Enhancing Java ME Security Support with Resource Usage Monitoring

Alessandro Castrucci, Fabio Martinelli, Paolo Mori, and Francesco Roperti

Istituto di Informatica e Telematica
Consiglio Nazionale delle Ricerche
via Moruzzi,1 - 56124 Pisa (Italy)
{alessandro.castrucci,fabio.martinelli,paolo.mori}@iit.cnr.it

Abstract. Both the spreading and the capabilities of mobile devices have dramatically increased over the last years. Nowadays, many mobile devices are able to run Java applications, that can create Internet connections, send SMS messages, and perform other expensive or dangerous operations on the mobile device. Hence, an adequate security support is required to meet the needs of this new and evolving scenario.

This paper proposes an approach to enhance the security support of Java Micro Edition, based on the monitoring of the usage of mobile device resources performed by MIDlets. A process algebra based language is used to define the security policy and a reference monitor based architecture is exploited to monitor the resource usage. The paper also presents the implementation of a prototype running on a real mobile device, along with some preliminary performance evaluation.

1 Introduction

In these last years, the market of mobile devices, such as mobile phones or Personal Digital Assistants (PDAs), has grown significantly. The computational power and the capabilities of mobile devices have increased too. For example, modern mobile devices are able to connect to Internet, to read and write e-mails, and also to run Java applications.

Java Micro Edition (Java ME) is a version of the Java platform for mobile and embedded devices. Java ME for mobile devices mainly consists of two components: the *Mobile Information Device Profile* (MIDP) and the *Connection Limited Devices Configuration* (CLDC). The security model provided by Java ME is not flexible enough to allow the spreading of Java ME applications, MIDlets, developed by third party companies, because it takes into account the provider of the MIDlet only, i.e. the principal that signed the MIDlet. If the MIDlet provider is trusted according to the list of trusted principals stored on the device, the MIDlet is allowed to perform any security relevant action. Instead, MIDlets that come from unknown providers are not allowed to perform security relevant actions, and the mobile device user is tediously prompted to explicitly allow each of them. To avoid to be asked again, the user could choose to allow any further invocation, thereby disabling any further control on the MIDlet.

This paper proposes an approach to enhance the Java ME security support based on the continuous monitoring of the usage of the mobile device resources. The usage of the resources of the mobile device is defined in terms of the sequences of actions that a MIDlet performs on them, i.e. the behavior of the MIDlet. The resource usage monitoring we define is fine-grained, history-based and continuous. The monitoring is *fine-grained* because we define a set of security-relevant actions, i.e. actions performed on the mobile device resources that could be critical for the device security, and we monitor all the invocations to such actions performed by MIDlets. The monitoring is also *history-based*, because to decide whether a MIDlet is allowed to perform a given action, we take into account the sequence of all the actions that have been executed by the MIDlet itself since it was started. This implies that the existence of the right to perform an action is not static, because it depends on the actions that have been previously executed. Furthermore, the monitoring is *continuous* because our approach allows to define conditions that are continuously (repeatedly) checked, and as soon as a condition is violated, a system action is executed, such as interrupting the MIDlet, even if it is not executing a security relevant action. The security policy describes the allowed resource usage patterns in terms of sequences of actions that MIDlets are allowed to perform, and the conditions that should be satisfied before the execution of each action, after, or continuously. The security policy is stored on the mobile device and is applied to each MIDlet that is executed.

Hence, with respect to Java ME security, the main novelties of the proposed security support are that: *i)* the rights granted to a MIDlet to access mobile device resources do not depend on the MIDlet provider; *ii)* these rights are not static; *iii)* the monitoring of the resource usage is continuous, i.e. the controls are not executed before the access only, but also while the access is in progress, and, consequently, a MIDlet could be interrupted while running, even if it is not performing a security relevant action.

