Monitoring External Resources in Java MIDP

David Aspinall Patrick Maier[1] Ian Stark

Laboratory for Foundations of Computer Science
School of Informatics, The University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ, United Kingdom

Abstract

We present a Java library for mobile phones which tracks and controls at runtime the use of potentially costly resources, such as premium rate text messages. This improves on the existing framework (MIDP — the Mobile Information Device Profile [6]), where for example every text message must be authorised explicitly by the user as it is sent. Our resource management library supports richer protocols, like advance reservation and bulk messaging, while maintaining the security guarantee that attempted resource abuse is trapped.

Keywords: Runtime Monitoring, Resource Control, Java MIDP, Security.

1 Introduction

Modern mobile phones are powerful computers. Their primary task, providing mobile wireless telephone services, is comparatively losing importance as they are being used for a range of other applications, from personal information managers to web browsers, from media players to games. Most of these applications access the network [2], either because it is integral to their functionality (e. g. web browsers, online games), or because networking is adding desired features (e. g. playing streaming media or synchronising diaries).

The cost of the standard computational resources, like execution time or memory space, is determined solely by the computational device (i. e. the hardware of the mobile phone) itself. The cost of network access, however, is determined by external entities, e. g. the business model of the phone operator, which is why we classify network access as an *external resource*. Moreover, it is a resource the spending of which users generally would like to control tightly because it costs them money. The last point actually goes double: If network access is maliciously exploited it could be very expensive, but even if it is not exploited, users care about each 10p [3], i. e. they want to know the exact cost beforehand.

In MIDP [6], the current standard framework for Java applications on mobile phones, monitoring external resources, like communication via text message, is left to the user, as

[1] Email: pmaier@inf.ed.ac.uk

[2] Refers to the operator's mobile phone network; access to other networks (like the Internet) is routed through this one.

[3] The standard cost of sending a text message in the United Kingdom.

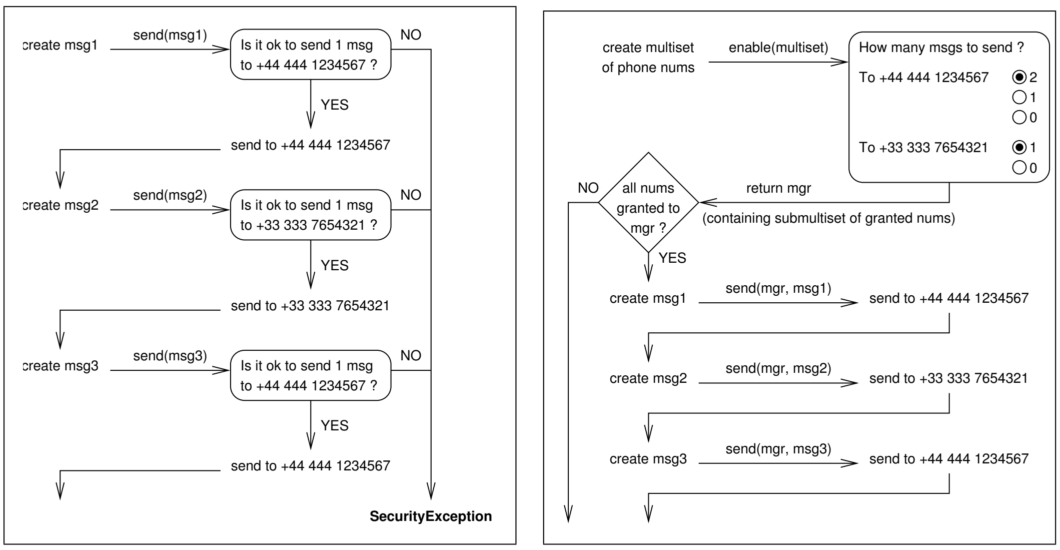

Fig. 1. Transaction sending 3 text messages; in MIDP 2.0 (left) and with explicit resource management (right).

illustrated by the flowchart on the left hand side of Figure 1. For each of the three messages, the application pauses to ask the user for authorisation before sending. This one-shot authorisation is clearly prohibitive for applications wishing to send many messages because users will get annoyed by the many pop-up screens, which malicious applications may exploit to trick users into authorising messages to premium rate numbers. Such social engineering attacks [12] have been reported in the wild [14]. Yet, even if an application sends only few messages, one-shot authorisation can lead to undesirable results, like transactions aborted midway by an exception because the user stops authorising messages (see the left hand side of Figure 1).

