Context-Aware Monitoring of Untrusted Mobile Applications

Andrew Brown and Mark Ryan

School of Computer Science, University of Birmingham, B15 2TT, UK
{A.J.Brown,M.D.Ryan}@cs.bham.ac.uk

Abstract. Current measures to enhance the security of untrusted mobile applications require a user to trust the software vendor. They do not guarantee complete protection against the behaviours that mobile malware commonly exhibits. This paper expands *execution monitoring*, building a more precise system to prevent mobile applications deviating from their intended functions. User judgements about program execution can be specified abstractly and compiled into a monitor capable of identifying an event's context. We demonstrate our development of a prototype system for the BlackBerry platform and show how it can defend the device against unseen malware more effectively than existing security tools.

1 Introduction

The number of mobile devices in operation today exceeds the size of the population in over 30 countries. Of these, roughly 30% are "smart" devices that allow the user to download, install and execute third-party applications. Such an expansion in device capabilities has made them more vulnerable to attack. This paper focuses on the prevention of *unseen mobile malware*, distributed by a malicious third-party in order to compromise the confidentiality, integrity and availability of a user's interaction with mobile data and services.

1.1 Mobile Malware Defence

Java Micro Edition (Java ME) is the most popular framework for executing third-party mobile applications. It consists of two components: the *Connection Limited Devices Configuration* (CLDC) [11], which facilitates program access to the native methods which control device hardware, and the *Mobile Information Device Profile* (MIDP) [12], which provides programmers with abstractions of generic device functionalities (e.g., sending SMS and e-mail messages). Java ME applications are called MIDlets.

Java ME's security model is not flexible enough to allow guarantees to be made about all security-relevant events an application might perform. In most cases, it only takes into account the application provider (i.e., the signatory of the MIDlet). Java Security Domains [14] are implemented in some architectures to block security-relevant API calls, though this can also prevent an application being used for its primary purpose.

Application-level virtualisation allows users to run additional security tools to verify the identity and integrity of the downloaded application. *Code signing*, *discretionary access controls* and *signature-based anti-virus software* are the most commonly used approaches. Each has the following associated problems:

- *Digitally signing* a hash of an executable can confirm its author and guarantee that it has not been altered since it was signed, but does not guarantee the quality or security of code the application will execute.
- *Access controls* contribute to a systematic security framework[1] but can be inflexible, with default settings leaving the device vulnerable to attack and stricter ones impeding program functionality.
- *Mobile anti-virus* software requires signature dictionaries to be downloaded and updated, which is infeasible for devices with low power and/or limited network connectivity. Further, attack recovery is rather rigid; it simply deletes executable files.

1.2 Execution Monitoring

We employ a well-known technique called *execution monitoring* [1,3,5,7,8,9,16,17,18] to defend devices against malicious and defective software. An execution monitor is a co-routine that runs in parallel with a target program, fully regulating its interaction with its host machine. The approach can prevent and recover from harmful behaviour in real-time, rather than after an attack has succeeded, with minimal disturbance to legitimate program features. Execution monitors analyse either system calls or API calls, providing their host with a more *fine-grained* view of target application behaviour.

Despite these advantages, execution monitoring has not been widely adopted by developers building anti-malware products. The following reasons explain this:

Specifying an execution monitor is a complex process. Monitor implementation requires sequences of triggering and recovery events to be defined. High-level policy languages exist to counteract this problem, but most require reasoning about misuse event sequences in an imperative fashion and so are difficult for non-technical users comprehend.

Execution monitors cannot precisely capture malicious behaviour patterns. Identification of the *context* in which a target program invokes an event is required to achieve this. Existing monitoring schemes, including those for Java ME [5], cannot store sufficient information about an event's properties and so restrict application behaviour beyond the set of security-relevant events.

[1] Access controls can prevent an unsigned application being linked to the CLDC APIs which control native device functions. This type of mitigation is often still too coarse-grained.