1.1 Paper Structure

The paper is structured as follows. Section 2 describes the standard Java ME security support, and reports previous attempts to enhance it. Section 3 describes our approach to improve the security of the Java ME architecture by monitoring the MIDlet execution. In particular, Section 3.1 describes the security policy we adopted, and Section 3.2 describes the architecture of our framework. Section 4 describes the implementation of a prototype running on a real mobile device, the HTC Universal smart-phone, along with a preliminary performance evaluation. Section 5 draws the conclusion.

2 Related Work

The Java ME security support involves all the basic components of the Java ME architecture: Mobile Information Device Profile (MIDP), Connected Limited Device Configuration (CLDC) and Kilobyte Virtual Machine (KVM). The security support provided by CLDC [17] concerns the low level and the application level security. The low level security regards the KVM, and 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 [7,8] 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 [9], 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 problem are still present. A security analysis of Java ME has been presented also by Debbabi et al. in [2], [3] and [4]. 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 [6]. 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 *Security Policy Language* (SPL) [16]. 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).

The main difference with the approach proposed in this paper is in the kind of security controls that are performed. Our approach is focused on the continuous monitoring of the mobile device resources usage, and the security policy defines the rights of a MIDlet by describing the resource usage patterns that the MIDlet is allowed to perform. These patterns could be very complex, and are expressed through a process algebra based language. Moreover, our approach defines continuous controls, that consist of conditions that are continuously checked while the MIDlet execution is in progress. When one of these conditions is violated, the monitor executes a control action, such as interrupting the MIDlet execution. This requires a more complex support then the one for enforcing access control policies.

3 Runtime Monitoring

This paper proposes to enhance the security support of Java ME by monitoring the usage of the resources of the mobile device. This implies the monitoring of the execution of MIDlets to intercept the security relevant actions that they perform on the mobile device resources and the enforcing of a security policy that defines the admitted patterns of these actions.

The actions that are considered as security relevant are the ones that allow the MIDlet to interact with the underlying resources, such as establishing a network connection, sending an SMS message, initiating a phone call, and so on. Hence, we identified a set of methods of the MIDP and CLDC core classes that implement the security relevant actions, and we monitor the execution of these methods. For example, `javax.microedition.io.Connector.open(url)` is the method that creates a connection with the entity represented by `url`, and the method `javax.wireless.messaging.MessageConnection.send(msg)` interacts with the mobile device to send an SMS message to a remote device.

3.1 Security Policy

This section gives a short description of the language for defining security policies. We adopt an operational policy language because we believe that it is closer to user's expertise than denotational ones. Since we deal with sequences of actions, we use a *PO*licy *L*anguage based on *P*rocess *A*lgebra (*POLPA*) (see also [1,11,10]). A policy results from the composition of security relevant actions, control actions, predicates and variable assignments, as described by the following grammar:

$$P ::= \bot \parallel \top \parallel \alpha(\boldsymbol{x}).P \parallel c.P \parallel p(\boldsymbol{x}).P \parallel \boldsymbol{x} := \boldsymbol{e}.P \parallel P_1 or P_2 \parallel P_1 par_{\alpha_1,..,\alpha_n} P_2 \parallel \{P\} \parallel Z$$

where P is a policy, $\alpha(\boldsymbol{x})$ is a security relevant action, c is a control action, $p(\boldsymbol{x})$ is a predicate, $\boldsymbol{x}$ are variables and Z is a constant process definition $Z \doteq P$. The difference between security relevant actions and control actions is that security relevant actions are the ones that the MIDlet tries to perform on the mobile device resources, while control actions are executed by our monitoring support to enforce the security policy. Interrupting and suspending the MIDlet execution are two examples of control actions. The informal semantics is the following:

