A Location-based Policy-specification Language for Mobile Devices

Joshua Finnis, Nalin Saigal, Adriana Iamnitchi, Jay Ligatti*

Department of Computer Science and Engineering
University of South Florida
4202 E. Fowler Ave., ENB 118
Tampa, FL 33620

Abstract

The dramatic rise in mobile applications has greatly increased threats to the security and privacy of users. Security mechanisms on mobile devices are currently limited, so users need more expressive ways to ensure that downloaded mobile applications do not act maliciously. Policy-specification languages were created for this purpose; they allow the enforcement of user-defined policies on third-party applications. We have implemented LoPSiL, a location-based policy-specification language for mobile devices. This article describes LoPSiL's design and implementation, several example policies, and experiments that demonstrate LoPSiL's viability for enforcing policies on mobile devices.

Keywords: Policy-specification languages, location-dependent policies, mobile devices, security and privacy.

*Corresponding author. Tel: +1-813-974-0908; fax: +1-813-974-5456
Email addresses: `jfinnis@mail.usf.edu` (Joshua Finnis), `nsaigal@cse.usf.edu` (Nalin Saigal), `anda@cse.usf.edu` (Adriana Iamnitchi), `ligatti@cse.usf.edu` (Jay Ligatti)

1. Introduction

The widespread adoption of mobile devices and their rich computational and communication capabilities has led to a plethora of applications for mobile platforms. Apple's App Store for the iPhone is the most prominent example of this, containing over 130,000 applications [1] only a year and a half after launch, with a total of over 3 billion mobile applications downloaded [2]. The Android Market, which was developed by Google for their Android mobile operating system, lists over 30,000 applications as of March 2010 [3]. It is projected that 8 billion mobile application downloads will occur during 2010 across all mobile platforms [4].

The producers of such applications range from large companies to individual hobbyists who mostly provide no guarantee and no accountability for the quality and security of their product. In particular, mobile devices have unprecedented access to private, personal information through the use of location services such as GPS. The security infrastructure of mobile devices such as roaming laptops, cell phones, and PDAs, does not include sufficient control over how location information is accessed by applications. On current mobile application platforms, users have no more information than a description of the application from the developers and comments from other users, which can easily be gamed. Many users simply do not think of their mobile device as a PC—despite the current generation mobile chipsets being as powerful as PC chipsets of only 10 years ago [5]—and do not realize that they need to protect their mobile devices against applications just as they do PCs.

We identify two security risks that may occur from running untrusted mobile applications on mobile devices:

1. Developers may insert code into an application a user would have no reason to question. Such an application could, for example, access personal information

from the address book of a smartphone, retrieve photos taken by a camera, take new photos, or access the user's location using the GPS API. The user's location and other personal information could then be sent to an unknown server through a Wi-Fi or 3G connection, all without the user's knowledge. In fact, a 2010 study of 30 popular third-party Android applications found that half shared location data with advertisement servers without user consent [6].

2. Despite the best intentions of developers, bugs within an application can be exploited. In a large application, dozens of security checks might exist. Still, a single bug might be enough to negate all the security checks. Extra care must be taken to ensure mobile devices are free from such bugs, as mobile users are less aware that their mobile devices can be subject to the same problems they are used to computers having.

Some security measures in mobile operating systems attempt to guard against these threats. For the first problem presented above, some mobile operating systems have user alerts. For example, the iPhone alerts the user the first time an application requests location data; subsequent requests can go unnoticed. The Android operating system requires applications that use hardware capabilities like the GPS, camera, and Internet access, to list all such capabilities up front. The user is able to see the capabilities the application requested at install time and decide whether to allow it or not. However, the average user is unaware of the complete functionality of the application: the application may perform the tasks it says it will (e.g., updating a Facebook page) while performing malicious actions in the background (e.g., sending the user's information to advertisers.

For the second problem presented above, protecting against application bugs that can be exploited for malicious attacks, Android runs each application in its own virtual machine and requires permissions for one application to access another's storage space. Thus, if one running application is compromised, it will not be able to infect other running processes or their files. The compromised application will, however, still have access to previously granted permissions, such as GPS, camera, and Internet. There is no existing, transparent mechanism in place to allow the user to discover if an application has unwanted behavior.

These problems can be mitigated by the use of policy-specification languages. Policy-specification languages are intended to simplify the task of specifying and enforcing security policies on untrusted (i.e., potentially insecure) software. A rich variety of expressive, Turing-complete policy-specification languages and systems have been implemented [7, 8, 9, 10, 11, 12]. All of these languages enable users to centrally specify security and privacy policies to be enforced on untrusted software at runtime. All of the cited languages have also been implemented as compilers that convert untrusted code into trustworthy applications. They take as input the application program and a policy and output a new application program equivalent to the one input, except that the output program contains inlined enforcement code to ensure that it satisfies the specified policy at runtime.

