be defined for one application or for the whole system.

- The *System Information Service* (SIS) is responsible for providing information about the system, such as the current date and time, the battery state, the CPU load, and so on.

The second approach, instead, integrates the PEP directly in the Java ME platform components, as shown in Figure 2.

With respect to the previous architecture, the main difference is that the in-liner component is missing, because the PEP instances are embedded in the MIDP and CLDC components of the Java ME architecture, instead of being in the MIDlet executable code. The collocation of the PDP component, and hence of the PIS and SIS ones, are the same as with the previous architecture. To embed the PEP in the MIDP and CLDC, the Java ME platform source code has been modified. In particular, the source code of the security relevant methods that have to be monitored has been modified by inserting the invocation to the PDP at the beginning and at the end of the code that implements each of these methods. In this way the policy is evaluated and enforced both before and after the execution of the security relevant method. The PEP also enforces the decision of the PDP because, if the PDP decision is positive, it continues the execution of the original method code. But, if the execution of the method is denied by the PDP, the PEP terminates it by throwing a Java Exception.

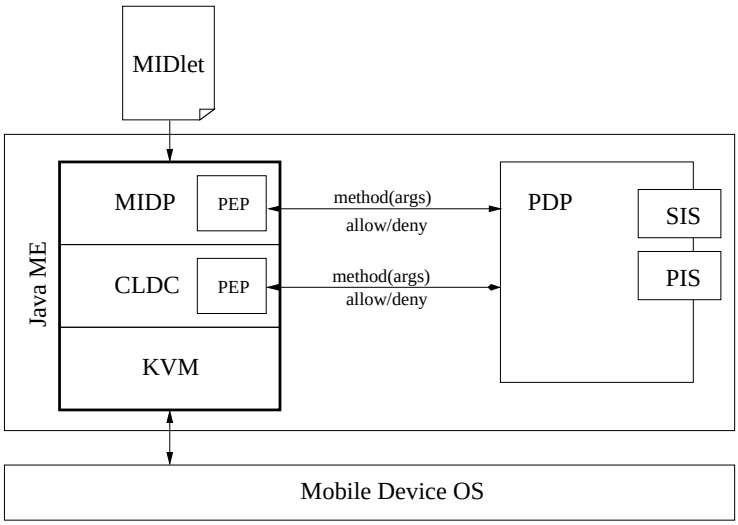


Figure 2: Runtime monitoring architecture: customized Java ME environment approach

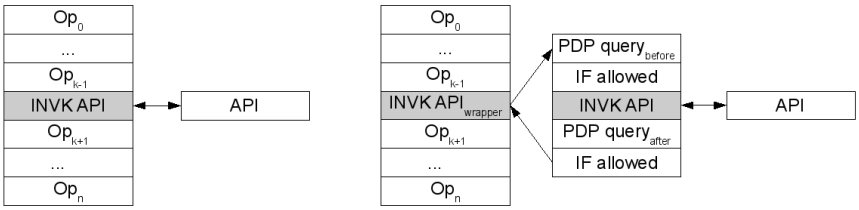
In this case, if the PDP invocation has been made before the execution of the method, the method execution is skipped.

In both environments the PDP is responsible for loading the security policy to be enforced. Hence, the device owner is in charge of installing on the device the policy, that is loaded and used by the PDP. We imagine that new policies can be written directly by the device owner or downloaded from some trusted policy provider.

The implementation of the components of the architecture is discussed in Sections 6.

4. In-lining Process

The in-lining process is an instrumentation step performed on the MIDlets before being executed by the Java ME platform. Firstly, the in-liner extracts from the current policy specification the list of the security-relevant API calls, i.e. the list of Java ME methods to be monitored. Then, it inspects the MIDlet bytecode sequence looking for corresponding invocation operational codes. These positions are replaced by the invocations to the PEP. The PEP mainly performs three basic operations: converts the current method call and its parameters (the first n positions in the operand stack where n is the number of the method arguments) into a message; sends the message to the PDP using an internal, inter-process connection; receives the decision from the PDP and, if it is positive, calls the corresponding API to actually execute the method, otherwise it throws a security exception. Furthermore, the PEP, before returning to the original operational flow, communicates to the PDP the resulting value



(a) Original bytecode structure (b) Instrumented code

Figure 3: The instrumentation of a single API call in a bytecode sequence

obtained from the APIs. This communication is implemented similarly to the previous one.

The current PEP strategy allows the in-lined application only to run if the PDP is working properly. Indeed, we always wrap the MIDlet's state changes (*Paused*, *Active* and *Destroyed*) that are necessary for managing the policies scope. In particular, when a MIDlet starts its actual execution, namely moving its state from Paused to Active, a signal is sent to the PDP that acts accordingly. This is obtained by always instrumenting the `StartApp` call, that is the MIDlet entry point, both when it is explicitly required by the current policy or not. If the PDP does not reply, i.e., it is not running or has been corrupted, the PEP handles the lack of information as a negative response and aborts the current operation.

The PEP is implemented as a set of external classes masking the core APIs. These classes declare exactly the same methods as the original classes, but their implementation includes the PEP logic. In this way, the bytecode rewriting process simply consists in re-addressing the invocations of the original methods to the corresponding PEP classes and methods, as shown in Figure 3. As a consequence of this technique, our instrumentation procedure is quite general with respect to the bytecode structure. Indeed, it can be successfully applied also to obfuscated code since the Java APIs references can not be re-named. Finally, we observe that the modifications to the structure of a MIDlet invalidates its signature (if any). Clearly this means that, on the current devices, an in-lined MIDlet loses all the access privileges provided to its original, unmodified version. However, since our framework is intended to be an alternative to the certificate-based one, this is not a drawback in our approach. Furthermore, our system seems particularly appropriate for the current mobile software scenario. Indeed many MIDlets are produced by companies and developers that for various reasons, e.g. high cost, do not apply a proper signature to their applications. Using our security model, users can install and execute unsigned, well-behaving MIDlets with no limitations on their usability.