- $\bot$ is the *deny-All* operator;
- $\top$ is the *allow-All* operator;
- $\alpha(\boldsymbol{x}).P$ is the *sequential operator for security relevant actions*, and represents the possibility of performing an action $\alpha(\boldsymbol{x})$ and then behave as P;
- $c.P$ is the *sequential operator for control actions*, and represents the possibility of performing a control action c and then behave as P;
- $p(\boldsymbol{x}).P$ is the *sequential operator for predicates* and behaves as P in the case the predicate $p(\boldsymbol{x})$ is true;
- $\boldsymbol{x} := \boldsymbol{e}.P$ assigns to variables $\boldsymbol{x}$ the values of the expressions $\boldsymbol{e}$ and then behaves as P
- $P_1 or P_2$ is the *alternative operator*, and represents the non deterministic choice between P_1 and P_2;
- $P_1 par_{\alpha_1,\ldots,\alpha_n} P_2$ is the *synchronous parallel operator*. It expresses that both P_1 and P_2 policies must be simultaneously satisfied. This is used when the two policies deal with actions (in $\alpha_1, \ldots, \alpha_n$);
- $\{P\}$ is the *atomic evaluation*, and represents the fact that P is evaluated in an atomic manner. P here is assumed only to have one action, predicates and assignments;
- Z is the constant process. We assume that there is a specification for the process $Z \doteq P$ and Z behaves as P.

As usual for (process) description languages, derived operators may be defined. For instance, $P_1 par P_2$ is the *parallel operator*, and represents the interleaved execution of P_1 and P_2. It is used when the policies P_1 and P_2 deal with disjoint actions. The policy sequence operator $P_1; P_2$ may be implemented using the policy languages operators (and control variables) (e.g., see [5]). It allows to put two process behaviors in sequence. By using the constant definition, the sequence and the parallel operators, the iteration and replication operators, `i(`P`)` and `r(`P`)` resp., can be derived. Informally, `i(`P`)` behaves as the iteration of P zero or more times, while `r(`P`)` is the parallel composition of the same process an unbounded number of times.

Many different execution patterns may be described exploiting POLPA. Figure 1 shows a simple example of security policy to avoid redirections to other web sites while accessing a predefined web site, "www.siteA.it". At the beginning of the execution, this policy allows the MIDlet to open any network connection. However, if the MIDlet opens a HTTP or a HTTPS connection with the predefined site, then it cannot open any other connection with any other site. On the other hand, if the MIDlet opens a connection with an URL that is not the predefined one, then in this session it cannot open this site anymore.

Moreover, the policy does not allow the MIDlet to open any connection to the predefined site if the protocol is not HTTP or HTTPS. For instance, this policy could be adopted when executing MIDlets that implement Internet browsers, such as Opera Mini [14], to avoid redirections to malicious sites when accessing the predefined site.

```
r([(address(url)!=“www.siteA.it“) ].javax.microedition.io.Connector.open(url))
or
r([( (protocol(url)==HTTP) or (protocol(url)==HTTPS) ) and
   (address(url)==“www.siteA.it“) ].javax.microedition.io.Connector.open(url))
```

Fig. 1. Example of security policy

Figure 2 shows another example of POLPA policy. In this case, the policy allows the MIDlet to send no more than 10 SMS messages to italian users only (i.e. if the telephone number begins with “+39“). As a matter of fact, the policy allows the MIDlet to execute the `javax.microedition.io.Connector.open` method only if the protocol is the SMS one and if the telephone number begins with “+39*“, and it allows the MIDlet to invoke for 10 times only the method `javax.wireless.messaging.MessageConnection.send`.

```
N:=0.
i([((protocol(url)==SMS) and (address(url)==“+39*“)].
  javax.microedition.io.Connector.open(url)
  or
  ( [(N<10)].javax.wireless.messaging.MessageConnection.send(msg).
    N:=N+1)
)
```

Fig. 2. Example of security policy

Figure 3 shows a further example of POLPA policy where the MIDlet is allowed to open a network connection with the site “`http://www.siteA.it`“, and then, either it opens a network connection with the site “`http://www.siteB.it`“ within 10 seconds, or it is interrupted by the control action `revoke_execution`. As a matter of fact, the control action `revoke_execution` is executed as soon as the predicate `[(timer > 10)]` is satisfied. In this example we suppose that the variable `timer` represents a timer.

```
[(url == "http://www.siteA.it")].javax.microedition.io.Connector.open(url).
timer:=0.
( [(timer > 10)].revoke_execution()
  or
  [(timer ≤ 10) and (url == "http://www.siteB.it")].
  javax.microedition.io.Connector.open(url)
)
```

