Extending the Java Virtual Machine to Enforce Fine-Grained Security Policies in Mobile Devices

Iulia Ion, Boris Dragovic
Create-Net, Trento, Italy.
firstname.lastname@create-net.org

Bruno Crispo*
University of Trento, Italy
crispo@dit.unitn.it

Abstract

The growth of the applications and services market for mobile devices is currently slowed down by the lack of a flexible and reliable security infrastructure. The development and adoption of a new generation of mobile applications depends on the end user's ability to finely manage system security and control application's behavior. The virtual execution environment for mobile software and services should support the security needs of users and applications. This paper proposes an extension to the security architecture of the Java Virtual Machine for mobile systems, to support fine-grained policy specification and run-time enforcement. Access control decisions are based on system state, application and system history data, as well as request specific parameters. The prototype implementation is running on desktops, as emulator, and on mobile devices, proving the high level of flexibility and security, with excellent performance provided by the extended architecture.

1 Introduction

Mobile computing and communications technology has evolved tremendously over the last decade. New mobile computing devices with better design and increased capabilities are released frequently on the market. Consequently, rich mobile services such as e-mail, scheduler, contact synchronization and even scaled-down versions of word processors, spreadsheets and presentation software have become more and more common among mobile users, especially in the business sector.

However, the security model prevailing at mobile platforms does not offer the flexibility required to support the market expansion. The trust model implemented at mobile platforms currently is binary: *trusted* applications are given all requested rights while the *untrusted* ones are locked out of the platform completely. To be deemed as trusted, an application must carry a valid certificate, provided by the platform provider or network operator. This means that application developers must have direct agreements with the certifying parties. As a consequence, the market for mobile software development is inherently closed to third-party developers. Furthermore, the current security models lack the ability for fine-grained, user-defined security policy definition and enforcement - e.g. to control the number of SMSs (Short Message Service) sent, limit the bandwidth used by an application, etc. The main reason behind the rigidity of the security model is found in stringent resource constraints of initial mobile computing devices and also in certain market forces.

In this paper, we propose an extended security architecture and policy model to address the lack of flexibility of the current security model employed at mobile computing platforms. The proposed model has the potential to open up the mobile device software market to third-party developers and it also empowers the users to tailor security policies to their requirements in a fine-grained, personalized manner. Our work focuses on the Java 2 Micro Edition (J2ME) - one of the most widely used virtual machine execution environments for mobile computing devices today.

Our main contribution is the design and implementation of an extended version of the current J2ME, which we baptize **xJ2ME**, from e**x**tended **J2ME**. xJ2ME enables run-time enforcement of a much more expressive class of security policies compared to the current state-of-the-art, allowing for a fine-grained behavior control of individual applications. Furthermore, initial evaluations show no significant, even noticeable, performance overheads.

The rest of the paper is organized as follows. Section 2 motivates this work with an example scenario, and Section 3 reviews the related work. In Section 4, we give a

*This work has been supported partly by the EU project S3MS IST-STREP-27004 and the EU project Re-Trust contract N 021186-2 brief overview of the Java architecture for mobile devices and its security architecture with the current limitations. In Section 5 we describe our proposed solution, Section 6 introduces the policy language we use, and we present the implementation in Section 7. Section 8 evaluates the prototype while Section 9 concludes and outlines future works.

2 Motivation

To better illustrate the limitations of the existing security model for mobile applications and clarify the motivation for the work presented in this paper, we present the following example.

Alice is traveling with her new car equipped with an infotainment system which is connected to the outside world via a UMTS (Universal Mobile Telecommunications System) connection. As she enters Florence, her mobile phone detects a tourist guide service provided by the local tourist information office. If Alice allows her mobile phone to connect to the service and download the corresponding applet, the navigation system will be able to show sites of historical interest and restaurants in close proximity, and download additional information.

However, the attestation service of her trusted computing platform does not recognize the signature of the applet and, therefore, sandboxes it from the navigation system. Annoyed by the fact, she forces the platform to treat it as a trusted application and enjoys the tourist sites in the area. Afterwards, Alice regrets her choice when she discovers that the applet did not only retrieve the information needed, but in addition it downloaded numerous pictures, causing an undesired, costly amount of network bandwidth consumption. Furthermore, in a few areas without direct UMTS connectivity, the applet used the costly Multimedia Messaging Service (MMS) service to transfer information.

Although all the technology for supporting advanced use-scenarios is available, the lack of trust and security for mobile services makes complex applications unusable. The execution environment should allow users to control the behavior of the applications running on their devices on a much finer grain. Some concrete examples of such security policies, none of which are currently enforceable are: (1) The application can only send SMS to specific phone numbers, and for a value that does not exceed 3 Euro per day; (2) The application cannot make international calls, nor phone calls on a data connection to a premium phone number. Normal phone calls are allowed only on weekends and after-work hours; (3) The application may not generate more than 300Kb of traffic per session, over the UMTS connection; (4) Set up maximum number of MMS to be sent per unit of time (hour/day/month/etc.).

The solution that we propose not only allows users to define their own policies for each of the applications to be executed on the platform, but also allows for fine-grained behavior control of those applications. We accomplish this by defining a suitable policy model and by extending the current security architecture of J2ME to provide for flexible run-time policy evaluation and enforcement.