1.3 Paper Structure

Section 2 presents our approach to allow more fine-grained control of mobile applications. We develop declarative language called Application Behaviour Modelling Language (ABML) to express policies that identify the context of a program event. In Section 3, we use our operational experiences in mitigating BlackBerry intrusions to demonstrate our work and provide an empirical evaluation. Section 4 details our enforcement mechanism for ABML, which works by translating a policy into Java source code, for input to a policy enforcement engine called Polymer [3]. We conclude by relating our contributions to those of others and proposing future work.

2 Modelling Application Behaviour

Mobile malware often uses HTTP, HTTPS, or other popular protocols to compose an attack. By employing seemingly 'legitimate' combinations of events and protocols to transmit sensitive information, unseen malware can avoid detection. Context-aware monitoring of event invocations can mitigate such attacks and preserve program functionality.

2.1 Approach Summary

Our approach places application monitoring wrappers around arbitrarily downloaded MIDlets in order to implement a security policy and so constrain undesirable functionality. We use these to apply user-specified policies in an aspect-oriented manner, creating flexible security against malicious components. Our policies are generic first-class objects and can be applied to any MIDlet as follows:

1. At download-time, a MIDlet's bytecode is scanned to identify all API calls it makes. Only application-relevant calls can be contained in a policy.
2. At specification-time, every high-level policy is compiled into monitor source code, which is contained within a Java class.
3. At load-time, every API call in the libraries a MIDlet can access is instrumented with calls to the monitor.
4. At runtime, control is passed to the monitor when any event which the policy reasons about is invoked.

Figure 1 illustrates the technologies we employ to implement our framework. Our approach uses Polymer[2] [3], a fully-implemented engine for enforcing execution monitoring threads on arbitrary Java programs. It responds to a policy violation by transforming a target to adhere to conditions of the applied policy.

[2] We have modified Polymer to allow it to operate on MIDlets, using MIDP2 and CLDC libraries [11,12].

Our system architecture targets two classes of user separately:

– *Administrators* can construct and distribute groups of policies to the devices which they manage, specifying them textually or by means of a comprehensive user interface.
– *Users* can manage and modify policies set by an administrator[3], or select a pre-specified policy from a repository, based on the class of program they perceive to have downloaded. Our MIDlet policy classes are *editor*, a *browser*, a *shell*, a *viewer*, a *transformer*, a *game*, and a *messenger*.

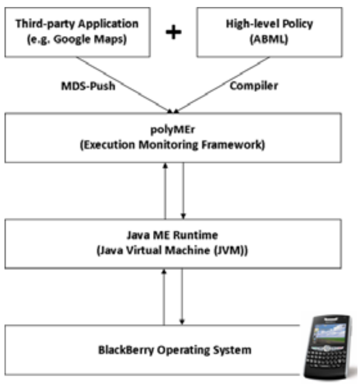

Third-party Application (e.g. Google Maps)
+
High-level Policy (ABML)
MDS-Push
Compiler
polyMEr (Execution Monitoring Framework)
Java ME Runtime (Java Virtual Machine (JVM))
BlackBerry Operating System

Fig. 1. Integration of Polymer and Java ME on the BlackBerry

2.2 Application Behaviour Modelling Language (ABML)

ABML is a policy specification language for reasoning about application behaviour in a declarative manner using arbitrary events. It is inspired by Behaviour Modelling Specification Language (BMSL) [17], in which event-based security-relevant properties can be expressed in order to capture the *intended* behaviour of a system, or *misuse* behaviours associated with vulnerabilities and their exploitation. ABML builds on BMSL with:

– High-level constructs for specifying event *histories* and *recovery sequences*;
– The notion and *strongly*- and *weakly*-ordered events;
– *Abstraction mechanisms* for referring to events *and* the data they use;
– Operators to increase the power of comparison between event conditions.

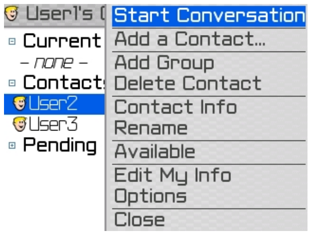
ABML's primary constructs are:

1. **Policies:** A policy consists of a set of rules, or a set of variables followed by a set of rules. Variable declaration may be *local* or *global*: a local variable's scope is limited to a single rule, whereas a global variable can be applied within all rules in a policy.
2. **Rules:** A rule is of the form $\mathsf{H} \rightarrow \mathsf{R}$, where H denotes a *history of actions* invoked by the target application which triggered a rule, and R denotes some postcondition which (i) either must hold true for that history, or (ii) must specify some responsive steps to be followed where a rule is triggered.
3. **History patterns:** A history, H, models the target application's instruction stream and is expressed by a pattern over a sequence of events: pat. A pat can measure event *occurrence*, *non-occurrence*, *sequencing*, *alternation* and *repetition*, providing a means of specifying the temporal properties of application behaviour [17].