We propose explicit accounting and monitoring of external resources to better protect the user from accidental or malicious resource abuse. Our approach revolves around *resource manager* objects, which keep an account of which external resources an application is granted to use and how often. The right hand side of Figure 1 illustrates this on the messaging example. Before sending messages, the application computes a multiset of phone numbers encoding how many messages it will send to which recipients. In a single authorisation dialogue the user then gets to decide how many messages the applications may send to whom. This information (a submultiset of the multiset of requested numbers) is stored in a resource manager. The application only proceeds if all the requested numbers have actually been granted, in which case it calls instrumented methods for sending the messages, taking an extra resource manager argument, which monitors the resources being spent (and would abort the application if it was overspending).

Explicit resource management has additional benefits besides runtime monitoring. It forces the application to determine early on how many resources to request. It provides a clear user interface by centralising the choice of which of the requested resource to grant into a single dialogue. Plus, it enables the application to react flexibly to the amount of resources it has been granted, i. e. the application can choose whether it is feasible to continue with the resources granted or whether it has to abort because of insufficient resources.

The rest of this paper is structured as follows. Section 2 gathers some facts about MIDP which are relevant to us. Section 3 introduces the resource management library, which Section 4 extends by adding policies. Section 5 describes the security properties that library guarantees and outlines a deployment scenario. Section 6 discusses related work, and Section 7 concludes.

2 Background: The MIDP Security Model

The *Mobile Information Device Profile* (MIDP, current version 2.0 [6]) is the current standard framework for Java applets (also called *MIDlets*) on networked mobile devices. MIDP builds upon the *Connected Limited Device Configuration* (CLDC, current version 1.1 [7]). Together, CLDC and MIDP, which are part of the *Java Micro Edition Platform* (Java ME), define a set of APIs for programming small devices like phones and PDAs. With security in mind, they restrict Java in several ways. In particular, reflection and custom class loading are not supported; all of a MIDlet's classes must be loaded from a single JAR using the standard CLDC class loader, which renders possible to statically check the MIDlet's classes for certain properties (see Section 5.4).

As of MIDP 2.0, access to sensitive APIs and functions (e. g. for sending text messages) is regulated by a permission-based security model. MIDlets are bound to *protection domains* based on whether and by whom they are signed (where a signature expresses the signer's trust in the MIDlet but does not provide any guarantees about the code itself). Each protection domain holds a set of *permissions*, each of which is either flagged as *Allowed* or *User*. The former grants unconditional access whereas the latter requires access to be authorised by the user. How often this authorisation has to be obtained depends on whether a *User* permission is flagged as *Blanket*, *Session* or *OneShot*; the latter requires authorisation for every single access.

According to the MIDP specification, only MIDlets signed by the device manufacturer or the network operator may obtain unconditional access to cost-sensitive functions (e. g. for sending text messages). The protection domains for other MIDlets must insist on OneShot authorisation for access to these functions. As a consequence, MIDlets wishing to use messaging more than just occasionally are faced with the choice of either having to be signed by the operator (or manufacturer) or having to annoy their users with lots of authorisation screens.

3 Basic Resource Management API

This section presents an API for monitoring the use of external resources. The API introduces special objects, called *resource managers*, which encapsulate multisets of resources that a MIDlet may legally use (according to the user's approval) and which are passed as arguments into instrumented MIDP methods that actually use the resources. These methods, e. g. the method for sending text messages, check the resource manager before consuming the resources. If the required resources are not present, the instrumented methods abort the MIDlet with a runtime error.

3.1 Resource Managers

Figure 2 shows a class diagram of resource management package. The core of the API is the final class `ResManager`, which encapsulates a multiset of resources and whose meth-

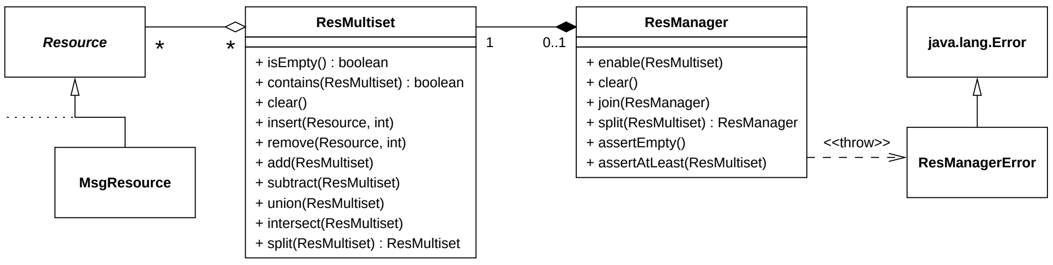

Fig. 2. UML class diagram of the basic resource management API. All terminal (w. r. t. generalisation) classes are final.