Fig. 3. Example of security policy

3.2 Runtime Monitor Architecture

The architecture for the runtime monitoring follows the *reference monitor* model, and consists of two main components: a Policy Decision Point (PDP) and a Policy Enforcement Point (PEP), as shown in Figure 4.

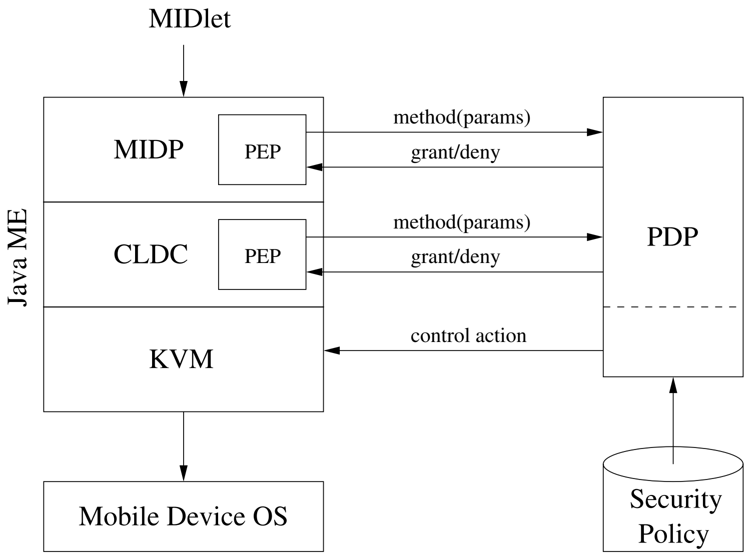

Fig. 4. Runtime monitoring architecture

The PEP is integrated in the MIDP and CLDC components of the Java ME architecture, while the PDP is implemented as a distinct component. This solution requires the modification of the Java ME architecture to embed the PEP, while does not require any modification of the MIDlets, hence allowing the execution of standard MIDlets.

The PEP has two main tasks during the execution of a MIDlet: *i)* intercepting the security relevant methods invocation, and *ii)* enforcing the decision resulting from the evaluation of the security policy on this method. When a security relevant method is intercepted, the PEP invokes the PDP, by passing it the method name and all the invocation parameters. To embed the PEP in the MIDP and CLDC methods we modified the source code of those methods by inserting the invocation of the PDP at the beginning and at the end of the method code. In this way the policy is evaluated and enforced both before and after the execution of the method. The PEP also enforces the decision of the PDP. If the PDP decision is positive, the execution of the method is permitted, then the PEP continues the execution of the original method code. Instead, if the result is negative, the execution of the method is denied, and the PEP terminates it by throwing a Java Exception. In this case, if the PDP invocation has been made before the execution of the method, the method execution is skipped.

The PDP is the component that decides whether a given security relevant method can be performed in a given state according to the security policy. The PDP is initiated by the KVM before beginning the execution of the MIDlet bytecode. The PDP initially gets the security policy from the local storage, and builds an internal representation of the policy. This internal representation is used to efficiently evaluate the policy against the security relevant actions that the MIDlet tries to perform. The PDP consists of two parts: a passive one and an active one. The passive PDP is invoked by the PEP for each security relevant method that the MIDlet tries to execute, before and after the execution of the method. When the policy evaluation process terminates, the passive PDP returns the decision to the PEP that enforces it. The active PDP, instead, repeatedly tests whether a control action should be executed, by evaluating the predicates before the active control actions. A control action is active when the previous actions in the sequence defined by the policy have been already executed by the MIDlet. For example, in the security policy shown in Figure 3, the `revoke_execution` control action is active only after that the connection to the site "`http://www.siteA.it`" has been established. The passive PDP, for each security relevant action executed by the MIDlet, updates the set of active control actions.