Taking into account the consequences of rewriting (part of) MIDlets is another crucial issue. Indeed, as Java is not provided with a formal semantics, actually we can not prove formally that the segments of code added for each PEP do not introduce some unpredictable behaviour. However, since every

PEP is composed by few, atomic instructions not interacting with the other parts of the MIDlet, we are confident that in-lined MIDlets preserve their integrity (remember that an in-lined wrapper takes the same arguments, returns the same value and, possibly, throws the same exceptions as the corresponding original method). The only critical operation that a PEP introduces is the communication with the PDP. The two components interact by passing each other messages using a internal, inter-process (socket) connection. Clearly, this sensitive channel needs to be secured against unauthorized writing operations (imagine the consequence of sending a false event to the PDP or a false authorization to a PEP). However, the PDP can prevent MIDlets from accessing this special socket providing a sort of self-protection. Note that, even if this technique does not prevent attacks from non-Java applications, this approach is absolutely coherent with the purposes of our system, namely creating a secure environment for executing Java programs.

5. Security Policy Specification and Evaluation

For security policies specification we adopted the ConSpec language, that has been defined by the *Security of Software and Services for Mobile Systems (S3MS)* project [16] [17]. ConSpec was designed to express policies specifically for resource limited devices, such as PDAs and other mobile devices. ConSpec is inspired by Erlingsson and Schneider's PSLang [18], and its formal semantics is presented in terms of security automata [15]. Here we provide a brief introduction to ConSpec. A detailed description of the features of ConSpec language, such as its semantics, can be found in [19] and [20].

ConSpec polices consist of a set of rules. Each rule has a *Scope* that defines the limits of applicability. There are four possible scopes: *Object, Session, Multi-Session* and *Global*. The *Object* scope indicates that the policy rule refers to a specific instantiation of an object. *Session* scope defines that the policy rule is to be enforced for the entire session of the MIDlet. Policy rules with *Multi-session* scope are enforced on consecutive execution of the same MIDlet. Finally, *Global* scope rules are enforced on all the MIDlets running on the mobile device. Hence, policies with *Multi-session* and *Global* scope can be exploited to regulate interactions among the same MIDlet and different MIDlet, respectively.

Each rule of the policy defines the authorization conditions that must be satisfied to allow the MIDlet to execute some security relevant actions.

The security relevant actions are represented by the Java ME core classes methods that mediate the access to the valuable resources of the underlying device. In particular, we consider as security relevant actions the methods of the Networking package, such as `javax.microedition.io.Connector.open` to create a network connection with a remote site, and the methods of the Wireless Message API (WMA) package [21], such as `javax.wireless.messaging.MessageConnection.send` to send an SMS message. However, our approach is general, and other methods can be easily added to the set of security relevant actions. As an example, the methods of the Personal Information Management (PIM) package could be considered as security relevant as well.

CONSPECVERSION 1.2	1
RULEID Rule_1	2
VERSION 1.0	3
SCOPE session	4
SECURITY STATE	5
int smsNo = 0;	6
BEFORE java.microedition.io.Connector.open(String url)	7
PERFORM	8
(url.getProtocol().equals("sms") && url.getNumber().startsWith("+39"))→skip	9
BEFORE javax.wireless.messaging.MessageConnection.send(Message msg)	10
PERFORM	11
(smsNo < N)→smsNo++;	12

Table 1: Example of ConSpec security policy

Figure 5 shows an example of ConSpec policy. The scope of the policy is *Session* (line 4), hence the policy is enforced during the execution of each MIDlet instance.. The first clause of the policy (lines 7-9), authorizes MIDlets to open a connection using the protocol "sms" only if the telephone number starts with "+39". The second clause of the policy (lines 10-12) concerns the method `javax.wireless.messaging.MessageConnection.send` that is used to send SMS messages, and states that before the execution of the method, the value of the variable `smsNo` must be less than `N`. If this condition is satisfied, the execution of the method is allowed, and the value of `smsNo` is incremented. Hence, this policy allows each MIDlet to send no more than `N` SMS messages.

Furthermore, enforcement of the policy also requires that system information be retrieved from the device. As ConSpec was specifically designed to be utilised on mobile devices, there is a number of mobile device specific classes of information that can be retrieved. These include the types of networking available, such as WiFi, Bluetooth or IrDA; the battery level remaining, or more mundane information such as the current date and time. The policy rule can define limits over these types of information, such as preventing specific applications executing when the battery level is below a particular level.

The evaluation of the security policy to decide whether, at a given point of the execution, the MIDlet has the right to perform a given action, is executed by the Policy Decision Point that, in turn, exploits the other two components of the architecture: the Policy Information Service (PIS) and the System Information Service (SIS). The PDP initially gets the security policy from the PIS, and

builds an internal representation, that is used to efficiently evaluate the policy against the security relevant actions to be authorized. The PDP is invoked by the PEP both before and after the execution of each security relevant action. In particular, the PEP sends a message containing the name of the action, i.e. the standard Java syntax for methods names, its actual parameters and a unique MIDlet identifier, that is, the MIDlet name extracted by the Java descriptor file (JAD). The PDP examines all the rules of the security policy that concern that action to find one that is satisfied, i.e. that allows the execution of the action itself. As a matter of fact, the policy could include a number of rules for the same action that have distinct authorization predicates. Since we adopt a default-deny approach, the current action is permitted only if one rule of the policy explicitly allows it.