3 Related Work

Two most widely deployed mobile execution environments are *.NET* and *Java* frameworks. The former is supported only by Windows-based platforms, which restricts the portability of the applications written for the .NET framework. This is not the case with the applications developed for the Java framework.

In the case of .NET framework, application code is translated into Common Language Runtime (CLR) and executed under the security policies of the underlying operating system. The security policy on the device is usually set by the service provider (e.g. Cingular, Sprint, T-Mobile). To provide a different device policy, a special agreement with the service provider is necessary. This effectively locks out small software developers from the market since the process is lengthy and costly. Windows Mobile Security Model is based on a three permission tier, which are granted per application:

- Privileged: can call any API, have full access to the registry, file system, and can install certificates. Very few applications need to run as privileged.
- Normal: cannot access the privileged areas
- Blocked: does not allow application execution.

The Security Policy model of mobile devices running Windows OS offers no mechanisms to set fine-grained access control for system resources. As far as we know, no work has been done in extending the .NET Security Policy model for mobile applications.

As with respect to the Java framework, flexible security models have gathered considerable attention in the past. With Java 2 Standard Edition it is possible to use alternative *Security Managers* - classes implementing security-relevant operations. However, due to the limited capabilities of mobile devices, the J2ME security architecture is, by design, not extensible, and therefore does not support this functionality. Users cannot specify alternative Security Managers; they cannot extend nor customize the predefined security policies. No work has been done in the direction of fine-grained security policy specification and monitoring for the Java 2 Micro Edition.

In this paper, we discuss policy enforcement through run-time monitoring [10]. The run-time monitoring approach leaves target application intact and, as shown, poses no significant performance overheads.

4 Java Security Architecture

The two most popular platforms for mobile application development today are Java and .NET. The former, however, still tends to be more widely deployed. In order to set the foundation of our contribution and support the material presented in the following sections, we start by briefly overviewing the Java architecture with a focus on Java 2 Mobile Edition. Next we present the principles of the Java Security architecture, with a focus on the mobile edition.

4.1 Java Architecture

Since the version 2, Java technologies are divided into three editions: Enterprise Edition (J2EE), Standard Edition (J2SE) and Micro Edition (J2ME). Each caters for a different deployment platform. Figure 1(a) contrasts the high level architecture and context of the three editions. J2EE is intended to support multi-tier enterprise applications, J2SE provides for basic Java applications while the J2ME is targeted at resource constrained environments such as PDAs (Personal Digital Assistant) and mobile phones. At the bottom of each of the editions lies a Virtual Machine runtime environment - JVM for J2EE and J2SE, KVM and Card VM for highly constrained platforms.

To support various target platforms and their capabilities, J2ME defines the notions of *configurations* and *profiles*. A J2ME configuration specifies the features and requirements of the Java runtime environment and its APIs that correspond to different classes of devices. The current J2ME specification defines two main configurations: Connected Device Configuration (CDC) [1] and Connected Limited Device Configuration (CLDC) [1]. The former targets high-end mobile devices with richer features. CLDC, on the other hand, is aimed at highly constrained consumer devices. It supports only a subset of a JVM (including APIs, libraries etc.), called KVM.

As depicted in Figure 1(a), the layer above CLDC is the Mobile Information Device Profile (MIDP) [2]. J2ME profiles have the role of defining API libraries that enable specific type of applications to be developed for the target platform - in accordance with the underlying configuration. In conjunction, MIDP and CLDC represent application execution environment and provide for the related functionality. The standardized J2ME environment for highly constrained consumer devices consists of MIDP, CLDC and supporting libraries.

Applications running on top of MIDP are referred to as MIDlets. The various files representing MIDlet code (a JAR file), application supporting data and other resources are bundled together in *MIDlet suites*. Namely, a MIDlet suite is comprised of: (1) a single JAR file containing the Java class (MIDlet), the manifest file, and application resources (images, etc.), and (2) Java Application Descriptor file (JAD) which specifies information related to the application.

The contribution of this paper builds on the principles of the Java security architecture [7]. To outline those principles, we refer to the generalized security setting as provided in J2SE (Java 2 Standard Edition). This gives us grounds to introduce the constrained model of J2ME (Java 2 Micro Edition) and clearly position our contribution relative to it.

4.2 J2ME Security Architecture

The fundamental concept in the generalized Java security architecture is the *sandbox*. Sandbox represents an execution environment with strict, policy-based resource access control and strong isolation properties. Code executing within a sandbox (i.e. the sandbox itself) is associated with a *protection domain* which, in turn, determines the *permission* set *granted* to the application[1].

Generally, JVMs allow the definition of protection domains through Java security policy files. The static sets of permissions thus specified are dynamically mapped at runtime by the JVM. The policy file entries specifying permissions are referred to as *grant* entries. A policy file for a J2SE run-time environment is fully externally configurable by the platform users and administrators. This includes the freedom to define and specify both permissions and domains in J2SE. Figure 1(b) shows Java 2 security architecture.