4 Implementation

We developed a prototype of the modified Java ME runtime environment that runs on a real mobile device, a HTC Universal smart-phone, exploiting the PhoneME Feature Software MR2 [15]. The PhoneME feature software is an 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 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 [12] and configuring the cross compiler for the specific mobile device architecture. The PhoneME was installed on a HTC Universal smartphone (also known as QTEK 9000) running Linux (Openmoko distribution [13]).

The PEP and the PDP have been integrated in the PhoneME source code, according to the architecture described in Figure 4. From the implementation point of view, the Policy Decision Point consists of two threads developed in C language mainly for efficiency reasons. The PDP is started by the KVM before the execution of the MIDlet bytecode. Once activated, one PDP thread suspends itself waiting for an invocation from the PEP component, while the other repeatedly check the predicates paired with the active control actions every t seconds, where t is a system parameter. If one of these predicate is violated, this thread enforces the corresponding control action through the native AMS support provided by the phoneME.

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 invocations to the checkPolicy method are embedded in the source code of the Java ME methods that implement the security relevant actions, 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 PEP communicates with the PDP exploiting shared variables and semaphores. The enforcement of the PDP decision, when the right to execute an action has been forbidden, is implemented by throwing a SecurityException error in the code of the Java ME method. This error will be reported to the MIDlet.

4.1 Experimental Results

This section evaluates the impact of our enhanced security support on the performances of the Java ME Virtual Machine. As a matter of fact, the MIDlet monitoring slows down the execution of the MIDlet because of the time spent to check the security policy. The overhead on the execution time depends on the enforced policy. As a matter of fact, in general, complex security policies take more time to be evaluated than simple ones. Moreover, the performance degradation also depends on the specific MIDlet, i.e. on the methods it invokes. In particular, the overhead depends on the number of security relevant methods invoked by the MIDlet with respect to the invocations to other methods, because the security relevant methods are the ones that introduce the overhead.

The MIDlet used for our tests performs 10 HTTP connections to a remote site. This is the worst case from the performance point of view, because most of the methods invoked by this MIDlet are security relevant ones, and introduce the monitoring overhead. In a real case MIDlet we expect that the most of the methods invoked are not security relevant ones and, consequently, the overhead due to our security support will be less relevant. To perform these tests the MIDP permissions support has been disabled.

Figure 5 shows the execution times of the chosen benchmark. The three experiments have been executed with the original phoneME software, and with the phoneME software instrumented with our enhanced security support enforcing