ods are explained below. The final class `ResMultiset` provides modifiable multisets of resources, with the usual operations on multisets, including multiset intersection, sum and inclusion. Internally, multisets are realised by hash tables, mapping resources to multiplicities (which may be infinite). Every `ResMultiset` object encapsulates its mutable state, so that it cannot be changed other than by calling its public methods. The abstract class `Resource` serves as an abstract type for resources; actual resources (e. g. the class `MsgResource` representing the permission to send one text message to a given phone number) must be final subclasses. Being used as keys in hash tables, resources must abide by the following contract: They must be immutable objects, and resources constructed from the same arguments must be indistinguishable by the `equals` method.

The class `ResManager` encapsulates a multiset of resources via a private field `rs` of type `ResMultiset`. All public methods are synchronised to avoid races in case different threads access the same resource manager. The table below lists the methods with a JML-style[4] semantics, where the symbols $\subseteq$, $\uplus$ and $\cap$ stand for multiset inclusion, sum and intersection, respectively.

	requires	**ensures**	**modifies**
ResManager()	*true*	this.rs $= \emptyset$	this.rs
void enable(ResMultiset req)	*true*	this.rs $\uplus$ req $=$ \old(this.rs) $\uplus$ \old(req) $\wedge$ req $\subseteq$ \old(req)	this.rs, req
void clear()	*true*	this.rs $= \emptyset$	this.rs
void join(ResManager mgr)	*true*	this.rs $=$ \old(this.rs) $\uplus$ \old(mgr.rs) $\wedge$ mgr.rs $= \emptyset$	this.rs, mgr.rs
ResManager split(ResMultiset bound)	*true*	\fresh(\result) $\wedge$ \result.rs $=$ \old(this.rs) $\cap$ bound $\wedge$ \result.rs $\uplus$ this.rs $=$ \old(this.rs)	this.rs
void assertEmpty()	this.rs $= \emptyset$	*true*	\nothing
void assertAtLeast(ResMultiset bound)	bound $\subseteq$ this.rs	*true*	\nothing

The `enable` method takes a multiset `req` of requested resources and lets the user decide (in a pop-up dialogue) how many of these resources to add to the manager's multiset `rs`. As a side effect, `enable` modifies its argument `req`; upon return from `enable`, the MIDlet should check `req` to learn which of the requested resources it is being denied; in particular, if `req` is empty then all of the requested resources have been granted.

The methods `clear`, `split` and `join` provide some control over the contents of a resource manager, by consuming all its resources, transferring some resources to a new

[4] The \operators generally bear the same meaning as in JML [11], except that `\old(e)` refers to the pre-state of expression `e` in the *pre*-state of the heap.

```
void sendBulk(MessageConnection conn,
              Message msg,
              PhonebookEntry[] grp)
{
  ResMultiset rs = new ResMultiset();
  for (int i=0; i < grp.length; i++) {
    String num = grp[i].getMobileNum();
    rs.insert(new MsgResource(num), 1);
  }

  ResManager mgr = new ResManager();
  mgr.enable(rs);

  if (rs.isEmpty()) {
    for (int i=0; i < grp.length; i++) {
      String num = grp[i].getMobileNum();
      msg.setAddress(num);
      conn.send(mgr, msg);
    }
    mgr.assertEmpty();
  }
  else mgr.clear();
}
```

```
public void send(ResManager mgr, Message msg)
throws IOException, InterruptedIOException
{
  synchorized (msg) {
    String num = msg.getAddress();
    ResMultiset rs = new ResMultiset();
    rs.insert(new MsgResource(num), 1);

    ResManager local_mgr = mgr.split(rs);
    local_mgr.assertAtLeast(rs);

    try {
      send(msg);
      local_mgr.clear(); local_mgr = null;
    } catch (InterruptedIOException e) {
      local_mgr.clear(); local_mgr = null;
      throw e;
    } catch (IOException e) {
      mgr.join(local_mgr); local_mgr = null;
      throw e;
    }
  }
}
```

Fig. 3. Bulk messaging example, left: MIDlet code, right: instrumented MIDP method.

manager, or joining the resources in two managers, respectively. Thanks to `split` and `join`, the MIDlet may keep resource managers thread local, avoiding contention over shared managers.