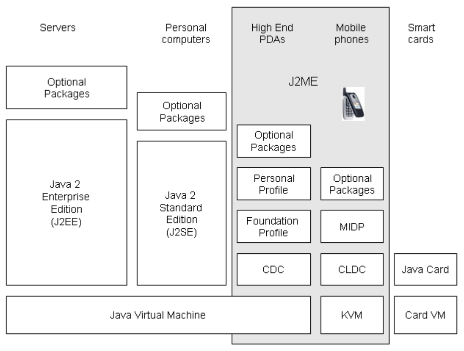
The trust model of Java 2 distinguishes between two main classes of applications: trusted and untrusted. The former is generally allowed to run unrestricted, the latter (applets and remote code) are always subjected to the security policy.

Due to the limited capabilities of the devices running Java 2 Micro Edition (J2ME), the corresponding security architecture has been considerably simplified. While solving the resource consumption issue, such simplification represents a clear trade-off against a number of other aspects of the respective security model. Our work targets the adverse effects that this simplification has on flexibility and granularity of policy specification and enforcement.

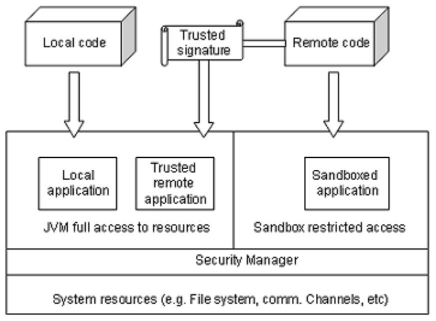
While code in J2SE runs within the JVM, applications executed in J2ME (on top of CLDC and MIDP, as depicted in Figure 1(a)) on a constrained mobile device run within KVM - a scaled-down version of JVM. The sandbox model defined by KVM is considerably different to that of JVM: it restricts the exposed API to that predefined by CLDC; application management occurs at native code level; user is forbidden from touching the classloader or downloading any native libraries.

MIDP retains the general concept of the domain-based security model of J2SE. Permissions in MIDP protection

[1]Note that we use the term application in its most general sense.



(a)


Local code
Trusted signature
Remote code
Local application
Trusted remote application
Sandboxed application
JVM full access to resources
Sandbox restricted access
Security Manager
System resources (e.g. File system, comm. Channels, etc)

(b)

Figure 1. (a)Java technology overview. (b)Java 2 Platform security architecture.

domains are classified in two groups:

- Permission classified as *ALLOWed* are granted automatically, without recourse to user confirmation.
- The *USER* permissions are granted only upon explicit user approval.

Furthermore, the validity period associated with the latter category of permissions may vary between single permission request (*oneshot*), application session wide (*session*) or until explicit revocation (*blanket*).

CLDC and MIDP (Figure 1(a)), as stated previously, represent the execution environment for MIDlets - J2ME applications designed for mobile devices. While MIDP 1.0 introduced the concept of sandbox for MIDlets, MIDP 2.0 defined the MIDlet trust model, as detailed below.

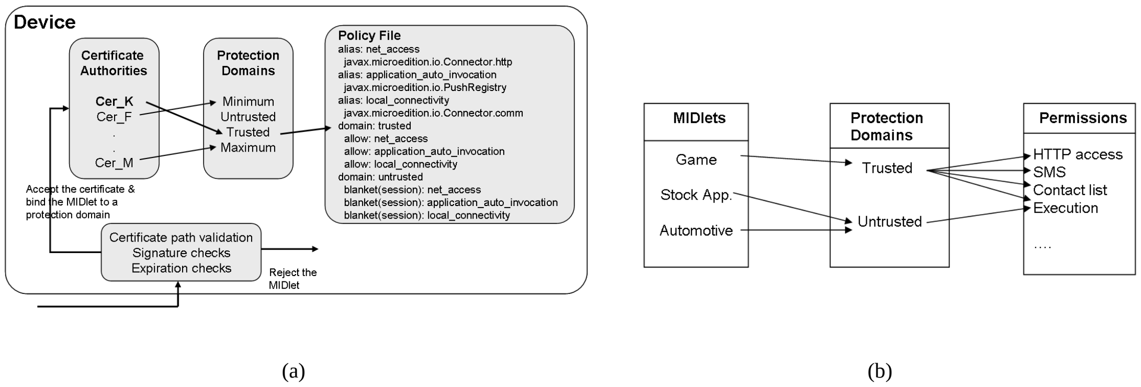
Trust model of J2ME is based on the ability to verify the origin and integrity of a MIDlet. The verification is based on certificate associated with the MIDlet suite. Should the certificate be known and verifiable against a trusted authority, the MIDlet is considered as trusted. Otherwise, should the user decide to proceed with MIDlet installation, it will be designated as untrusted. This effectively means the user shall be asked to explicitly confirm every sensitive operation [6]. Figure 2(a) shows the process of signing and assigning a MIDlet to a protection domain based on the trusted certificate. Typically, MIDP default (unmodifiable) policy consists of *trusted* and *untrusted* protection domains - associated with trusted and untrusted MIDlets respectively. Figure 2(b) shows the mapping from specific application to the assigned protection domain and associated security permission. For example, the untrusted MIDlets Stock App and Automotive are allowed to execute, while the trusted MIDlet Game is allowed HTTP access, sending SMSs and accessing the contact list.