[3] Where the administrator has granted the user sufficient access permissions.

   (a) **Strongly-ordered patterns:** A strongly-ordered pat considers events strictly in the order they occur in. It is specified by separating events with a semi-colon (';').
   (b) **Weakly-ordered patterns:** A weakly-ordered pat allows events which a policy does not reason about to occur in between those events which it does reason about. This representation is suitable where events may happen simultaneously, or housekeeping operations are to be performed between events. It is specified by separating events with a double semi-colon (';;').
4. **Events:** The following may be referred to within a rule:
   (a) **Atomic events** have the form $\mathsf{e}(\mathsf{value}_1, \mathsf{value}_2, ..., \mathsf{value}_n)|\mathsf{cond}$, where e is an API call name, value is an argument to that call and cond is a boolean-valued expression on that call's arguments. We add comparison operators such as contains, startsWith and endsWith to enable more advanced reasoning about event conditions.
   (b) **Abstract events**, E, provide a means of concatenating atomic events to form suitable abstractions that reflect a host's ontology (e.g., *read from a local file*, *send an SMS message over carrier network*, *read data from device's GPS module*). An E is referenced using the constructs zone, resource and action, which allow one to refer to the zone a program may interact with (e.g., internet, network, localdevice), the resources within *that* zone it can access and the actions it may perform on *those* resources.
   (c) **Recovery events** allow users to reason about policy violations without having in-depth knowledge of the target application. A postcondition, R, assists this reasoning process. It allows the policy author to:
      – Deny the invocation of a triggering event ($\perp$);
      – Enforce a condition on a triggering event (cond):
         – If cond is *true* when a rule is evaluated, program execution is permitted to continue;
         – Otherwise, a sequence of recovery events of type E are derived from cond at compile-time.
      – Refer to an alternative event, which may be abstract (E);
      – Make a call to terminate the target application (`halt`).
5. **Event data usage:** An event's context is normally specified at the atomic level by populating the arguments to that event (e.g., setting one or more values in $\mathsf{e}(\mathsf{value}_1, \mathsf{value}_2, ..., \mathsf{value}_n)$). Because abstract events are collections of atoms, each may have a different argument signature. ABML therefore provides an event data tracking construct, which allows some abstract data object, O, to be instantiated at runtime with the concrete data used by that event. For example, HttpResponse represents a response over the HTTP protocol. It contains instance variables to capture that response's `head` (e.g., status code, date, server, content-type), `body` (e.g., (X)HTML, XML) and a reference to the HttpRequest which invoked it. Section 4.2 illustrates this process' enforcement.

3 Demonstration



In order to demonstrate ABML policy enforcement, we present our operational experiences in monitoring the execution of an untrusted MIDlets on the BlackBerry 8800 series device [10]. The MIDlet we consider is an instant messaging application, which was downloaded to the device over GPRS at the user's request. The user wishes to control the MIDlet's features with the device's input apparatus; initiate 'live' messaging sessions with remote devices in a contacts list; send asynchronous messages (e.g., SMS, e-mail) to a contact; and, receive software updates from a remote server.

3.1 Existing Device Protection

The MIDlet may invoke other behaviours which the user is unaware of, some of which may occur despite the BlackBerry's code signing or access control mechanisms. It must be linked to the CLDC libraries in order to execute and so must be signed. The user can set the device's access controls, as illustrated in Table 1, to prevent the signed code behaving maliciously.