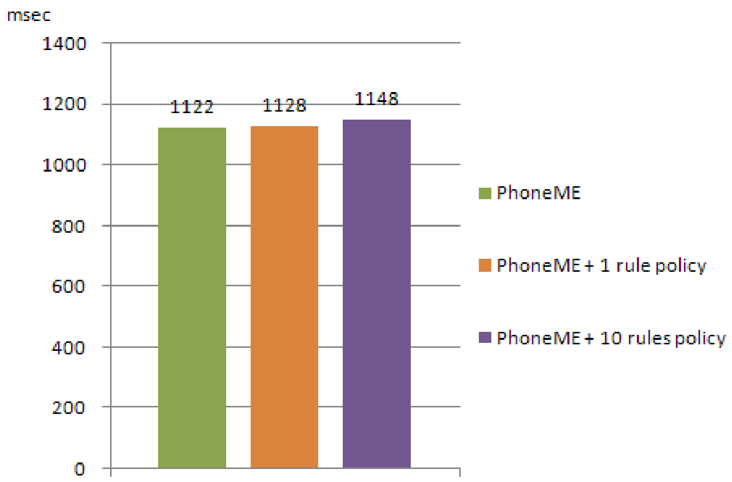

Fig. 5. Performance evaluation

two policies, one with one rule only and the other with 10 rules. In the policy with 10 rules, we chose a worst case again, because the policy has been written in a way such that the PDP must examine all the rules before finding the one that allows the method invoked by the MIDlet. The execution time of the test MIDlet executed on the original PhoneME environment is compared against the one of the same MIDlet executed using PhoneME with our enhanced security support. The results show that the overhead introduced by our system is small. In fact, the enforcing of a policy with one rule results in a 0,5% overhead, while the enforcing of a 10 rules policy results in an overhead of 2,6%. As previously discussed, this results represent the worst case, and we think that in case of a real MIDlet the overhead will be even less.

5 Conclusion and Future Work

We proposed an approach to enhance the security support of the Java ME architecture based on the monitoring of the behavior of the MIDlets. The experiments we carried out on the prototype we developed showed that the overhead due to security controls is very low. Hence, we think that this approach can be successfully adopted on modern mobile devices to allow the secure execution of MIDlets.

References

1. Baiardi, F., Martinelli, F., Mori, P., Vaccarelli, A.: Improving grid service security with fine grain policies. In: Meersman, R., Tari, Z., Corsaro, A. (eds.) OTM-WS 2004. LNCS, vol. 3292, pp. 123–134. Springer, Heidelberg (2004)

2. Debbabi, M., Saleh, M., Talhi, C., Zhioua, S.: Java for mobile devices: A security study. In: ACSAC 2005, pp. 235–244. IEEE Computer Society, Los Alamitos (2005)
3. Debbabi, M., Saleh, M., Talhi, C., Zhioua, S.: Security analysis of mobile java. In: Proceedings of the Sixteenth International Workshop on Database and Expert Systems Applications, 2005, pp. 231–235. IEEE Computer Society, Los Alamitos (2005)
4. Debbabi, M., Saleh, M., Talhi, C., Zhioua, S.: Security evaluation of J2ME CLDC embedded java platform. Journal of Object Technology 2(5), 125–154 (2006)
5. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM 21(8), 666–677 (1978)
6. Ion, I., Dragovic, B., Crispo, B.: Extending the java virtual machine to enforce fine-grained security policies in mobile devices. In: Choi, L., Paek, Y., Cho, S. (eds.) ACSAC 2007. LNCS, vol. 4697. Springer, Heidelberg (2007)
7. JSR 118 Expert Group. Mobile information device profile for Java 2 micro edition. Java Standards Process JSP 118 (November 2002), `http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html`
8. JSR 118 Expert Group. Security for gsm/umts compliant devices recommended practice. addendum to the mobile information device profile. Java standards process (November 2002), `http://www.jcp.org/aboutJava/communityprocess/maintenance/jsr118/`
9. Kolsi, O., Virtanen, T.: Midp 2.0 security enhancements. In: Proceedings of the 37th Annual Hawaii International Conference on System Sciences 2004(2004)
10. Martinelli, F., Mori, P., Vaccarelli, A.: Towards continuous usage control on grid computational services. In: Proc. of International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services 2005, p. 82. IEEE Computer Society, Los Alamitos (2005)
11. Martinelli, F., Mori, P.: A model for usage control in grid systems. In: Proceedings of GRID-STP. IEEE Press, Los Alamitos (2007)
12. Openembedded project, `http://www.openembedded.org`
13. OpenMoko project, `http://openmoko.org`
14. Opera Mini, `http://www.operamini.com`
15. phoneME project. phoneME Feature Software Milestone Release 2, `http://phoneme.dev.java.net`
16. Riberio, C., Guedes, P.: An access control language for security policies with complex contraints. In: Proceedings of Network and Distributed System Security Symphosium (NDSS 2001) (2001)
17. Sun Microsystems Inc. The connectected limited device configuration specification. Java Standards Process JSR 139 (March 2003), `http://jcp.org/aboutJava/communityprocess/final/jsr139/index.html`