As far as we are aware, no expressive policy-specification languages have yet targeted location-dependent policies for securing applications on mobile platforms. Previously developed declarative (and Turing-incomplete) policy-specification languages [13, 14, 15, 16] are limited to location-based access-control policies. Access-control policies are a proper subset of the policies enforceable with expressive enforcement systems, which can specify security constraints even outside the context of accessing resources. Expressive enforcement systems can enforce non-access-control policies such as that a navigational aid needs to be displayed whenever a mobile device deviates from its expected path of travel, or that a user's friends need to be notified whenever the user's device enters a new geographic region.

This paper extends our work on LoPSiL [17], an expressive location-based policy-specification language, by proving its viability through implementation and experimentation on an Android mobile device. LoPSiL provides abstractions for conveniently accessing and manipulating location information in policy specifications. Android was chosen because of its recent popularity, its open source platform based on the Linux operating systems, and its support for Java. Still a somewhat nascent operating system, Android is expected to have the second-highest market share by 2012 [18], and the number of mobile applications for Android is expected to grow to 150,000 by the end of the 2010 [19].

LoPSiL addresses the problems listed above as follows:

1. In contrast to Android's coarse-grained permissions, which either allow or disallow all requests to a piece of hardware, a LoPSiL policy provides fine-grained access over time and resources. For example, a LoPSiL policy can enforce that locations will not be retrieved by an application while the user is in a certain region. Another LoPSiL policy might enforce that outgoing network messages will only be sent if the recipient is on a list of trusted servers, while messages to other recipients will be dropped. This policy would allow, for example, an untrusted application to contact Facebook's servers to retrieve information, but prohibit it from contacting an advertiser's servers to deliver the personal information.
2. Developers can use LoPSiL policies as a centralized security module, rather than scattering security checks throughout the application. Separating the security policy from the core application code provides application developers all the standard software-engineering benefits one would expect from modularization: it

makes the centralized policy easier to create, locate, analyze, and maintain. Centralizing all the security checks to a single module prevents small updates to unrelated application code from causing security holes.

These policies can be written by tech-savvy users in the LoPSiL programming language and stored in a shared policy repository. If a shared LoPSiL policy repository exists, then all users, even those who are less tech savvy, will be free to select (ideally via a user-friendly GUI) the specific policies to enforce on their devices. One example policy might prevent a device's GPS-location data from being sent over the network outside of work hours; a user could install such a policy as a safeguard against an employer-mandated supervisorial application. This example policy could also be extended to disallow any personal information from being sent to unknown (e.g., advertiser's) servers. While we do not explicitly address the aspects of usability and policy adoption in this paper, they are important directions for future work.

Section 2 describes the design of LoPSiL, with its core constructs for simplifying the specification of location-dependent policies in Section 2.1 and several examples highlighting its ease of use in Section 2.2. Section 3 discusses the technical aspects of LoPSiL, including the implementation of a LoPSiL compiler in Section 3.1 and the port to Android in Section 3.2. Experimental setup and results can be found in Section 4, related work in Section 5, and the conclusion and future work in Section 6.

2. LoPSiL: A Location-based Policy-specification Language

Policy-specification languages enable the previously mentioned centralized security modules to be specified as a policy. This section describes how LoPSiL structures policies while providing language constructs for accessing and manipulating location information.

2.1. Core Linguistic Constructs

LoPSiL is built on six core abstractions:

Locations

In LoPSiL, `Location`s are places. They may be abstract places, such as rooms, floors, buildings, or campuses, or concrete places given by GPS coordinates. LoPSiL provides many built-in utility methods for manipulating GPS locations such as calculating distances between locations, defining regions of locations, and making containment queries to determine whether and how points and/or regions overlap.

LocationDevices

A `LocationDevice` is LoPSiL's interface to real-time location information. Concrete `LocationDevice`s must implement two abstract methods. The first simply informs policies of the device's current location, which could be determined using GPS, Wi-Fi triangulation, IP address geolocation, or by inputting location information from a user, a file, or another (networked) host, etc. The second abstract method `LocationDevice`s must implement informs LoPSiL policies of the device's location accuracy (or granularity), that is, with what precision is the device's location information accurate (e.g., accurate within 1 meter, 1 kilometer, etc). LoPSiL policies can require devices to provide location information with particular granularity thresholds.

Our LoPSiL implementation includes concrete implementations of three `LocationDevice`s: one represents and connects to a Garmin GPS device using Java's communication API and the GPSLib4J library [20], another provides a simple GUI with which users can manually select their current location from a list of known locations, and another is tailored to the Android API [21] and uses Android's GPS. Other implementations of `LocationDevice`s are also possible.