Limitations of the J2ME (with respect to J2SE) that serve as the main motivation for our work can be summarized as:

- User can not modify the security policy (including the file), implying: predetermined permission set and predetermined domain set
- No ability to specify per-application policy
- SecurityManager is fixed and cannot be replaced by user

4.3 J2ME Security Model - Operational Aspects

With reference to the original J2ME security architecture, the modules involved in the process of making resource access decisions and their interactions are shown in Figure 3(a). Access requests are triggered when applications invoke respective, resource-related (e.g. communications channel, system settings, etc.) library calls. The security architecture of the JVM/KVM assures that the application may not bypass the JVM/KVM libraries and directly access native calls and protected operations. In other words, all *sensitive* calls must pass through libraries. On receiving a resource access request, the respective Java 2 library invokes the `SecurityToken` class to make the access decision. As Figure 3(a) depicts, to make an access decision, the SecurityToken module checks whether the particular domain to which the requesting MIDlet is assigned (as specified in *_policy.txt* file) includes the corresponding access granting permission. If the permission is ALLOWed, access is granted. If the permission is of type USER, the user is prompted for a decision. Otherwise, if nothing is specified in the policy file, access is denied and



```
alias: net_access
  javax.microedition.io.Connector.http
alias: application_auto_invocation
  javax.microedition.io.PushRegistry
alias: local_connectivity
  javax.microedition.io.Connector.comm
domain: trusted
  allow: net_access
  allow: application_auto_invocation
  allow: local_connectivity
domain: untrusted
  blanket(session): net_access
  blanket(session): application_auto_invocation
  blanket(session): local_connectivity
```

Figure 2. (a) Trusting a MIDlet suite and binding it to a protection domain (b) Mapping MIDlets to protection domains and to Java permissions

a `SecurityException` is thrown. The decisions thus derived can be referred to as *atomic* in the sense that no access history or other contextual attributes are considered. The policy itself, i.e. the protection domain specification, is loaded by the `Permissions` module. At execution time, the module answers queries on the permissions of the running MIDlets. The policy definition file itself comes preloaded on the virtual execution environment and can not be accessed for any type of modification. This effectively implies complete lack of flexibility with respect to protection domains structure and permissions specification.

5 Extended J2ME Architecture

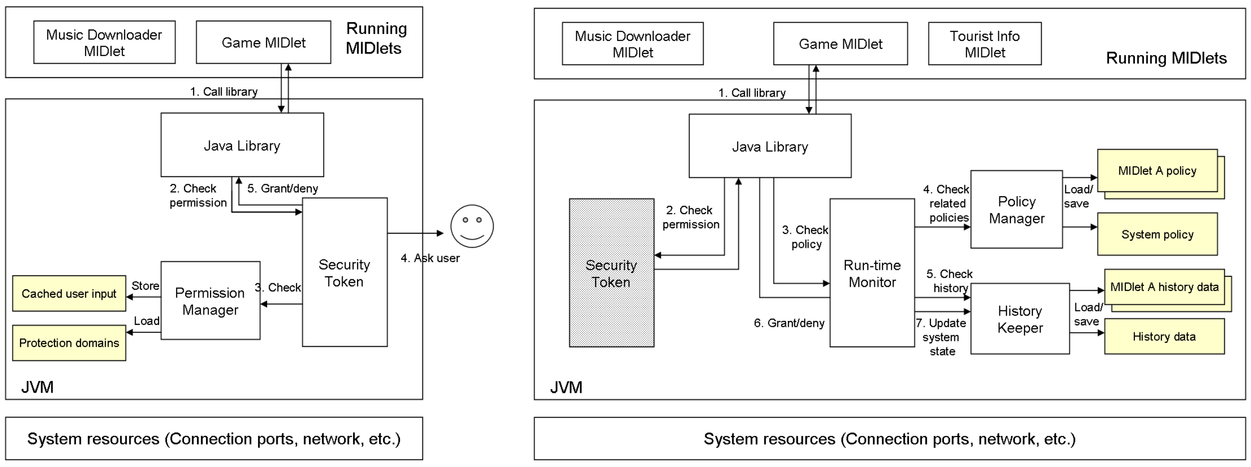
Having introduced the fundamental aspects of the J2ME architecture, in this section we present the extensions to the architecture and modification of its operational aspects that enable the support for fine-grained, history-aware, user-definable, per-application policy specification and enforcement. The contribution made in this section represents the basis for addressing the constraints of the J2ME security model specified in Section 4. We preserve the fundamental aspects of the J2ME security model, such as its domain-based nature. Figure 3(b) depicts the complete J2ME architecture incorporating our modifications.

As previously stated, the aim of the presented work is to provide the support for fine-grained, history-based, application-specific policy specification and enforcement within the J2ME framework. A simple example of a security policy that is supported by our extended J2ME architecture (xJ2ME) is "browser may not download more than 300 KB of data per day". Figure 3(b) depicts the new Java security architecture. The `Run-time Monitor` is in charge of making resource access decisions. In order to grant or deny resource access, the Run-time Monitor relies on the `Policy Manager` to identify the relevant application-specific policy. Once the policy is identified, the Run-time Monitor evaluates its conditions in conjunction with resource usage history information of the system and MIDlet, as obtained from the `History Keeper`. If the policy conditions are fulfilled, access is granted, otherwise a `SecurityException` is thrown.

In order to enable per-application policies, we associate each MIDlet with a specific policy which becomes an integral part of the corresponding MIDlet suite. This is in addition to the system-wide security policy. The `Policy Manager` is in charge of managing (reading, loading and interpreting) both MIDlet-specific and system-wide policies. We envisage the user to be able to set the desired policies on his mobile phone through a graphical interface.