Table 1. Settings for device access controls to prevent a MIDlet forwarding data

Attack on...	**Application Permission**	**Set to...**
1 – SMS	Connections > Carrier Internet	Deny
2 – E-mail	User data > E-mail	Deny
3 – TCP/IP	Connections > Carrier Internet	Deny
4 – Bluetooth	Connections > Bluetooth	Deny

Such measures clearly inhibit the primary functions of a messenger. For example, the only means the user has to prevent the MIDlet intercepting and forwarding SMS messages to an attacker is to deny its use of the BlackBerry Internet Service. This renders it unusable for the purpose for which the user downloaded it.

3.2 Attacks and Countermeasures

Users can apply an ABML policy which predicts and defends against attacks in a more fine-grained way. Such a policy would be crafted by an administrator or device vendor and applied by the user when they first download the MIDlet (§2.1). The ABML rules we outline below are contained in our *messenger* policy, from which a Java `Policy` is computed that generates an execution monitor. Both the MIDlet and its monitor execute on the device as a separate threads inside the Polymer framework, so the device's application permissions for Polymer (for *connections* and *user data*) must all be set to "allow".

[1] Bluetooth backdoor attack: MIDlets can transmit data to and from the device via its Bluetooth serial port. They must be signed to establish a Bluetooth connection, but can use an open serial port without being signed. Sensitive data (e.g., e-mail, SMS messages, Personal Information Manager (PIM) contacts) can be captured and transmitted to other in-range devices, some of which may be attacker-controlled. To prevent this, the following rule should be applied: *"the target may send data to the Bluetooth serial port only if it has established a client-side Bluetooth connection with a paired device"*.

Policy 1. Policy to prevent a MIDlet using a Bluetooth backdoor

ABML rule	Connection b, c: device.data.Send(Stream s, c)|c.address startsWith "btspp://" → device.service.Connect(b)|b.address := c.address:
Rule format	var_1: var_2: $\mathsf{H}\{\mathsf{E}_2(\mathsf{O}_2)\mid\mathsf{cond}_{(2)}\} \rightarrow \mathsf{E}_1(\mathsf{O}_1)\mid\mathsf{cond}_{(1,2)}$:
Triggering event	device.data.Send(Stream s, Connnection c)

Monitored calls	`javax.microedition.io.Connector.<init>`
	`net.rim.device.api.bluetooth.BluetoothSerialPort.<init>`

A significant level of user interaction is required to establish such a connection, so users are unlikely to be coerced into doing so easily. The abstract event localdevice.data.Send(Stream s, Connnection c) triggers Policy 1. It relates to the low-level actions a MIDlet may use to stream data to some connection and is conditional on that connection being to a Bluetooth serial port. If the policy's conclusion evaluates to `true`, the monitor has recorded that a legal client-side Bluetooth connection has been established with a paired device and the Send event's address parameter matches that of the established connection – invocation is allowed. Otherwise, Send is suppressed and the MIDlet's execution continues without that data being transmitted.

[2] SMS interception attack: Sending and receiving an SMS message uses the MIDP 2.0 standard [12] and so does not require an application to be signed. Once the BlackBerry user has agreed to its standard prompt *"Allow Network Access?"*, they do not receive further warnings, even for subsequent executions. An untrusted MIDlet can create an 'SMS channel', which remains in place where the attacker has programmed it to run as a background process on receiving an *exit event*. We therefore add the rule: *"the target may send an SMS message only if the data that message contains was entered manually by the user and the user invoked the sending of that SMS by pressing the device's trackwheel"*.

If the application attempts to send an SMS message containing data that was not user-entered, invocation of internet.data.Send(SMSMessage s) is suppressed. Policy 2 uses the construct $\mathsf{H} \rightarrow \mathsf{R}$, where R refers to an abstract event sequence that must have already occurred (and any conditions on its occurrence satisfied) for H to be invoked. If this is not the case, *any* event which relates to sending an SMS message to the internet domain is prevented.