To evaluate the rules of the security policy, the PDP takes into account various factors, such as the MIDlet name or the value of the parameters of the specific action request, and it may need also the value of some policy variables. For example, a rule of the policy could allow to open network connections only with a given site. In this case, the security relevant action is represented by the method `javax.microedition.io.Connector.open`, whose parameter is the URL of the remote site, and the policy rule includes a condition on this URL. Another example is the one where the security policy allows to open a further network connection only if this MIDlet has opened less than `N` connections. The current number of connections is represented by a policy variable. The PDP retrieves the current value of this variable to perform the decision process, i.e. to decide whether a new connection can be opened. Moreover, if the right is granted, the PDP also increases the variable value to represent the fact that a new network connection has been opened. In these cases the PDP interacts with the PIS. The Policy Information Service is in charge of managing the state of the policy, that is represented by the current values of the variables in the policy. The values of the variables are permanently stored on the mobile device, because they should be available even when the device has been rebooted. The PDP asks the PIS both to retrieve and to update the variable value. The PIS manages the scope of the policy variables. If the scope of a variable is *Global*, all the MIDlets executed on the device read the same value of that variable. In the case of *Session* variables, the variables are only for one MIDlet, and each MIDlet sees its own value for that variable, while if the scope is *Multi-session* only distinct instances of the same MIDlet see the same value of the same variables. According to the scope of the variable that is specified in the security policy, the PIS creates a proper data structure to (persistently) store the variable value.

The PDP may also need some information regarding the current state of the mobile device to evaluate the policy. For example, a policy could state that an SMS message may only be sent if the battery level is above a given threshold. In this case, the PDP interacts with a further component of the architecture, the System Information Service, that is responsible for the collection of data concerning the mobile device state. Such data include the current time and date, the current battery level, the types of networks available, the signal strength,

the CPU load and the amount of free memory remaining. In support of this, the SIS exploits some functions that interacts with the operating system of the mobile device.

6. Implementation

To evaluate the effectiveness of the architectures we described, and to make a comparison among them, we developed the related prototypes, both running on real mobile phones.

6.1. In-lining Approach Prototype

The implementation of the prototype of the in-lining approach has been carried out using different mobile devices. In particular the whole architecture has been installed and tested on the Nokia phones E61 and N78. Nokia E61 works with the Symbian v9.1 S60 3rd edition operating system while Nokia N78 runs Symbian v9.3 FP2. Both of them support the Java 2 Micro Edition *Mobile Information Device Profile v2.0 (JSR 118)*. As explained in the previous sections, the adoption of the in-lining approach does not require the modification of any components on the mobile phone, such as the operating system or the Java ME platform, but it is sufficient to install additional software: the in-liner and the PDP (with PIS and SIS).

The in-liner component of the prototype has been implemented in Java ME. Considering the reduced computational capabilities and the restrictions on memory usage posed by most of the current mobile devices, the in-lining process seems to be on the edge of what is actually feasible on mobile devices. Indeed, it handles compressed Java archives, i.e., JAR files, reads and modifies class files, changes the inner structure of the target MIDlet (and the corresponding JAD description file) and, finally, builds a new executable MIDlet.

The files compression issue needs a first treatment. As a matter of fact, the compression/decompression algorithms require such a computational effort that the JVM prefers to delegate them to external, binary code. The Java Standard and Java Enterprise Editions obtain this by exploiting the *Java Native Interface* (JNI). Unfortunately, this approach is impossible for Java ME, that is provided with no such APIs. Thus, it has been necessary to create a light-weight, fully Java-implemented compression library. Another complication arises from large class files. Actually, class files can exceed the maximum memory size available for the running Java ME applications. This problem is solvable by implementing a partial-representation mechanism that can work on classes without having their complete representation.

The main operation of the in-liner consists in re-addressing sensitive API calls to the corresponding wrapper ones. Actually, a wrapper W_C is a (statically defined) class that re-implements every accessible method of its corresponding original system class C. In particular, for each security-relevant method m declared by C, the wrapper W_C provides a method m'. These methods communicate with the PDP and, if the current security constraints are respected,

call the execution of their original counterpart m. These wrapping classes are written and compiled together with the whole system. This means that, when defining a security policy, we can only refer to the set of methods effectively pre-loaded. As many of the security-relevant operations that a MIDlet can perform correspond to some system API call, this seems to be a reasonable approach. However other ways are also viable. For instance, it is possible to create these classes dynamically starting from the definition of the original class.

The last step of the in-lining process is the MIDlet re-assemblage. In doing this, we add the wrapping classes to the original code of the application. Since the APIs to be wrapped are a well-known, finite set, we preferred to create the corresponding classes and ship them with the in-liner. This solution requires a few KiloBytes of memory space but relieves the in-liner from the burden of creating these classes at runtime. Again we have to use our compression library to create a valid JAR file and, in addition, we modify the Java description file changing some entries, e.g. the archive size and the application entry point. The result of the procedure is a new JAR archive including the instrumented MIDlet, ready to be installed on the mobile device. Since the in-lining process modifies the MIDlet bytecode, it invalidates the MIDlet signature (if any), thus making the MIDlet untrusted to the Java ME security support. Hence, to skip the standard Java ME security support, the new MIDlet should be signed again, or the Java ME permission support should be deactivated.

The Policy Decision Point has been implemented in C++ for Symbian v9.1 and is running also on later Symbian OS versions. The C++ implementation has been chosen mainly for efficiency reasons. As a matter of fact, since the PDP code is executed twice for each security relevant method invoked by the MIDlet, an inefficient implementation could introduce a considerable overhead in the MIDlet execution time. The PDP is a daemon, and it is started at device initialization time and it is always active on the mobile device. Once activated, the PDP daemon reads the security policy, builds the policy internal representation that is used to test the actions against the security policy, and suspends itself waiting for an invocation from a PEP.