To support policies based on historic resource access and usage, the Run-time Monitor must maintain a history of relevant system and application behavior. This is the role of the `History Keeper` module. Application-specific behavior information is retriegved at the MIDlet loading time, accumulated over its execution and stored in a persistent manner upon its termination. The system-related behavior information is accumulated throughout the system operation. The behavior history information is always stored in a secure manner - not accessible to MIDlets.

The introduced policy model does not in any manner affect the original J2ME security model. Instead, our contribution builds upon the J2ME model, retaining its domain-based nature and permission structure. The extended policy model actually allows extra constraints to be placed on per-application basis, further strengthening the overall J2ME security model.



(a) (b)

Figure 3. (a) Permission-based initial J2ME security architecture. (b)Extending the J2ME security architecture with the Run-time Monitor.

6 Policy Specification

The existing J2ME policy specification language, based around the concept of protection domains, is too constrained to allow for the expressiveness required by our policy model. To express the policies in our model, we leverage *Security Policy Language* (SPL) [9]. SPL is an event-driven, constraint-based, declarative policy language that supports access-control, history-based and obligation-based policies[2]. It also has inherent support for policy prioritization which enables us to specify per-application policies in the context of a wider platform policy which guides the overall behavior of the system, as described below.

6.1 SPL Policies and Rule Restriction

An SPL policy is defined as a group of rules that govern a particular domain of events. In our work, domains of events group events related to individual MIDlets. In other words, we specify a policy per MIDlet on the mobile platform. An example of a policy definition would be the following:

```
policy GameApp {

    gprs_limit: ce.target.permission = "GPRS" ::
        local_state.gprs_consumed < (local_grps_limit
        + ce.target.amount);

    sms_number_limit: ce.target.permission = "SMS" ::
        local_state.total_sms_sent < local_sms_number_limit;

    ?GameApp: gprs_limit AND sms_number_limit;  }
```

[2]Specification of SPL is outside the scope of this paper, interested readers can refer to [9]

The above example shows the use of local, application specific limits on resource consumption. The last statement in the policy definition, `?GameApp: gprs_limit AND sms_number_limit;`, represents the policy *query rule*. A query rule is the rule that implements the policy, i.e. the one that is invoked to obtain a decision with respect to the specific policy. The query rule is in itself a logic expression that combines, using the standard logic operators, individual rules comprising the policy.

In general, logic expressions are normally defined for two-valued (true/false) logics. However, since SPL uses a tree-valued (true/false/not-applicable) logic, it defines a suitable tree-value algebra in which the not-applicable value acts as a neutral element with respect to the standard two-value algebra for AND, OR and NOT logic operators.

As previously stated, we define local MIDlet policies in the context of a platform-wide, global system policy. The global policy is defined using the same syntax as for the local policies. However, a global policy by its nature cannot refer to local state of any MIDlet. Therefore, it is defined in terms of global (cross-MIDlet), cumulative state of platform resources and the corresponding resource consumption limits.

The overall "system" policy will, owing to the above, consist of a number of local, per-MIDlet policies and a single platform-wide policy, setting the global limits. In the context of SPL, individual policies are active only if instantiated and inserted in another policy - effectively forming a tree of active policies. The only exception to this rule is the *master* policy which represents the root of the trees. In our case, the master policy is represented by the platform-wide policy. Instantiation of a policy uses a syntax similar to creation of a new object in Java, as shown in the following example of a global policy:

```
policy PlatformPolicy {

  gprs_limit: ce.target.permission = "GPRS" ::
  global_state.gprs_consumed < (global_grps_limit
  + ce.target.amount);
  sms_number_limit: ce.target.permission = "SMS" ::
  global_state.total_sms_sent < global_sms_number_limit;

  GameApp_Policy: new GameApp;
  FORALL r in GameApp_Policy {
    r @ {ce.MIDlet = "GameApp"};
  };

  BrowserApp_Policy: new BrowserApp;
  FORALL r in BrowserApp_Policy {
    r @ {ce.MIDlet = "BrowserApp"};
  };

  ?PlatformPolicy: gprs_limit AND sms_number_limit AND
  GameApp_Policy AND BrowserApp_Policy;  }
```

In the above example we can see (starting from the top) two rules applying to the global state, followed by instantiation of two MIDlet specific policies (note the `new` keyword). `FORALL` block following each of the policy instantiations denotes the SPL *universal* quantifier, leveraged here to specify policy restriction. The expression following it specifies that all rules `r` are applicable (`@` operator) only when the access request comes from the respective MIDlet. Definition of such a constraint ensures that there is no conflicts among policies defined for different MIDlets.