Our extensions to the Polymer engine determine the context of the triggering event by monitoring the MIDlet getting data from a text field after the user has

Policy 2. Policy to defend the BlackBerry against SMS interception

ABML rule	SMSMessage s: Text r, t: internet.data.Send(s) $\rightarrow$ device.keypad.Press(r);; device.trackwheel.Click(); device.gui.GetText(t)|t := s.body $\wedge$ t contains r:
Rule format	var_1: var_2: var_3: $H\{E_4(O_4)\} \rightarrow E_1(O_1);;\ E_2(..);\ E_3(O_3)\|\mathsf{cond}_{(3,1,2)}$:
Triggering event	internet.data.Send(SMSMessage s)

Monitored calls	`net.rim.blackberry.api.sms.OutboundMessageListener.notifyOutgoingMsg(Msg m)`
	`net.rim.device.api.system.EventInjector.KeypadEvent(..)`
	`net.rim.device.api.system.EventInjector.TrackwheelEvent(THUMB_CLICK, .., ..)`
	`net.rim.device.api.ui.component.TextField.getText(..)`

manually entered it and pressed an interface button to send an SMS message containing that text. Our compiler creates empty objects (O of type `Text`) that refer to all *live instances* of `Text` pertaining to the user's interaction. Further details on this process are provided in Section 4.2.

This policy uses a *strong* temporal order between E_2 and E_3. If events which this sequence does not reason about occur in between these, the policy is violated and its triggering event supressed. The method call `TextField.getText(..)` is tied as closely as possible to a `TrackwheelEvent` that uses a 'thumb click'. More complex reasoning here would require the policy to model the MIDlet's user interface, which is not possible for *unseen malware.*

[3] HTTP proxy attack: Unsigned MIDlets can create TCP connections on the BlackBerry, again prompting the user with the BlackBerry's *"Allow Network Access?"* dialog only once. Attack code can then use the device as a proxy for traffic; the attacker often having the intention of accessing illicit material or performing denial of service attacks.

In order to mitigate this, we construct a policy to analyse the messenger's use of the device's Internet connection. This states: *"the target may send an HTTP response* (r) *to the network only if it has not previously received an HTTP request* (n) *whose host's name matches that of* r*'s host; where* r *was proceeded by the sending of an HTTP request* (p) *and the receipt of an HTTP response* (q) *whose hosts match".*

Policy 3. Policy to prevent a MIDlet using the BlackBerry as an HTTP proxy

ABML rule	HttpResponse r, q: HttpRequest n, p: internet.data.Send(r) $\rightarrow$ internet.data.Receive(n)|n.host $\neg$ = r.host;; internet.data.Send(p);; internet.data.Receive(q)|p.host := q.host:
Rule format	var_1: var_2: var_3: var_4: $H\{E_4(O_4)\} \rightarrow E_1(O_1)\|\mathsf{cond}_{(3,1)};;\ E_2(O_2);;\ E_3(O_3)\|\mathsf{cond}_{(4,2)}$:
Triggering event	internet.data.Send(HttpResponse r)

Monitored calls	`javax.microedition.io.HttpConnection.openInputStream(..)`
	`javax.microedition.io.HttpConnection.openDataInputStream(..)`
	`javax.microedition.io.OutputStream.openOutputStream(..)`
	`javax.microedition.io.DataOutputStream.openDataOutputStream(..)`

The monitor constructed here analyses request calls to `HttpConnection` and ascertains whether a related HTTP response is ever sent to the network. Our enforcement model constructs `HttpRequest` and `HttpResponse` objects whenever abstract events whose arguments match these are triggered (§4.2). These encapsulate any data being sent or received over HTTP by the messenger application.

4 Policy Compilation and Enforcement

Our compiler translates ABML policies into Java classes (of type `Policy`). Translation from an ABML policy to a `Policy` class is performed in a compositional manner, so incremental changes can be handled with ease. Users may wish to weaken a policy where it denies some program behaviour they wish to permit, or strengthen it where they learn of some vulnerability in the target MIDlet.

4.1 Synthesising Monitors from ABML Specifications

Each `Policy` output by our compiler contains a method called `query()`, in which every rule is implemented by enumerating `case`s in a `switch()` statement. A policy is triggered where an event's signature is matched by precisely one `case`. The method then returns a `Suggestion` object which indicates how that event should be dealt with. At the highest level, our compiler's algorithms operate as follows to compose a `Policy`.