The assertion methods check whether their preconditions hold. If so they behave like no-ops, otherwise they throw an instance of `ResManagerError`. The latter case must be seen as a violation of the MIDlet's own logic (much like failing an assertion), and the MIDlet should not be allowed attempts at repairing the situation (by catching the error), which is why `ResManagerError` extends `java.lang.Error` rather than `java.lang.Exception`.

3.2 *Example: Bulk Messaging MIDlet*

We illustrate the use of resource managers by an example application built on top of the *Wireless Messaging API* (WMA, current version 2.0 [8]), a bulk messaging MIDlet, which lets the user send a text message to a group of recipients from his phone book. Figure 3 (left column) shows the MIDlet's method that actually sends the message. The method takes an (already open) message connection, a message and a group of recipients (represented as array of phone book entries). First, the MIDlet builds up a multiset of resources `rs` by iterating over the group of recipients and for each one, extracting the mobile phone number, converting it into a resource by constructing an instance of `MsgResource`, and adding one occurrence of that instance to the multiset. Next, the MIDlet creates an empty resource manager `mgr` and enables it to use the resources in the multiset `rs`. This will pop up a confirmation dialogue box where the user can approve or deny the planned resource usage, modifying `rs` as a side effect. Only if the user approves of all messages to be sent, i. e. if `enable` returns its argument `rs` empty, does the code proceed to the actual send loop. The send loop again iterates over the group of recipients, extracting for each one the mobile phone number, setting the address field of the message and sending the message using the instrumented `send` method, see below. After the loop, `assertEmpty` checks that the resource manager `mgr` is really empty, i. e. all enabled resources have been used. (Instead of checking, the manager could have been cleared explicitly, like in the else branch, to prevent unintended later use of left-over resources.)

3.3 *Instrumented Methods*

Resources are consumed by specific methods, e. g. in the case of messaging by the method `send(Message)` declared in the WMA interface `MessageConnection`. To monitor whether these methods consume only resources that have been granted, we wrap them with instrumentation code checking whether a given resource manager holds the required resources. These instrumented methods are declared in sub-packages of the resource management package.

To instrument messaging, we have to augment MIDP and WMA in three places. We supplement the WMA interface `MessageConnection` with a new wrapper method `send(ResManager,Message)`, provide a class which implements this extended interface, and revise the MIDP method `Connector.open` to return the new class.

The code for the wrapper method is shown on the right-hand side of Figure 3. It extracts the phone number `num` from the message and constructs a multiset `rs` containing a single occurrence of the resource corresponding to `num`. Then it splits the resources in `rs` off from the resource manager `mgr` and stores them in the new local resource manager `local_mgr`, which is checked for containing at least the resources in `rs`. If this check fails a `ResManagerError` will be thrown, aborting the calling MIDlet; if the check succeeds we know that `local_mgr` holds exactly the resources in `rs`. Finally, the message is actually sent by calling the uninstrumented `send` method.[5] Clearing `local_mgr` and nulling the reference afterwards is not strictly necessary but considered good practise; it signals that the resources in the local manager are now used up and that the manager itself is ready to be reclaimed by garbage collection.

In case of a send failure, the event that actually spends the resources (i. e. delivering the text message to the operator's network) may or may not have happened yet. We assume that an `IOException` is thrown before actually sending the message (e. g. because the connection to the operator's network is down), so the resources are not yet consumed, and the handler can return them to the caller (by joining the local manager to `mgr`) before propagating the exception. However, if an `InterruptedIOException` is raised, we do not know whether the send event has already happened, so we assume that the resources are already spent. In this case, the handler consumes the resources (by clearing the local manager) before propagating the exception.

Note that the instrumented `send` method method must synchronise on `msg`, which is accessed twice, but there is no need to synchronise on `mgr` (for there are no data dependencies between the first and second access) or on **`this`** (for it is accessed only once).

3.4 *Runtime Overhead*

Monitoring of external resources does cause some runtime overhead. In terms of execution time, the overhead is negligible, as very little time is spent on the instrumentation compared to what is spent on actually consuming the resource (e. g. transmitting a message). Due to the hash table based implementation of multisets, all operations on resource managers take (at most) linear time w. r. t. to the size of the multisets involved. In fact, the overhead of the instrumented send method in Figure 3 is constant because the argument of `assertAtLeast` is a singleton multiset.