6.2 Local and Global State Update

Finally, we leverage the concept of obligation in SPL to ensure in a verifiable manner that relevant state updates take place. The following is an example based on the previously defined global policy.

```
policy PlatformPolicy {

  gprs_limit: ce.target.permission = "GPRS" ::
  global_state.gprs_consumed < (global_grps_limit
  + ce.target.amount);
  gprs_state_update:
    EXISTS fe in FutureEvents {
      fe.action.target = "GPRS_STATE_UPDATE" &
      fe.action.value = ce.target.quantity :: true
    };
    gprs: gprs_limit AND gprs_state_update;

    sms_number_limit: ce.target.permission = "SMS" ::
    global_state.total_sms_sent < global_sms_number_limit;
    sms_state_update:
    EXISTS fe in FutureEvents {
      fe.action.target = "SMS_STATE_UPDATE" &
      fe.action.value = ce.target.quantity :: true
    };
    sms: sms_limit AND sms_state_update;

    ...

    ?PlatformPolicy: gprs AND sms AND
    GameApp_Policy AND BrowserApp_Policy; }
```

In SPL, the obligation policies use the existential operator (`EXISTS`) to condition an operation on a future event (represented by the set `FutureEvents`). Although Schneider [10] argues that policies conditioning present events on future ones are not enforceable, that does not apply for the cases in which both of the events are within the same atomic transaction. In our work, we assume that both policy evaluation and state update happen within the same atomic transaction (with ACID properties). In the above example, we require the future event to update the state by the value (`fe.action.value`) that corresponds to the amount of the resource used by the current, access requesting event (`ce.target.quantity`).

7 Implementation

In this section, we present the implementation specifics of the extensions made to J2ME to obtain the previously described xJ2ME architecture. For prototype xJ2ME implementation we used the recently released Sun's open source J2ME Reference Implementation (RI). The particular choice of target JVM to be extended was made based on the code availability, without any loss of generality. The presented architecture and prototype is fully portable to any J2ME implementation that adheres to the same original security architecture specification.

7.1 Background

Figure 4(a) shows the general call hierarchy of a Java virtual machine, including calls to the run-time monitor. To access resources, MIDlet calls the Java libraries provided by MIDP. The Java libraries, in turn, call the underlying native classes (e.g. C libraries). The calls to the security Run-time Monitor can, as opposed to the diagram in the figure, occur at different levels: i) at the Java MIDP library level (as in the figure); ii) at the native library level; or iii) beneath the running VM, intercepting the OS API calls.

We decided to implement the run-time monitor at the Java library level (as depicted in the figure) for two main reasons. Firstly, while Java library modifications produce completely portable solutions, both of the alternative approaches require platform specific knowledge. Secondly, Java security checks occur at a higher level of abstraction, eliminating considerable complexity in the implementation phase. The decision to perform the checks at the Java library level is also in-line with the original J2ME security architecture, further simplifying the implementation.

7.2 Run-time Enforcement

In MIDP 2.0 RI, the APIs that offer access to the resources relevant from the point of view of our security policy (e.g. SMS, TCP, HTTP etc.) are implemented in the form of `Protocol` classes in the corresponding packages (e.g. `com.sun.midp.io.j2me.`**`sms`**`.Protocol`). For example, in the case of the TCP stack, the Protocol class implements methods to open/close connections and send/receive data. As shown in Figure 4(b), all of the resource-specific Protocol classes extend the `ConnectionBaseAdapter` class. This class implements a unique method (`checkForPermission`) to check for permission to perform the requested operation. However, in the original J2ME implementation, the policy is not evaluated on every request to access a resource - if the policy grants the permission for the duration of the session, the access decision is cached upon the first check and used for any subsequent requests.

Our extended security architecture follows the the original class interaction model. However, we have augmented the functionality of the `ConnectionBaseAdapter` by providing a method to check policy (`checkPolicy`) which is to be invoked by individual `Protocol` classes. To support the various policy rule constraint types that we envisage in our policy model, we also extended the `ConnectionBaseAdapter` class to hold the relevant information related to the corresponding resource access requests. For example, such information includes, in the case of a request to send SMS, the number to which to send the SMS, the size of the text to be sent, etc. For a TCP connection, this information includes the URL of the connection destination and the amount of data to send.

To make an access decision, once invoked through the `checkPolicy` method, the run-time monitor needs not only the type of the request (permission required) but also the identity of the MIDlet making the request (i.e. the subject). In the J2ME architecture, the latter is unknown at the library level. Thus, to obtain this information, in xJ2ME the policy check calls are performed indirectly, at the level of MIDlet scheduler (`MIDletSuite` and `Scheduler` classes). As shown in Figure 4(b), it is the `MIDletSuite` class, rather than the respective `Protocol` class, that invokes the run-time monitor. If the access is denied, a `SecurityException` is thrown. Otherwise, the library proceeds and performs the requested operation.

Furthermore, as our policy model relies on historic information, both MIDlet-local and global, no caching of access decisions takes place. In other words, policy evaluation is performed on every resource request. The evaluation shows that such approach does not introduce significant, or even noticeable, performance impact.

Since our policy model allows for policy expression based on historic information, both the global and local system states need to be kept. This is the role of the newly introduced `HistoryKeeper` class. If a requested permission is granted, the `HistoryKeeper` updates the relevant state variables, but only after the operation for which the permission has been requested is performed successfully. Otherwise, the system state remains unchanged. The policy model presented in Section 6 accounts on these transaction-like properties of the system to be able to express obligation rules conditioned on the occurrence of a future event.