The communication between the PDP daemon and the PEPs embedded in the MIDlets has been implemented through a local socket. The security policy itself prevents MIDlets from communicating directly with the PDP to interfere with the policy evaluation process. This is possible because any connection that an in-lined MIDlet tries to perform should be first authorized by the PDP. Moreover, if the PDP is unavailable, the PEP automatically denies any action that the MIDlet tries to execute. An advantage of this choice is that all the MIDlets that are executed on the mobile device share the same PDP. This allows to naturally implement global security policies, i.e. policies that take into account the actions performed by all the MIDlets running on the device. As an example, a security policy could state that no more than `N` SMS messages can be sent every day. In this case, the PDP should take into account the SMS messages that have been sent by all the other MIDlets. Another example could be a policy that states that a MIDlet A can be executed only if another MIDlet B has not been executed yet.

The Policy Information Service provides a means to store and retrieve data. If data is to be stored, the PIS attempts to securely store the data and sends an response back to the PDP indicating whether or not the storage request was completed. Similarly, if the PDP needs data, it asks the PIS that attempts to read it. If the data was found, it is sent to the PDP. Otherwise, an error code is returned, indicating the error returned.

The System Information Service (SIS) is the module that interacts with the Symbian operating system to get the current state of the mobile device. The SIS exploits the Symbian APIs, such as the System Agent one, to get the current status of the battery, of the charger, and of the network signal. Moreover, the SIS also retrieves current time and date.

6.2. Security Enhanced MIDP and CLDC Prototype

To evaluate the second solution we proposed in Section 2, we also developed a prototype of the modified Java ME runtime environment. This solution requires the replacement of the standard Java ME platform installed on the mobile device with a security enhanced one, that embeds the PEP. To implement the security enhanced Java ME platform we exploited the phoneME Feature Software MR2 [22], an open source implementation of the main components of the Java ME architecture, such as the MIDP v2.0, the CLDC v1.1, the *Wireless Message API* and many others. The reference mobile device, instead, is an HTC Universal smartphone (also known as QTEK 9000) running the Linux Openmoko distribution [23]. The phoneME Feature Software MR2 release includes the full source code. In particular, the KVM code is developed in C++, both for efficiency reasons and because it interacts with the underlying operating system. The code of the Java ME core classes is developed partly in Java and partly in C or C++. In this case too, C functions are used mainly to implement the interactions with the underlying operating system. Our customized version of phoneME was built on a desktop computer exploiting the OpenEmbedded development environment [24] and configuring the cross compiler for the specific mobile device architecture.

The PEP and the PDP have been integrated in the phoneME source code, according to the architecture described in Figure 2. In this case too, the Policy Decision Point has been developed in the C language, mainly for efficiency reasons. Actually, the policy evaluation engine of the PDP is the same for the two implementations; the only difference is that each of them has been customized for the specific mobile device operating system. The PDP is started by the KVM before the execution of the MIDlet bytecode. Once activated, the PDP thread loads the security policy and suspends itself waiting for an invocation from the PEP component.

The PEP, in contrast, consists of a Java class and a C function. The Java class includes a method, checkPolicy, to activate the PDP. The source code of the classes whose methods implement security relevant actions have been identified in the phoneME Feature Software MR2 release. The invocations to the checkPolicy method are embedded in the source code of those methods, before and after the original code. In this way, the security policy is checked before

and after the execution of the security relevant action. The implementation of the checkPolicy method exploits the Kilo Native Interface (KNI). KNI is used to invoke the C function that actually resumes the PDP and suspends the execution of the Java ME method through the use of semaphores. The PEP communicates with the PDP exploiting shared variables. The data passed to the PDP includes the name of the MIDLet, the name of the method, and the value of its parameters. Once the PDP has evaluated the current action against the policy, its decision is stored in a shared variable as well, and the PDP resumes the PEP and suspends itself, waiting for a new invocation. The enforcement of the PDP decision, when the right to execute an action has been denied, is implemented by throwing an exception in the code of the Java ME method. This error will be reported to the MIDlet. If the MIDlet has been instrumented with the code the code to manage it, execution continues, otherwise the MIDLEt terminates returning the error.

6.3. Performance

The monitoring of the MIDlets during their execution introduces an overhead in the execution time, due to the evaluation of the security policy for each security relevant action they perform. The impact of the policy evaluation on the execution time depends on various factors, such as the complexity of the security policy and the actions executed by the MIDlet. As a matter of fact, the security policy evaluation time depends on the number of rules concerning the current action and on the complexity of the predicates to be evaluated in these rules. Moreover, the overhead impact on the MIDlet execution time is more relevant if the MIDlet performs a large number of security relevant actions with respect to the other computation.

To obtain a realistic evaluation of the overhead, we carried out our tests using a real case MIDlet, the Proteus PicoBrowser, an open-source Java ME Internet browser. We slightly modified the original version of Proteus by adding a timer to measure the execution time. When the browser is asked for loading a new HTML page, the timer starts. Then Proteus retrieves the required file, interprets its content and shows it on the screen. At this point, the timer stops returning the elapsed time. Since the MIDlet performs a connection to the site where the HTML page to be loaded resides, the security relevant method that is exploited is the `javax.microedition.io.Connector.open` one. In our experiments, we enforced distinct policies constituted by a different number of rules. All these rules concern the `javax.microedition.io.Connector.open` method, and hence they are all evaluated each time that the PDP is invoked. Hence, from the overhead point of view, this is the worst case. In fact, the computations required for analysing the policy when a security relevant action is executed, strictly depends on the number of rules referring to that particular action.

We executed the test MIDlet on both the prototypes, i.e. on the one that exploits the in-lining technique and the one that exploits a modified version of the Java ME platform. We repeated the same task, under the same conditions, for both the original and the monitored prototype. To avoid the influence of

network-dependent delays, we accessed a local page, i.e. a page stored on the mobile device itself. The HTML page contained only a couple of text lines in order to minimize the irrelevant computation. In this sense, the results of our tests must be interpreted as the maximum cost due to the monitoring process during a page loading. Indeed, loading a bigger file from the network would lengthen the total execution time reducing the impact of monitoring operations. Moreover, we performed several measures and we computed the average values.