1. $\forall$ E, select the pre-compiled `AbstractAction` class which E refers to. Insert appropriate `import` statements into `Policy` to provide access to these classes.
   (a) $\forall$ O in E, create an empty class of type `AbstractData` with instance variables to reflect O's parameters. Insert appropriate `import` statements.
      i. Provide appropriate accessor and mutator methods to access O's parameters.
2. Populate `query()`'s `switch()` statement with one `case` per E. Set that `case`'s trigger to the `AbstractAction` selected in (1).
   (a) If E is non-atomic, prepend each `case`'s trigger with the keyword "abs"[4].
3. $\forall$ cond in E, declare one boolean variable, $\mathtt{b}_n$, setting its value to `false`.
4. $\forall$ E, such that E's arguments contain O, insert a reference to that O inside that `case`'s trigger[5].
5. $\forall$ E, construct one conditional statement for each *related* $\mathtt{b}_n$ in that E's `case`. Populate it with an expression to represent E's cond[6].
   (a) $\forall$ E such that cond evaluates to `true`, set *related* $\mathtt{b}_n$ = `true`.
6. Compute reasoning constructs to recover from a rule violation. $\forall$ rule:
   (a) $\forall$ cond in a rule's *premise*:
      i. If cond = `true`, return an `OKSug` (e.g., that event can be executed);
      ii. Else if cond = `false`, this event is irrelevant: return an `IrrSug`.

[4] E.g., internet.data.Send(..) $\rightarrow$ `case <abs void internet.data.Send(..)>:`
[5] E.g., internet.data.Send(Request r) $\rightarrow$ `case <abs void internet.data.Send(r)>:`
[6] E.g., `head.startsWith(‘‘http://’’)`

(b) ∀ cond in a rule's *conclusion*:
   i. If R = ⊥, return `Suggestion` to halt the MIDlet (`HaltSug`);
   ii. Else if R = cond, and cond is `false`, halt the MIDlet;
   iii. Else if R = E:
      – If $\mathtt{b}_n$ = `false`, this event is irrelevant: return an `IrrSug`;
      – Else return `ExnSug` to suppress E but continue execution.

Rule precedence is determined by the order rules are listed in a policy. A monitor queries its first rule implementation and always follows the `Suggestion` of that `rule` if it considers the trigger action to be security-relevant (e.g., an `ExnSug` or a `HaltSug` is returned, but not an `IrrSug` or an `OKSug`). Otherwise, the `Suggestion` of the second `rule` is followed, and so on. Authors whom edit policies in a textual manner should therefore take care, though we presume most will modify policies using our specification GUI.

4.2 ABML Policy Enforcement

Figure 2 illustrates our monitoring model and depicts a target application and an execution monitor in execution on the JVM. The three grey-shaded elements represent the classical execution monitoring process, as was first described by Schneider [16], and as is implemented by Polymer [3,2]. All events invoked by the target program are analysed by the execution monitor: events which the policy does not reason about are executed without analysis and those which it reasons about are apprehended if any conditions on them evaluate to `false`.

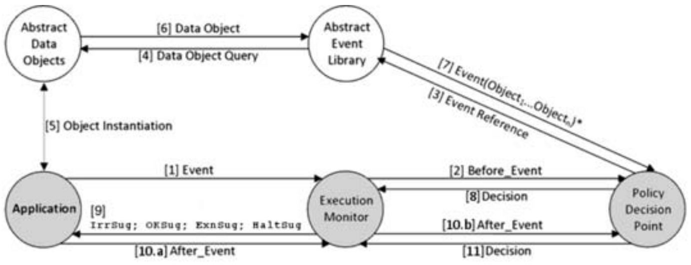


Fig. 2. Operation of our ABML policy enforcement mechanism

Our extensions to this approach (white-shaded elements) "investigate" the context in which an event occurs to achieve more precise separation between *legitimate* and *misuse* behaviours. This process works as follows:

– Empty objects, derived from any O in a rule at compile-time, are created. They extend `AbstractData` and contain the necessary instance variables to represent O. This mapping is pre-defined in an XML file.
– Accessors and mutators for each O are set by our compiler and are mapped to any ABML event (E) which refers to it.