7.2.1 Threading Issues

The reliance on the historic information to derive access decisions in multi-threaded environments introduces problems of potential race conditions on the system state. In other words, all accesses to the system state (local or global) must be implemented in a thread-safe manner. Otherwise, thread interleavings in the state-check and state-update phases may cause policy violation. For illustration, consider two threads, `A` and `B`, that both want to send an SMS. Let also the constraint on sending SMS allow only one more SMS to be sent. In this case, if the thread `A` is preempted after the state check but before the state update, and the thread `B` is scheduled to perform the state check, both would be allowed to perform the requested operation - causing a clear violation of the policy.

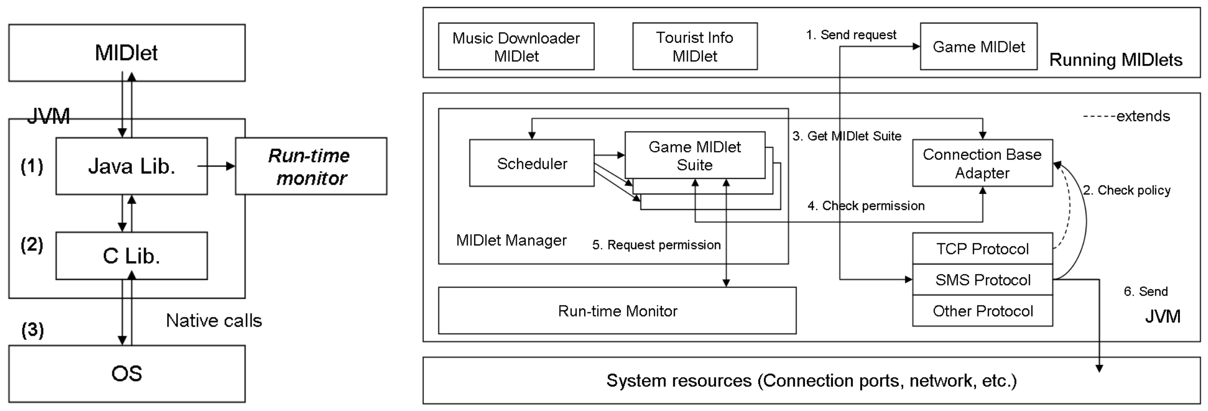
There are two solutions to the above-outlined problem. The first consists of thread serialization while the second relies on locking of relevant state variables. In the current implementation we opted for the latter one. The main reason for the choice was that we did not envisage nested state variable lock requests to occur - much simplifying the chosen approach. The approach in which `HistoryKeeper` locks relevant state variable does introduce potential delays in long-lasting resource access operations. This, for example, may arise in case of blocking send operations over TCP connections. However, taking in consideration the scope of the policy model, in terms of security-relevant resources that we are interested in, we do not expect this to be a major issue. Having said this, solving the thread-safety issue in a consistent form is one of the top items on the future research agenda.

7.3 Porting MIDP

We have integrated the presented architecture on two open-source MIDP implementations, MIDP 2.0 Reference Implementation [3] and PhoneME Feature [4]. The modularity of the architecture makes it easy to incorporate with other MIDP distributions as well, as they all follow the MIDP 2.0 specification [2].

Our implementation is currently running as emulator on Desktop PCs, both under Linux and Windows OS, and on

[3]MIDP 2.0 RI: http://wwws.sun.com/software/communitysource/j2me/midp/download20.html



(a)

Figure 4. (a) Run-time Monitor implementation at different levels: (1) Java (2) C (3) native OS calls (b)Checking security policy implementation

a PDA running Linux Embedded OS. The MIDP emulator for Desktop PCs comes with an integrated user interface which simulates mobile phones and its user-interaction model, with different skins. Furthermore, the CLDC and MIDP porting guides give extensive instructions on porting to new platforms.

The phoneME software is designed to run on or be easily ported to virtually any mobile device. For example, several ports for Linux Embedded and Windows CE devices are already in the phoneME distribution. After modifying the sources, one must rebuild the project and compile it for the target machine. Our extended J2ME prototype is running on an JASJAR I-Mate [8], with 64MB RAM and a Intel Bulverde 520 MHz processor (also knows as xScale ARM processor). Apart from the default Windows Mobile operating system that comes with the device, we have installed a dual boot Familiar Linux version 2.6,which is loaded from a 252Mb SD Memory Card [4].

The window manager that we installed on the devices is Opie [3] version 1.2. Opie is an open-source graphical user interface, developed as a fork from the commercial application environment Qtopia.

We have incorporated the fine-grain policy based security architecture and cross-compiled phoneME features based on the Qt/Embedded graphical libraries [5] for the xScale ARM processor of our i-mate device. A cross-compiler is a compiler capable of creating executable code for a platform other than the one on which it is run. To build the GCC and GNU cross-compiler we used the cross-tool scripts [5].

8 Evaluation

The improvement over the state-of-the-art brought about by our work lies in two dimensions: flexibility of the security models for mobile computing platforms and the granularity at which the policies can be specified and enforced. The flexibility of the model is proven by example, while the policy enforcement mechanisms lend themselves to being evaluated empirically.

It is important to state that owing to the J2ME architecture, MIDlets cannot access low level APIs directly. This implies that by placing run-time monitor hooks within the Java library code (rather than inlining them using bytecode rewriting) we do not open up space for bypassing of the security mechanisms.

As Section 4.2 presented, the previous J2ME permission model literally offered no flexibility in terms of defining and managing permissions, customizing protection domains and applying per application policies. In contrast, the proposed security architecture provides extensive flexibility for fine-grained security policies and control over the application's behavior.