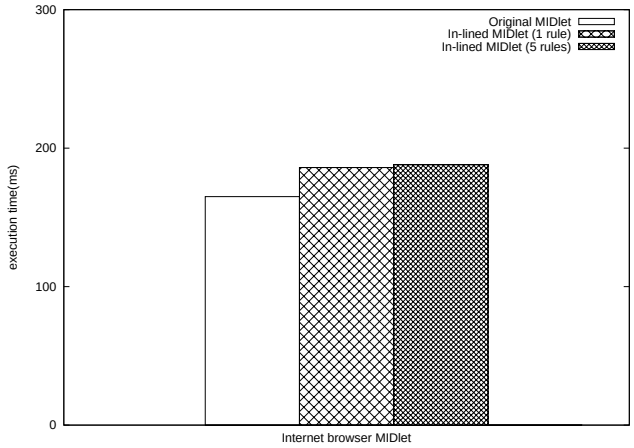


Figure 4: Performances of the in-lining prototype

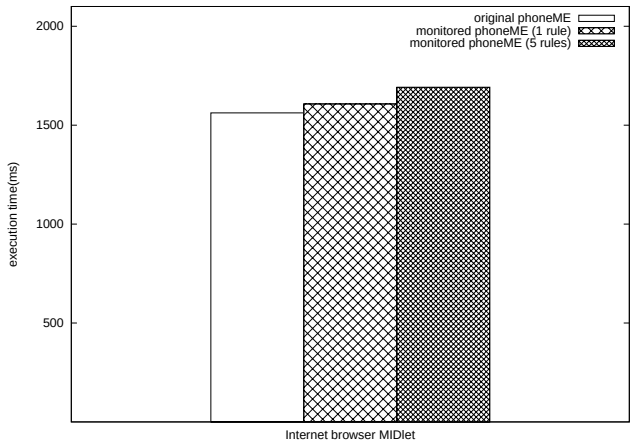


Figure 5: Performances of the security enhanced JVM prototype

Figures 4 and 5 show the results of the tests we performed. The chart in Figure 4 refers to the in-lining approach. We installed the required components, i.e. in-liner and PDP, on a Nokia N78 mobile phone, and we executed the test

MIDlet under different conditions. The first value of the chart, represented by the white bar, refers to the execution of the original MIDlet. The second is the time for the MIDlet guarded with a 1-rule policy and the third is monitored with a 5-rule one. Each rule has one clause only. The number of rules has been considered in our experiments since the PDP needs to verify more conditions before deciding whether an operation is granted or not. This, in principle, could lead to a longer computation and, consequently, to more noticeable delays. We observe that in the case of a policy with 1 rule the overhead is about the 12% of the whole execution time. This delay is due to the presence of an extra process in the system, i.e. the PDP, and to its interactions with the monitored application. When increasing the number of rules in the active policy to 5, we obtain a further 1% performances overhead. These results outline that our monitoring mechanism allows for applying very expressive policies with moderate consequences on the execution time.

The chart in Figure 5, instead, reports the experiments concerning the approach that embeds the PEP in the Java ME platform. In this case too, the first result, (the white bar), refers to the execution of the MIDlet on the original phoneME support, the second bar refers to the execution of the MIDlet on the security enhanced phoneME support enforcing a policy with 1 rule, and the third bar refers the execution of the MIDlet on the security enhanced phone ME support enforcing a policy with 5 rules. We notice that the overhead in the case of a policy with 1 rule is about 3%, while in the case of a policy with 5 rules is about 8%. We recall that the policies we used are entirely composed by rules concerning the only monitored Java ME method, i.e. the `javax.microedition.io.Connector.open` one, that the test MIDlet exploits to load the required HTML page. Hence, this test simulates a worst case and, in general, the overhead produced by the policy having the same number of rules should be even lower than the one we measured.

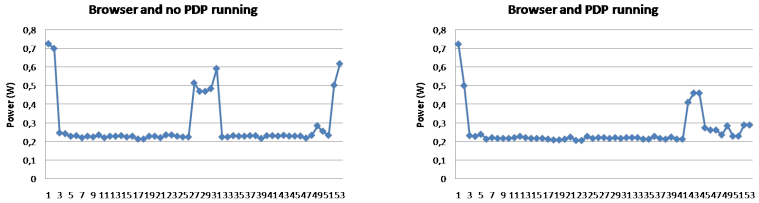
Finally, we recall that the absolute execution times reported in the charts are different from one chart to the other because our experiments have been carried out on two different mobile devices, as our prototypes have been developed on two different mobile phones, as described in Section 6.

6.4. Battery Power Consumption

Besides the performance overhead when running the PDP, we as well checked for the additional battery power consumption. To estimate the additional power consumption we run a couple of tests which are discussed in this section. We used the same mobile device as for the performance measurements for the inlining approach.

To measure the battery power consumption we run Nokia's Energy Profiler[1] The Energy Profiler monitors various parameters of the mobile device among which is the power consumption in Watt (W).

[1] `www.forum.nokia.com/Tools_Docs_and_Code/Tools/Plug-ins/Enablers/Nokia_Energy_Profiler/` last accessed July 1, 2009.