[5] Depending on the MIDlet's protection domain, the uninstrumented `send` method may again ask the user to authorise sending the message; Section 5.4 addresses this shortfall.

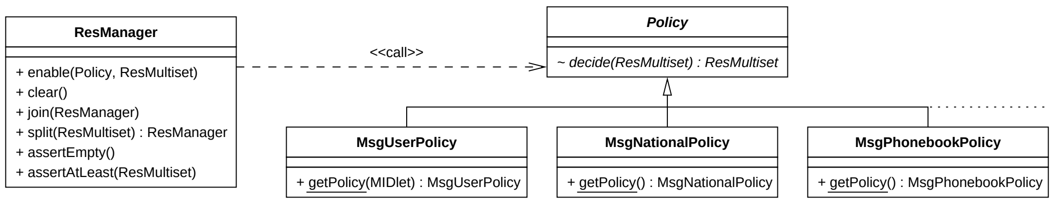
ResManager
+ enable(Policy, ResMultiset)
+ clear()
+ join(ResManager)
+ split(ResMultiset) : ResManager
+ assertEmpty()
+ assertAtLeast(ResMultiset)
<<call>>
Policy
~ decide(ResMultiset) : ResMultiset
MsgUserPolicy
+ getPolicy(MIDlet) : MsgUserPolicy
MsgNationalPolicy
+ getPolicy() : MsgNationalPolicy
MsgPhonebookPolicy
+ getPolicy() : MsgPhonebookPolicy

Fig. 4. UML class diagram of policy extension of resource management API. All terminal classes are final.

In terms of memory, the overhead may be more severe, particularly on small devices, because of the memory requirements of the hash tables. Additionally, resource monitoring puts a higher strain on garbage collection because the instrumentation code temporarily allocates resources, multisets and managers. If runtime checking is not necessary or desired, it can be switched off by “erasing” resource managers (see Section 5.2), which reduces the memory overhead significantly.

3.5 Extensibility

By design, the resource management API is extensible. Monitoring new resources (e. g. the number of bytes sent over a TCP/IP connection, or the space available in the persistent record store) simply amounts to adding new resource types plus adding the appropriate instrumentation. New resource types are added by extending the abstract class `Resource` with final subclasses, which abide by the contract on resources. Instrumented methods, which monitor the new resources before calling the corresponding uninstrumented methods, are added to sub-packages of the resource management package.

4 Extending the API with Flexible Policies

So far, the `enable` method involves the user, who is selecting to-be-added resources in a pop-up dialogue. That is, the user is acting as a *policy oracle* deciding which resources to grant and which to deny. In this section, we extend the API to include more flexible policy oracles, not just the user.

4.1 Changes to the API

Figure 4 shows the class diagram of the extension. It adds an abstract class `Policy` providing an abstract, package private method `decide` for deciding which resources to grant and which to deny. The table below shows the formal, non-deterministic semantics of `decide`; granted resources are returned in a new multiset, denied resources are returned via the modified argument.

	requires	**ensures**	**modifies**
ResMultiset decide(ResMultiset req)	*true*	\fresh(\result) $\wedge$ \old(req) $=$ req $\uplus$ \result	req

Actual policies must be final subclasses of `Policy` and must provide a package private implementation of `decide`. The latter requirement ensures that `decide` can be called by the resource management library only, not directly by MIDlets themselves. For a MIDlet to gain access to policies, each subclass of `Policy` provides a static `getPolicy` method which hands out the requested policy (i. e. an instance of the respective class) or **`null`** if the calling MIDlet is not authorised to use the requested policy.

MIDlets can only pass policies as arguments to other methods, in particular to the `enable` method of class `ResManager`, which consults its policy argument as an oracle to decide which resources to grant and which to deny, and which interprets a **`null`** argument as the deny-all policy, see the implementation below. Note that `enable` defers synchronisation on **`this`** as long as possible (i. e. until accessing the manager's encapsulated multiset `rs`) to avoid locking the manager during a call to `decide`, which may block for a long time (e. g. if the policy consults the user).

```
public void enable(Policy p, ResMultiset req) {
  if (p == null) return;
  synchronized (req) {
    ResMultiset granted = p.decide(req);
    synchronized (this) { rs.add(granted); }
  }
}
```