The original J2ME security architecture resorts to user consultation on every request for a permission not explicitly allowed in the respective protection domain. The presented fine-grained policy model and security architecture enhances the user experience and the control she has over

[4]Universal: how to install Linux http://www.handhelds.org/moin/moin.cgi/UniversalHowtoInstallLinux

[5]Crosstool: http://kegel.com/crosstool/

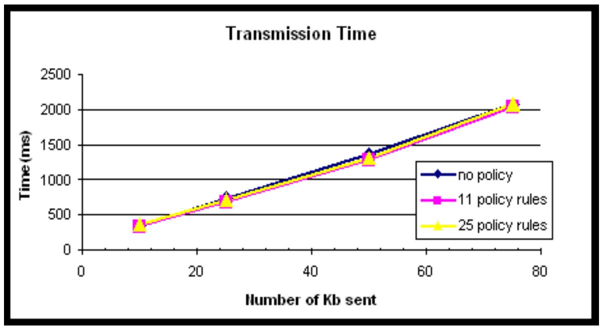


Figure 5. Performance tests: The overhead introduced by the extra policy check when sending data is unnoticeable.

the behavior of applications running on the system. For example, instead of manually counting the number of SMSs a MIDlet has sent when being prompted for confirmation each time, the user may now rely on an adequate policy.

We used the JASJAR i-mate with the configuration presented in Section 7.3 as a testbed for performance measurements. To measure the overhead introduced by the extra policy check, we conducted the following experiment. We measured the time needed by a MIDlet to send data, when no policy, 11 and 25 policies are defined. The number of policies defined on a mobile device is generally bound by the number of resources. As the number of resources is by itself limited, we expect the number of policies to remain reasonably low, as the values we have chosen for testing. To avoid the noise introduced by network delays, we start the TCP server locally, on the JASJAR i-mate. The MIDlet Client establishes a connection to the server and sends a variable amount of data, in chunks of 1Kb. This means that the policy check is done for each Kb sent. Therefore, if the client sends 50Kb, the library invokes the policy check invoked 50 times. Once the connection is established, we measured the average time needed by the client to send 10Kb, 25Kb, 50Kb and 75Kb of data. To compare the performance of the virtual machine, we considered the cases (1) no policy check is performed (previous MIDP architecture), (2) 11 policy rules to be checked and (2) 25 policy rules are defined. As Figure 8 depicts, the average time to send 10Kb of data is 350ms and 75Kb is 2 seconds, in all three cases. The overhead introduced by the policy check is unnoticeable compared with the time needed to perform the security relevant operation. It turns out that integrating the Java monitor into the virtual machine architecture and having it triggered by the Java MIDP libraries results in excellent performance.

9 Conclusions and Future Work

This paper has presented a practical extension to the Java virtual machine for mobile devices that supports fine-grained security policy and enforces them through run-time monitoring. By doing so, and by having proven an efficient and viable solution, it addresses the users' need for application control and opens the possibility of a new generation of mobile services and applications.

Although the presented model for Run-time Monitor has been implemented for the MIDP profile, the introduced architecture concepts can be applied to other J2ME profiles as well. An alternative to porting the J2ME Sun RI to run on smart phones and PDAs is to incorporate the run-time monitor in the proprietary Java implementations of specific operating systems. Since the extended security architecture is enforced only at the level of Java libraries and modules, the modifications done do not affect the KVM nor the operating system. Therefore, the Run-time monitor should be easily portable to alternative implementations.

In the future, we plan to deploy the prototype to a number of platforms and design graphical paradigms for specifying policies in a user friendly manner. In addition to the policy management tools, we plan to introduce support for defining and handling policies per groups of application. Furthermore, at the time of writing of this paper we are working on the multi-threading issue outlined in Section 7.

References

[1] Java ME, Java Platform Micro Edition, Sun Microsystems. Available from http://java.sun.com/javame/index.jsp/.

[2] JSR 118: Mobile Information Device Profile 2.0, Sun Microsystems. Technical report. Available from http://jcp.org/en/jsr/detail?id=118.

[3] Opie - Open Palmtop Integrated Environment Applications and libraries for mobile devices. http://opie.handhelds.org/cgi-bin/moin.cgi/.

[4] PhoneME. https://phoneme.dev.java.net/.

[5] Qt/Embedded. http://www.trolltech.com/download/qt/embedded.html.

[6] M. Debbabi, M. Saleh, C. Talhi, and S. Zhioua. Security Evaluation of J2ME CLDC Embedded Java Platform. *Journal of Object Technology*, 5(2):125–154, 2006.

[7] L. Gong, G. Ellison, and M. Dageforde. *Inside Java 2 platform security architecture, API design, and implementation (2nd Edition)*. Prentice Hall PTR, May 2003.

[8] JASJAR i-mate Technical details. Available from http://www.imate.com/t-JASJAR_technical.aspx.

[9] C. Ribeiro and P. Guedes. SPL: An access control language for security policies with complex constraints. In *Network and Distributed System Security Symposium (NDSS'01), San Diego, California*, February 2001.

[10] F. B. Schneider. Enforceable security policies. *ACM Transaction on Information and System Security*, 3(1):30–50, 2000.