(a) Power consumption without PDP (b) Power consumption with PDP

Figure 6: Battery power conumption without and with PDP as background process

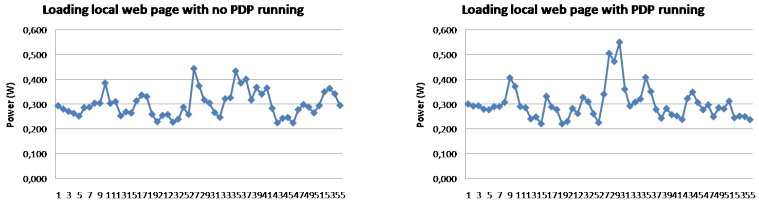
The set-up of our tests was as follows. First, we compared the battery consumption of a mobile device only running the original web browser with a mobile device runnig the web browser and, additionally, the PDP as a background process. We expected to see the impact that an additional process has on the battery consumption. Second, we compared the battery consumption loading a local web page on an unmodified system with the battery consumption loading the same local web page on a modified system, i.e., running the in-lined web browser and the PDP. With these set of tests we expected to get figures on the power consumption with an active PDP. Note that the PDP was using the policy with 5 rules to evaluate.

All test were run manually. In detail for the first set of tests: battery consumption without and with PDP as background process, we started the energy profiler, started the web browser, after which we started the data measuring in the energy profiler and switched back to the web browser. We kept the mobile device idle for some time (with backlight on), then switched back to the energy profiler and stopped data measuring.

The measured data for both scenarios, without and with PDP are shown in Figure 6. The numbers for the X axis are abstract time units. The measuring interval was 0.25 second and the measuring last for 13.25 seconds. On the Y axis the power consumption in Watt (W) is given. Ignoring the peaks in the curves the amount of power consumed shows no differences between the mobile device running the web browser (Figure 6(a)) and running the web browser and PDP as a background process (Figure 6(b)). Thus, the PDP alone does not impact battery power lifetime.

The second set of tests extends the first one by loading a locally stored web page. Thus, data measuring was started in the energy profiler. We switched back to the web browser and started loading the web page. When this was done we switched back to the energy profiler and stopped the data measuring. This test was redone four times and average values were computed. The result for the two scnearios, i.e. original web browser and in-lined web browser and PDP running, are shown in Figure 7. The test run for 13.75 seconds and every 0.25 second the power consumption was measured.

Since the tests were done manually the curves do not match exactly. Taking the peaks we have 0.445 Watt in Figure 7(a) and 0.551 Watt in Figure 7(b)



(a) Power consumption loading local web page without PDP

(b) Power consumption loading local web page with PDP

Figure 7: Battery power conumption loading local web page without and with PDP as background process

which gives an increase of approximately 24%. Overall, the curves show similar behavior with not much difference in power consumption. The average value over all measured data is 0,301 Watt for the orignal mobile device and 0,302 Watt for the modifed mobile device.

Taking into account the additonal performance overhead of 13%, the above figures do not match; one would expect that the difference on power consumption is much higher than the 0,001 Watt (averaged). The explanation is twofold. First, the figures on power consumption include all sources that contribute to the consumption such as data storage access (e.g. web page access), screen updates (displaying and selecting menu entries) as well as CPU load. Second, during our test the PDP is being asked once only which, thus, increases the CPU load only for a short time. Overall the other tasks performed by the mobile device such as data storage access and screen updates dominate the additional CPU load.

We note that the CPU load has an impact on the power consumption.[2] Obviously, the more complex the policies become and the more often the PDU is called, the more power is consumed due to the increased CPU load.

7. Discussion

This Section discusses the advantages and the drawbacks of the two approaches described in Section 3.1 to implement the runtime monitoring of MIDlets.

Following the first approach, the one that embeds the PEP in the MIDlet bytecode through the in-lining process, the deployment of the runtime monitoring framework on a mobile phone is a quite simple task, because it requires to install two software components on the device: the in-liner and the PDP. As described in Section 6, these components have been developed, respectively, as

[2]Details on the battery power consumption of mobile devices, specifically the impact of the CPU load, can be found under `mobiledevices.kom.aau.dk/research/energy_measurements_on_mobile_phones/` last accessed July 2, 2009.

standard Java ME and Symbian C++ applications. Consequently, they can be installed on the device like any other application, and their deployment does not require any change to the original software platform of the mobile device.

Hence, the major advantage of this approach is that it can be adopted on standard devices and Java ME platforms, because no changes to the Java runtime environment nor to the operating system installed on the device are required. Consequently, this approach can be easily adopted on common mobile phones available on the market supporting Java ME. As a matter of fact, as shown in Section 6, we installed and tested our prototype on very common mobile phones, such as the Nokia N61 and the Nokia N78 ones.

On the other hand, to allow their monitoring, this approach requires that MIDlets are instrumented with the PEP code through the in-lining process, before being installed on the mobile device. A drawback is that the in-lining process could be computationally heavy, and could take some time to be executed on the mobile device. However, this process has to be executed only once in the MIDlet's life cycle, i.e. before the MIDlet installation.

Moreover, to avoid the execution of MIDlets that have not been in-lined, the in-liner could be integrated directly in the MIDlet installer, but this requires the modification of the original software platform provided by the mobile device.

Another issue of this solution concerns the MIDlet signature. As a matter of fact, the in-lining process modifies the MIDlet executable code, thus compromising the MIDlet signature.

The second approach presented in Section 3.1, instead, embeds the PEP directly in the Java ME platform, thus creating a security enhanced Java ME platform. The security enhanced Java ME platform should be deployed on the mobile device, replacing the original one. Hence, this could be a drawback of this approach, because currently it can be adopted only on some advanced mobile devices that allow to replace system components such as the Java ME platform. As a matter of fact, to develop our prototype, we chose the HTC Universal smartphone running the Linux Openmoko distribution [23] operating system and the phoneME Feature Software MR2 [22] Java ME platform, that are both open source platforms.

An advantage of this approach is that it allows the execution of standard MIDlets, avoiding the overhead of the in-lining process, because it does not require any modification to the MIDlet to allow the runtime monitoring. This also means that signed MIDlets can be monitored without compromising their signature.