PolicyAssumptions

LoPSiL policies may make two important requirements of `LocationDevice`s. First, as mentioned above, a policy may require location information with a particular granularity (e.g., accurate within 15m). Second, a policy may require that location updates arrive with a particular frequency (e.g., a new update must arrive within 10 seconds of the previous update).

LoPSiL policies encapsulate these requirements, along with the `LocationDevice`s whose location data they trust, in a `PolicyAssumptions` object. A LoPSiL policy gets notified automatically whenever a `LocationDevice` violates the policy's granularity or frequency-of-updates requirements.

Actions

An `Action` encapsulates information about a *security-relevant method* (i.e., any Java application or library method of relevance to a LoPSiL policy). LoPSiL policies can interpose before and after any security-relevant action executes; the policy specification then determines whether that action is allowed to execute. Policies may analyze `Action` objects to determine which security-relevant method the action represents, that method's signature, run-time arguments, calling object (if one exists), whether the method is about to execute or has just finished executing, and the return value of the action if it has finished executing.

Reactions

LoPSiL policies convey decisions about whether and how to allow security-relevant `Action`s to execute by returning, for every `Action` object, a `Reaction` object. LoPSil implements the following reactions: an *OK reaction* indicates that the action is safe to execute; an *exception reaction* indicates that the action is unsafe, so an exception should be raised (which the application may catch) instead of allowing the method to execute; a *replace reaction* indicates that the action is unsafe, so a precomputed return value should be returned to the application in place of executing the unsafe action; and a *halt reaction* indicates that the action is unsafe, so the application program should be halted.

Policies

For expressiveness, LoPSiL policies incorporate all of the previously described language constructs. There are four parts to a LoPSiL Policy object:

1. A policy may declare `PolicyAssumptions` upon which it relies.
2. A policy may define a `handleGranularityViolation` method and a `handleFrequencyViolation` method, which get invoked when either all the `LocationDevice`s upon which the policy relies violate the policy's location-granularity assumption, or the `LocationDevice`s violate the policy's frequency-of-updates assumption.
3. A policy may define an `onLocationUpdate` method, which will be executed any time any `LocationDevice` associated with the policy updates its `Location` information. This method enables a policy to update its security state and take other actions as location updates occur in real time.
4. A policy must define a `react` method to indicate how to react to any security-relevant method. LoPSiL requires every policy to contain a `react` method, rather than providing a default allow-all `react` method; hence, policy authors wanting to allow all security-relevant methods to execute unconditionally must explicitly specify their policy to do so.

Figure 1 contains a simple LoPSiL policy with all of these components. Existing policy-specification languages, such as Naccio [10], PSLang [9], and Polymer [11], provide constructs similar to our `Actions`, `Reactions`, and `Policy` modules with

```
public class AllowAll extends Policy {
  public LocationDevice[] devices = { new AndroidGPS() };
  public LocationGranularityAssumption lga =
    new LocationGranularityAssumption(15, Units.METERS);
  public FrequencyOfUpdatesAssumption foua =
    new FrequencyOfUpdatesAssumption(10, Units.SECONDS);
  public PolicyAssumptions pa =
    new PolicyAssumptions(this, devices, lga, foua);
  public void handleGranularityViolation() {System.exit(1);}
  public void handleFrequencyViolation() {System.exit(1);}
  public synchronized void onLocationUpdate() {
    System.out.println("new location = " + devices[0].getLocation());
  }
  public synchronized Reaction react(Action a) {
    return new Reaction("ok");
  }
}
```

Figure 1: Simple LoPSiL policy that prints location information as it is updated and allows all security-relevant methods to execute as long as its location-granularity and frequency-of-update assumptions are not violated.

`react`-style methods. LoPSiL's novelty is its addition of optional location-related policy components: `Locations`, `LocationDevices`, granularity and frequency-of-update assumptions, and methods to handle granularity and frequency-of-update violations and to take action when location state gets updated (with the `onLocationUpdate` method).

2.2. Example Policies

We next survey four location-dependent runtime policies and show how to specify them in LoPSiL. The first is an example of the sort of policy a user might wish to enforce on untrusted third-party software; it is able to restrict the functionality of the original application under certain constraints. The second to fourth are examples of policies an application developer might wish to enforce on their own software; these

```
public class NoGpsOutsideWorkTime extends Policy {
  public LocationDevice[] devices = { new AndroidGPS() };
  ...
  public synchronized Reaction react(Action a) {
    if (matchesGpsRead(a) && !isWorkTime())
      //return a null location to the application
      return new Reaction("replace", null);
    else return new Reaction("ok");
  }
}
```

Figure 2: LoPSiL policy preventing an application from reading GPS data outside of work hours.

policies enhance the original application's functionality. We have enforced and tested versions of all these example policies on Java applications, executing both on a roaming laptop and on an Android mobile phone, as described later in Section 3.2.