- Each `AbstractData` object is instantiated when a MIDlet performs a related event which a `Policy` reasons about (cf. step 5 in Figure 2):
  - A new class is created, whose unique reference identifies the data used by that call;
  - Garbage-collected instances of this class are checked for in `classMap`, which is a hash map that collects class names and their references to other classes;
  - If an object has not been garbage-collected then it is a live instance. Its reference is set to its associated instance variable in `Policy`.
- A `Policy` is able to query that object using the accessors and mutators it provides.

Preventing calls which bypass APIs. Malicious programs may try to load native code in order to mirror the functions of an API and so effectively bypass it. Our approach can avoid such an attack. MIDlet bytecode is "pre-verified" by the CLDC Class Verifier, which acts to sandbox an application by inserting certain security checks as class file annotations to be carried out when the device loads the class. Changes to bytecode after this has occurred are detected at runtime by the JVM, which will reject the class [6]. Furthermore, our system sets a Java `SecurityManager` policy which itself does not set a `checkLink` permission for any native library. Attempts at executing the `load()` or `loadLibrary()` methods therefore result in a `SecurityException`.

4.3 Performance Analysis

Expressing policies: We have tested our language and enforcement engine using the attacks presented (§3.2). In order to mitigate these, our compiler generated 356 lines policy code, split between three `Policy` classes and three `Combinator` classes. The latter enforce rule precedence. An equivalent ABML requires only 15 lines of code: 5 lines of global variable declarations and 10 lines of ABML rule specifications (an arithmetic mean of 5 lines per attack). The policies ABML can express are more readable than their imperative counterparts.

Mitigating information flow: Our model enables a class of policies which prevent malicious programs from leaking sensitive information. We ensure that if an information flow constraint cannot be mitigated, the events which cause it are denied. Execution monitoring cannot capture all cases of information flow [16]; full information flow analysis is required to achieve this [13]. The significant difference between our language and others when considering information flow is its ability to specify live instances of data as arguments to events.

Execution-time overhead: As mobile devices have limited computing power, we examine the costs of enforcing policies using our approach at:

- **Download-time:** Scanning a MIDlet's bytecode for API calls is dependant on the number of calls it makes. Our test code contained 83 such calls, which our scanner took 4.4s to extract and add to an event definition file.

- **Specification-time:** Compilation of a `policy` is also dependant on its complexity. Compiling our rules to prevent attacks 1 through 3 (§3.2) took 1.2s.
- **Load-time:** The total time to instrument every method in the Java ME and BlackBerry APIs (i.e., the 7352 methods in the 933 classes in the `javax` and `net.rim` packages of the BlackBerry API v.4.5.0) was 185s, or an average 25ms per instrumented method. This cost is reasonable because library instrumentation need only be performed once per download.
- **Runtime:** The cost of transferring control to and from a policy while executing a target is very low (approximately 1.44 ms per policy decision point). Therein, the run-time overhead of monitoring is is dependent on the complexity of the security policy.

5 Related Work

The foundations of execution monitoring are described by Schneider in [16], where a formal treatment is given to techniques which analyse the actions of a target program and terminate it where it violates a policy. Ligatti [1, 4] have extended this model to suppress program actions, allowing program execution to continue after a policy is violated. This class of monitor is modelled by the *edit automaton*, which is implemented by Polymer [2, 3]. Its policies are separated from the target program and are therefore easy to maintain, re-use, or compose into a hierarchy. Furthermore, it allows only *sound* execution monitors to be computed. A weakness of Polymer is its input language (a constrained Java syntax), which has high expressive power but is difficult for end-users to write policies in.

Much work has been undertaken in developing languages for specifying process behaviours that signify an intrusion. Sekar et al. devised Behaviour Modelling Specification Language (BMSL) [17, 18], a more user-friendly policy language for reasoning about a program's execution through combining system call abstractions. BMSL can express policies that capture the values of arguments to events, but a policy author must forecast the strong temporal order of trigger and recovery events: an infeasible expectation of most.

Later work on policy enforcement seems not to have focussed much on enhancing its precision. Castrucci et al. [5] describe how to monitor MIDlets, although their policy language requires imperative reasoning using atomic method call signatures and cannot precisely capture the temporal order or the context of event invocations. Those which have attempted this [9] have done so in a low-level manner, focussing on system calls on traditional hosts. It is often difficult, or even impossible, to attribute a system call sequence to a particular program event [15].