4.2 Use of Policies in MIDlets

The basic resource management API knew only one implicit policy: ask the user. Yet, typically each resource type has its own policy or policies. The policies for `MsgResource` include a `MsgUserPolicy`, which behaves like the implicit policy of the basic API, asking the user how many messages to send to which phone numbers. To use this policy, the call `mgr.enable(rs)` in the bulk messaging MIDlet (Figure 3) must be replaced by `mgr.enable(MsgUserPolicy.getPolicy(this), rs)`.[6]

There could be other policies for `MsgResource`, e. g. a `MsgNationalPolicy`, which grants only messages to national phone numbers. This policy could be combined with `MsgUserPolicy` by chaining calls to `enable` as in the following code snippet.

```
mgr.enable(MsgNationalPolicy.getPolicy(), rs);
mgr.enable(MsgUserPolicy.getPolicy(this), rs);
```

The first call enables all requested messages to national numbers, without asking the user. The second call asks the user to authorise the messages to the remaining (international) numbers. In the end, `rs` contains only those international numbers that the user has denied.

Another interesting policy for messaging could be a `MsgPhonebookPolicy`, which automatically grants all messages to numbers in the user's phone book. If the bulk messaging MIDlet used this policy, the user would not have to confirm anything. In return, the MIDlet could maliciously send more messages than the user intended, but only to phone numbers in the user's phone book, not to premium rate numbers (unless the MIDlet was allowed to modify the phone book).

4.3 Extensibility

By design of the API, adding new policies simply amounts to extending the abstract class `Policy` with final subclasses, which abide by the contract on policies: No public fields and methods (in particular, `decide` must be package private) except the static `getPolicy` methods, and the implementation of `decide` must agree with the formal semantics as shown in the table in Section 4.1.

[6] `MsgUserPolicy.getPolicy` requires an argument of type `MIDlet` so that the policy can access the MIDlet's screen.

5 Security Properties of Explicit Resource Management

This section informally summarises and motivates the security guarantees provided by the resource management API and a trusted library implementing it.

5.1 No Abuse of Resources

Property 1 *MIDlets using the resource management API cannot consume more resources than granted; any attempt to do so will result in the MIDlet being aborted before the abuse happens.*

The property holds for two reasons.

(i) Before performing any actions, the instrumented methods, e. g. the `send` method from Section 3.3, check their `ResManager` argument for the required resources and throw a `ResManagerError` (which will abort the MIDlet) if there aren't enough. If there are enough resources, the instrumentation deduces the required amount from the resource manager, even if the underlying uninstrumented method throws an exception.

(ii) The implementation of the resource management API ensures that policies cannot be bypassed. Resources may be moved back and forth between managers by the methods `split` and `join`, but there is no way to sneak new resources into the managers other than by calling `enable`, in which case a policy gets to decide which resources to grant and which to deny. Furthermore, the implementation confines the multiset held by a manager, i. e. it ensures that there are no pointers from outside a manager into its mutable state, hence a manager's multiset cannot be modified from the outside.

Of course, the above argument assumes that the MIDlet does not bypass or subvert the resource management library itself; see Section 5.4 on how to ensure this.

5.2 Erasure

Tracking the use of resources with resource managers does induce some overhead, mainly in terms of the memory required for storing the multisets. On small devices, one might want to avoid this overhead if a MIDlet is known to be *resource safe*, i. e. if it cannot ever throw a `ResManagerError`. In this case, resource managers can be "erased".

Erasure cannot be performed as a simple source code transformation removing all occurrences of resource managers from a MIDlet, for two reasons. First, MIDlets must be able to access resource managers in order to call the `enable` method, even after erasure, to let a policy decide which resources to grant. Second, resource managers may appear in conditions like `(mgr1 == mgr2)`, from where they cannot be removed unless the condition can be evaluated statically. What can be done, however, is a "soft" erasure, which keeps the managers themselves in place but erases their multisets, resulting in very lightweight *erased* resource managers.

Soft erasure can be achieved by retaining the public interface of class `ResManager` but replacing its implementation with a stateless dummy implementation. More precisely, erasure removes the private field `rs` (storing the manager's multiset), which turns all public methods into no-ops, except for `split` and `enable`. The latter still calls the policy and reports the denied resources back to the MIDlet, whereas the former creates a fresh (erased) manager.

Property 2 *If a MIDlet is resource safe then erasing the resource managers does not change its observable behaviour.*

The property holds because by design of the resource management API, the value of a resource manager can only affect the values of other resource managers; it cannot affect the values of other types.