A noticeable difference between these two techniques is the flexibility with respect to the accepted events alphabet. Actually, the PDP running on both implementations of our system does not depend on a fixed, predefined set of security-relevant actions. As the PDP starts, it loads the definition of the current policy. The policy also defines the accepted symbols, that is the set of symbolic names paired with actual method calls. Using a customised Java ME platform, the names referred by the rules of a policy are bound by the methods that the current PEP implementation can intercept. Indeed, if the PEP sends no signal to the PDP before and after the execution of a method, there is no

way for the PDP to monitor it. The only solution consists in implementing a new, customized platform to replace the previous one. On the other hand, an in-lined MIDlet communicates with the PDP through bytecode instructions. As we mentioned in Section 4 the instrumentation process replaces direct calls to some APIs calls with proper PEPs, i.e. calls to a corresponding, wrapping class. The current version of our framework uses an internal library containing the necessary wrappers. Such library handles exactly the same methods as the other approach. Hence the two implementations can use policies defined on the same set of events. However, the in-liner does not depend directly on the particular library used. Indeed, without modifying the application, we can use external libraries enriched with further wrappers that, thereafter, can be used by the PDP. Moreover, being totally independent from the original classes, we could even imagine that wrappers are created during the in-lining process obtaining a completely general implementation. Finally, the in-lining method offers some interesting compositionality issues. Indeed an in-lined MIDlet can be processed more than once. Then, if we decide to monitor calls that were neglected during the first process, we can apply our transformation again with no interferences.

¿From the performance point of view, the difference among the two approaches is due to the PEP component only. As a matter of fact, the PDP engine, that is the code that decides whether the security policy allows a given action performed by the MIDlet or not, is the same for the two approaches. The difference is in the mechanism used to intercept the security relevant actions and to interact with the PDP. The in-lining approach adopts a PEP that is written in Java ME and that is embedded directly in the MIDlet executable code. Hence, for each security relevant method, a set of Java instructions is added before and after the method invocations in the executable code. In the second approach, instead, the PEP is integrated in the Java ME platform code. Hence, the second approach has a lower impact on the MIDlet execution time than the first one, because the wrapping of the security relevant method invocations and and the interactions with the PDP are managed at a lower level, i.e. inside the Java ME platform.

8. Demonstration Scenario

Besides the implementation of the core components of the runtime monitoring we have looked into demonstrating its applicability. For this, we have deployed the components on an off-the-shelf mobile device and have defined and implemented a demonstration scenario. With the demonstration scenario we are aiming at exploring the suitability and effectiveness of the runtime monitoring in a setting close to reality.

The demonstration scenario has been chosen from the area of parental control. We assume that parents would like to control the mobile phone usage of their children and may define the following (non-exhaustive) list of policies:

- Control which applications are used. We allow a white-listing of applications, i.e. parents can control which applications are allowed or disallowed.

Further, parents may define policies that allow a conditional execution of applications. For instance, the child may use an application (e.g., a game) if another application (e.g., learning application) was used before.

- Control for how long some applications are used. If the total accumulated execution time of an application exceeds the defined limit then the application cannot be executed anymore. Note that we check the total execution time only before the application is being started and that we do not force the termination of an application while executing.

- Control which web sites are accessed. We allow a black-listing of web sites although a white-listing approach is followed technically. The black-listing is done by defining strings such that when a URL contains any of the listed strings the web site should not be accessible and all others are allowed.

With the above policies we are checking essential features of the runtime monitoring. First, whether the policy language itself is sufficiently expressive. Second, whether the runtime monitoring correctly implements the parameter checking. Note that in order to allow for the above checking of string containments an external function is to be provided and linked into the runtime monitoring. Third, we are using the SIS to store information (e.g. accumulated execution time of an application) that is being checked again later (e.g. when the application is about to be executed next time).

In our demonstration setup we have installed several Java MIDlets on our off-the-shelf mobile phone (games, learning application, Java-based web browser). Next we have developed a policy editor for the mobile phone in order to allow for a change of the policies. The editor is tailored to the scenario and allows for editing above discussed policies. Besides the runtime monitoring an in-lining MIDlet has been installed on the phone and is used for the in-lining of the Java MIDlets for gaming, learning and web browsing.

We have noticed that the in-lining of MIDlets for the above policies is the most troublesome task for the following reason. Since there are quite a lot different methods to start a MIDlet covering all in the policy definition and in-lining is error prone.

¿From an operational point of view when in-lining a MIDlet the signature on the MIDlet cannot be verified anymore as the MIDlet is modified. Thus, MIDlet execution may require some further user interactions that are not required when properly signed.

9. Conclusion

This paper presented a solution to improve the security support of the Java ME platform that is not based on the trust on the MIDlet provider, but on the runtime monitoring of the MIDlet enforcing an advanced security policy. This new security support enhances the flexibility of the Java security model for mobile phones, because it allows the secure execution of third party MIDlets

(i.e. untrusted MIDlets) without explicitly asking the user for the permission of every action these MIDlets tries to perform. Hence, the adoption of this support will allow the secure execution of MIDlets that have been downloaded from the Internet, developed by unknown providers.

Acknowledgement

The authors would like to thank the anonymous reviewers for their valuable and fruitful comments.

References

[1] A. Castrucci, F. Martinelli, P. Mori, F. Roperti, Enhancing Java ME security support with resource usage monitoring, in: 10th International Conference on Information and Communications Security (ICICS08), no. 5308 in Lectures Notes in Computer Science, 2008.

[2] F. Martinelli, P. Mori, T. Quillinan, C. Schaefer, A runtime monitoring environment for mobile Java, in: 1st International ICST workshop on Security Testing (SecTest08), 2008.

[3] Sun Developer Network, Java ME, http://java.sun.com/javame/index.jsp.

[4] Sun Microsystems Inc., The Connected Limited Device Configuration Specification, Java Standards Process JSR 139, http://jcp.org/aboutJava/communityprocess/final/jsr139/index.html (March 2003).