6 Conclusion

We have designed a user-operable sandboxing technique to control the execution of untrusted mobile software. It consists of an abstract policy language and an enforcement model based on Polymer. Our language is more expressive than other policy languages, but uses purely declarative constructs. Its policies require lower development effort and can be composed and applied incrementally.

ABML's main contribution is its association of atomic events to program data and its mapping of this process to suitable abstractions, which helps to increase accuracy and reduce vulnerabilities and redundancies. We have provided an enforcement model that is more powerful than existing execution monitors – it is aware of the context in which an event has occurred and so broadens the functionalities of the target application. Our future work will prove that the policies we compute are at least as strong as the ABML policies which specified them when enforced by our model and that their translation into a `Policy` class is sound. This will enable users to place guarantees in our work.

References

1. Bauer, L., Ligatti, J., Walker, D.: More enforceable security policies. In: FLoC 2002: Proceedings of the 2002 Workshop on Foundations of Computer Security, pp. 95–104 (2002)
2. Bauer, L., Ligatti, J., Walker, D.: A language and system for composing security policies. Technical Report TR-699-04, Princeton University (2004)
3. Bauer, L., Ligatti, J., Walker, D.: Composing security policies with Polymer. In: PLDI 2005: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp. 305–314. ACM Press, New York (2005)
4. Bauer, L., Ligatti, J., Walker, D.: Edit automata: Enforcement mechanisms for run-time security policies. International Journal of Information Security 4(1-2), 2–16 (2005)
5. Castrucci, A., Martinelli, F., Mori, P., Roperti, F.: Enhancing Java ME Security Support with Resource Usage Monitoring. In: Chen, L., Ryan, M.D., Wang, G. (eds.) ICICS 2008. LNCS, vol. 5308, pp. 256–266. Springer, Heidelberg (2008)
6. Java Community Process: CLDC 1.1 Specification, `http://jcp.org/aboutJava/communityprocess/final/jsr139/`
7. Erlingsson, U., Schneider, F.B.: IRM enforcement of Java stack inspection. In: SP 2000: Proceedings of the 2000 IEEE Symposium on Security and Privacy, pp. 246–259. IEEE Computer Society Press, Washington (2000)
8. Evans, D., Twyman, A.: Flexible policy-directed code safety. In: SP 1999: Proceedings of the 1999 IEEE Symposium on Security and Privacy, pp. 32–45. IEEE Computer Society Press, Washington (1999)
9. Giffin, K., Jha, S., Miller, B.: Efficient context-sensitive intrusion detection. In: NDSS 2004: Proceedings of the 11th Annual Network and Distributed Systems Security Symposium. Internet Society Press, Reston (2004)
10. Research in Motion: Blackberry Simulators, `http://na.blackberry.com/eng/developers/downloads/simulators.jsp`
11. Sun Microsystems: Connected Limited Device Configuration (CLDC), `http://java.sun.com/products/cldc/`
12. Sun Microsystems: Mobile Information Device Profile (MIDP), `http://java.sun.com/products/midp/`
13. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Transactions on Software Engineering Methodology 9(4), 410–442 (2000)
14. Forum Nokia: Java Security Domains, `http://wiki.forum.nokia.com/index.php/JavaSecurityDomains/`

15. Provos, N.: Improving host security with system call policies. In: Proceedings of 12th USENIX Security Symposium, pp. 128–146. USENIX Press, Washington (2003)
16. Schneider, F.B.: Enforceable security policies. ACM Transactions on Information Systems Security 3(1), 30–50 (2000)
17. Sekar, R., Uppuluri, P.: Synthesizing fast intrusion prevention/detection systems from high-level specifications. In: SSYM 1999: Proceedings of the 8th USENIX Security Symposium. USENIX Association, Berkeley (1999)
18. Sekar, R., Venkatakrishnan, V.N., Ram, P.: Empowering mobile code using expressive security policies. In: NSPW 2002: Proceedings of the 10th New Security Paradigms Workshop, pp. 61–68. ACM Press, New York (2002)