Note that an optimiser can eliminate all of the calls on erased resource managers, except calls to `enable`, by inlining. As a result, resource managers may become unused and can be optimised away. In fact, a clever optimiser could optimise away the entire instrumentation code from the instrumented `send` method in Figure 3, leaving just the call of the uninstrumented method.

5.3 Information Flow Security

It may seem as if resource managers could infringe information flow security. Is it not possible that sensitive data (e. g. phone numbers from the address book) leaks from a manager while it is passed from method to method? We argue that at least for resource safe MIDlets, this is not the case.

Property 3 *If a MIDlet is resource safe then its resource managers do not leak information.*

This is a corollary of Property 2. If a MIDlet is resource safe, the resource managers can be erased without changing the MIDlet's observable behaviour. Yet, erased resource managers are stateless, so they cannot leak information. Hence, no leakage is observable.

5.4 Secure Deployment

As mentioned in Section 5.1, the security guarantees do not only depend on the correctness of the resource management library itself but also on the MIDlet correctly using the API (i. e. not bypassing or subverting the library).

Property 4 *Correct use of the resource management API can be checked statically by inspecting the MIDlet's JAR only.*

The property holds due to the restrictions imposed by CLDC and MIDP (see Section 2), which imply that all of the MIDlet's classes are statically known (since all classes must be loaded from a single JAR) and the signature of each method call is statically known (since reflection is not supported). Thus, the following properties of the MIDlet's class files can be statically checked.

- The MIDlet does not bypass the instrumentation. More precisely, if the MIDlet allocates a particular resource type (e. g. `MsgResource`) then it does not call uninstrumented methods for consuming resources of that type (e. g. the method `send(Message)` declared in the WMA interface `MessageConnection`).
- The MIDlet does not suppress failing assertions. More precisely, it does not catch `ResManagerError` or any of its superclasses.
- The MIDlet does not pass policies of its own to the `enable` method. More precisely, none of the MIDlet's classes extend the abstract class `Policy`.
- The MIDlet does not subvert the implementation of resource multisets by adding re-

source types of its own.[7] More precisely, none of the MIDlet's classes extend the abstract class `Resource`.

- The MIDlet does not exploit non-public methods (e. g. `decide`) of the resource management library. More precisely, none of the MIDlet's classes are declared to be part of the packages that constitute the resource management library.

The correctness of the resource management library itself cannot be checked easily, hence the library (including the instrumented methods) has to be trusted. Yet, as MIDP does not support the download of trusted libraries, MIDlets using the resource management API have to provide the library as part of their own JAR. To establish trust in the library, a trustworthy third party (e. g. the network operator) should vouch for it by signing the MIDlet. In detail, the deployment process should comprise the following steps.

(i) In the MIDlet's JAR, the signer replaces the untrusted resource management library with its own trusted implementation.

(ii) The signer checks for correct use of the resource management API by checking the above properties.

(iii) The signer signs and deploys the MIDlet (possibly after it passed other checks, too).

The signer may choose to erase resource managers by replacing the resource management library with the library for erased managers (see Section 5.2) if there is additional confidence in the MIDlet's resource safety (where this confidence may have been gained by type checking, extended static checking, interactive verification or extensive testing). Of course, Property 1 is not guaranteed by the library for erased managers.

There is a reason, why MIDlets should be signed by the network operator (or device manufacturer) rather than just by any trusted third party. For otherwise, the MIDP specification (see Section 2) demands that the uninstrumented methods which are called by the instrumented ones do still pop-up authorisation screens, despite the fact that the user (or the policy) has already approved all of the resources held by resource managers.

As an alternative deployment scenario, the resource management library could be integrated into future versions of MIDP. In this case, the MIDP class loader would have to check for correct use of the API, rendering unnecessary the requirement that MIDlets be signed by the network operator.

6 Related Work

Runtime monitoring to increase software reliability is at the heart of the Java language [4] with its mandatory runtime checking of array bounds and null pointer dereferences. Several frameworks have been proposed for enhancing Java with runtime monitoring of resource consumption, for example JRes [3], J-Seal [1] and J-RAF [10]. Real-time Java (RTSJ [5]) provides resource monitoring as part of its support for real-time applications. These frameworks monitor specific resources (CPU, memory, network bandwidth, threads), relying on instrumentation of either the JVM (for CPU time), low level system classes (for memory and network bandwidth) and the bytecode itself (for memory and instruction counting). Where our resource management API is designed to enforce security, these frameworks

[7] The hash table based implementation of multisets may fail to function correctly if resources are added that breach the contract that Java imposes on the `equals` and `hashCode` methods.

were developed to support resource aware applications, which can adapt their behaviour in response to resource fluctuation, for example by trading precision for time (by returning an imprecise result to meet a deadline), or time for memory (by caching less to reduce memory consumption).