[5] JSR 118 Expert Group, Mobile Information Device Profile for Java 2 Micro Edition, Java Standards Process JSP 118, http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html (November 2002).

[6] JSR 118 Expert Group, Security for GSM/UMTS compliant devices recommended practice. addendum to the mobile information device profile., Java standards process, http://www.jcp.org/aboutJava/communityprocess/maintenance/jsr118/ (November 2002).

[7] O. Kolsi, T. Virtanen, MIDP 2.0 security enhancements, in: System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on, 2004.

[8] M. Debbabi, M. Saleh, C. Talhi, S. Zhioua, Java for mobile devices: A security study, in: Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC05), IEEE Computer Society, 2005, pp. 235–244. doi:http://doi.ieeecomputersociety.org/10.1109/CSAC.2005.34.

[9] M. Debbabi, M. Saleh, C. Talhi, S. Zhioua, Security analysis of mobile Java, in: Proceedings of the Sixteenth International Workshop on Database and Expert Systems Applications, 2005, IEEE Computer Society, 2005, pp. 231– 235.

[10] M. Debbabi, M. Saleh, C. Talhi, S. Zhioua, Security evaluation of J2ME CLDC embedded Java platform, Journal of Object Technology 2 (5) (2006) 125–154.

[11] I. Ion, B. Dragovic, B. Crispo, Extending the Java Virtual Machine to enforce fine-grained security policies in mobile devices, in: In Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC07), 2007.

[12] C. Riberio, P. Guedes, An access control language for security policies with complex contraints, in: In Proceedings of Network and Distributed System Security Symphosium (NDSS01), 2001.

[13] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, D. Vanoverberghe, Security-by-Contract on the .NET platform, Inf. Secur. Tech. Rep. 13 (1) (2008) 25–32. doi:http://dx.doi.org/10.1016/j.istr.2008.02.001.

[14] L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, D. Vanoverberghe, The S3MS .NET run time monitoring, BYTECODE '09 (March 2009).

[15] F. B. Schneider, Enforceable security policies, ACM Transactions of Information and Systems Security 3 (1) (2000) 30–50.

[16] S3MS project, Security for Software and Services for Mobile Systems (S3MS), http://www.s3ms.org.

[17] N. Dragoni, F. Martinelli, F. Massacci, P. Mori, C. Schaefer, T. Walter, E. Vetillard, Security-by-contract (sxc) for software and services of mobile systems, in: At your service: Service Engineering in the Information Society Technologies Program. MIT press (2009) ISBN 978-0-262-04253-6, 2009.

[18] U. Erlingsson, F. B. Schneider, IRM enforcement of Java stack inspection, in: IEEE Symposium on Security and Privacy, IEEE Computer Society, Oakland, California, USA, 2000, p. 246.

[19] I. Aktug, K. Naliuka, Conspec: A formal language for policy specification, in: Proceedings of the First Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS '07), Oslo, Norway, 2007.

[20] I. Aktug, K. Naliuka, ConSpec: A formal language for policy specification, in: Proceedings of the First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 07), ESORICS, Dresden, Germany, 2007.

[21] JSR 120 Expert Group, Wireless messaging api, Java Standards Process JSP 120, Java Community Process, http://jcp.org/aboutJava/communityprocess/final/jsr120/index2.html (2003).

[22] phoneME project, phoneME Feature Software Milestone Release 2, http://phoneme.dev.java.net.

[23] OpenMoko project, http://openmoko.org.

[24] Openembedded project, http://www.openembedded.org.

10. Bibliographical Sketch

Gabriele Costa is a Ph.D. student in Computer Science at University of Pisa and a researcher of the Information Security Group of IIT-CNR. His research interests concern the foundational and practical aspects of the security of programming languages.

Fabio Martinelli (M.Sc. 1994, Ph.D. 1999) is a senior researcher of IIT-CNR. He is co-author of more than 80 papers on international journals and conference/workshop proceedings. His main research interests involve security and privacy in distributed and mobile systems and foundations of security and trust. He serves as PC-chair/organizer in several international conferences/workshops. He is the co-initiator of the International Workshop series on Formal Aspects in Security and Trust (FAST). He is serving as scientific co-director of the international research school on Foundations of Security Analysis and Design (FOSAD) since 2004 edition. He has been recently awarded by NATO as co-director for a Advanced Training Course. He chairs the WG on security and trust management (STM) of the European Research Consortium in Informatics and Mathematics (ERCIM). He usually manages R&D projects on information and communication security and he is involved in several EU projects.

Paolo Mori received his M.Sc. in Computer Science from the University of Pisa in 1998, and his Ph.D. in Computer Science from the same university in 2003. He is currently a researcher of IIT-CNR, member of the Information Security Group. His main research interests involve high performance computing, and security in distributed systems, such as the Grid, and in mobile devices. He is (co-)author of more than 30 papers published on international journals and conference/workshop proceedings. He is involved in EU projects on information and communication security, (e.g. S3MS, GridTrust).

Christian Schaefer received his Diploma degree in Computer Science from the University of Karlsruhe (TH), Germany. Since September 2003 he is working as a researcher for DOCOMO Euro-Labs in Munich, Germany. His main research interests are the enforcement of security policies in distributed systems with a focus on usage control and security of mobile handsets. He is a member of IEEE.

Thomas Walter is a senior manager in the Smart and Secure Services Group of DOCOMO Euro-Labs, Germany. His research interests include security of software and services for mobile devices, security policies, and access and usage control in distributed environments. Thomas has an Diploma degree in computer science (University of Hamburg, Germany) and a Doctorate in electrical engineering (Swiss Federal Institute of Technology Zurich, Switzerland). He is a member of Gesellschaft für Informatik (GI) and the IEEE.