Runtime monitoring can be used to check whether a program meets a safety property specified in a propositional temporal logic. Tools like JPaX [9] compile a specification into a finite automaton which runs in parallel with the program, observing its behaviour. This kind of temporal specification can express resource protocols like authorise-before-use but is not expressive enough to capture protocols that involve counting potentially unbounded resources.

Schneider [13] advocates a similar use of (not necessarily finite) automata for enforcing security policies at runtime. [15] extends this by allowing an application to query the policy for compliance with a planned sequence of actions. Thus, the application can react gracefully to the policy's decisions; our resource managers provide a similar policy query feature through the `enable` method.

7 Conclusion

We have designed a Java library for tracking and monitoring the use of external resources on MIDP mobile phones (e. g. sending text messages). The library improves the flexibility of runtime monitoring in MIDP (which previously was in the hands of the user), providing a clear user interface and flexible policies while maintaining the security guarantee that any attempt to abuse resources will be trapped.

Our technical contribution is an API for fine-grained accounting of external resources, where fine-grained accounting is achieved by resource managers tracking not just a fixed set of resources but an input-dependent unbounded set (e. g. phone numbers from the user's address book). The API is extensible, admitting to add new resource types and new policies by extending the class hierarchy. Moreover, we have outlined how a trusted library implementing the API can be deployed to MIDP phones as part of a potentially malicious application in such a way that the application cannot subvert the security guarantee (turning the application into a less malicious one). Finally, resource monitoring can be switched off by "erasing" resource managers, which reduces the overhead without changing the observable behaviour of resource safe applications (and we are working on a type system for certifying resource safety [2, chapter 3.3]).

Acknowledgements
This work was funded in part by the Sixth Framework programme of the European Community under the MOBIUS project FP6-015905. This paper reflects only the authors' views and the European Community is not liable for any use that may be made of the information contained therein. Ian Stark was also supported by an Advanced Research Fellowship from the UK Engineering and Physical Sciences Research Council, EPSRC project GR/R76950/01.

References

[1] Walter Binder, Jarle Hulaas, and Alex Villazón. Portable resource control in Java: The J-SEAL2 approach. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 139–155, 2001.

[2] Mobius Consortium. Deliverable 2.1: Intermediate report on type systems. Available online from `http://mobius.inria.fr`, September 2006.

[3] Grzegorz Czajkowski and Thorsten von Eicken. JRes: A resource accounting interface for Java. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 21–35, 1998.

[4] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, third edition*. The Java Series. Addison-Wesley Publishing Company, 2005.

[5] JSR 1 Expert Group. JSR 1: Real-time specification for Java. Java specification request, Java Community Process, January 2002.

[6] JSR 118 Expert Group. JSR 118: Mobile information device profile 2.0. Java specification request, Java Community Process, November 2002.

[7] JSR 139 Expert Group. JSR 139: Connected limited device configuration 1.1. Java specification request, Java Community Process, March 2003.

[8] JSR 205 Expert Group. JSR 205: Wireless messaging API 2.0. Java specification request, Java Community Process, June 2004.

[9] Klaus Havelund and Grigore Rosu. Monitoring Java programs with Java PathExplorer. *Electr.Notes Theor. Comput. Sci.*, 55(2):200–217, 2001.

[10] Jarle Hulaas and Walter Binder. Program transformations for portable CPU accounting and control in Java. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 169–177, 2004.

[11] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. *JML Reference Manual*, July 2007. In Progress. Available from `http://www.jmlspecs.org`.

[12] Kevin D. Mitnick and William L. Simon. *The Art of Deception*. John Wiley and Sons, Inc., 2002.

[13] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[14] Unknown. Redbrowser.A, February 2006. J2ME trojan. Identified as Redbrowser.A (F-Secure), J2ME/Redbrowser.a (McAfee), Trojan.Redbrowser.A (Symantec), Trojan-SMS.J2ME.Redbrowser.a (Kaspersky Lab).

[15] Dries Vanoverberghe and Frank Piessens. Supporting security monitor-aware development. In *International Workshop on Software Engineering for Secure Systems*. IEEE Computer Society, 2007.