Access-control Policy

Our first example is a privacy-based access-control policy that constrains an application's ability to read location data at particular times. The policy, shown in Figure 2, requires that monitored applications only access the device's GPS data from 8am to 6pm on workdays. A user might want to enforce such a policy to prevent an employer-provided application from learning the device's location when the user is not at work (e.g., so the employer does not know where the employee shops, or how much time the user spends in certain places during the employee's off hours). In fact, users' ability to install such policies themselves might be the only way the user would allow the work-related application to run on their mobile device.

```
public class ShowNavigation extends Policy {
  public LocationDevice[] devices = { new AndroidGPS() };
  ...
  public synchronized void onLocationUpdate() {
    if (distanceToExpectedPath(devices[0].getLocation(), Units.MILES) > .5)
      displayNavigationalAid();
    else clearNavigationalAid();
  }
}
```

Figure 3: Abbreviated LoPSiL policy requiring that navigational aid appear when the device's current location deviates from its expected path.

Deviation-from-path Policy

Our second example policy requires navigational aid to appear when the device's location deviates more than some specified distance off its expected path. The policy code, shown in Figure 3, invokes a method called `distanceToExpectedPath` to determine how far the user has drifted off course.

Safe-region Policy

Another example policy expressible in LoPSiL is shown in Figure 4. This policy, intended to monitor software on a robot, requires the robot to encrypt all outgoing communications when the robot's location is outside a secure-region perimeter. This policy is a subset of the type of policies a company might enforce on all of its applications to ensure the protection of intellectual property or confidential data when the application user is not in a trusted zone. When the application user is in a trusted zone, though, the software is spared from having to perform costly cryptographic operations.

```
public class SafeRegion extends Policy {
  private Location[] safeRegionEndpoints;
  private boolean inRegion;
  public SafeRegion() {
    safeRegionEndpoints = getSafeRegionLocs();
    inRegion = devices[0].getLocation().inRegion(safeRegionEndpoints);
  }
  public PolicyAssumptions pa = ...
  public synchronized void onLocationUpdate() {
    inRegion = devices[0].getLocation().inRegion(safeRegionEndpoints);
  }
  public synchronized Reaction react(Action a) {
    if(!inRegion && ActionPatterns.matchesPlainWrite(a)) {
      String encMsg = encrypt(a.getArgs()[0].toString());
      try { //to replace the unencrypted send with an encrypted send
        ((BufferedWriter)(a.getCaller())).write(encMsg);
      } catch(IOException e) {...}
      return new Reaction("replace", null);
    } else return new Reaction("ok");
  }
}
```

Figure 4: Abbreviated LoPSiL policy requiring robot-control software to encrypt outgoing messages when the robot is outside a secure-region perimeter.

Social-networking Policy

Our final example is a social-networking policy in which the user's friends get invited to rendezvous when the user travels to a new location. Specifically, the policy requires that if:

- the device has traveled more than 50 km over the past 2 hours (i.e., average speed has been more than 25km/hr),
- the device has traveled less than 100m over the past 10 minutes (implying that the user's travels have at least temporarily ended), and
- the policy enforcer has not sent invitations to friends in the past hour,

```
public class InviteFriendsInNewArea extends Policy {
  //maintain a buffer of two hours' worth of location data
  private LocBuffer longBuf = new LocBuffer(2, Units.HOURS);
  //maintain another buffer of twenty minutes' worth of location data
  private LocBuffer shortBuf = new LocBuffer(10, Units.MINUTES);
  private Time timeLastInvited = Time.NEVER;
  ...
  public synchronized void onLocationUpdate() {
    Location currentLoc = devices[0].getLocation();
    longBuf.add(currentLoc);
    shortBuf.add(currentLoc);
    if(longBuf.earliest().distance(currentLoc, Units.KILOMETERS)>50
      && shortBuf.earliest().distance(currentLoc, Units.METERS)<100
      && timeLastInvited.elapsed(Time.getCurrentTime(),Units.HOURS)>1)
    {
      Location[] friendLocs = getFriendLocations();
      inviteLocalFriends(friendLocs,currentLoc,1,Units.KILOMETERS);
      timeLastInvited = Time.getCurrentTime();
    }
  }
}
```

Figure 5: A location-dependent social-networking policy specified in LoPSiL.

then the enforcement system must:

- broadcast a "Where are you?" message to all friends in the user's address book,
- collect responses from the friends, and
- send invitations to meet to those friends now within 1km of the user.

An abbreviated LoPSiL policy specifying such constraints appears in Figure 5.

3. A LoPSiL Compiler

Due to the popularity of Java, particularly Java ME, as an application programming language for mobile devices [22], we have chosen to design and implement LoPSiL constructs in Java source code. Also, to make it easy for security engineers to learn and use LoPSiL, and to simplify the implementation of a LoPSiL compiler, we have packaged LoPSiL as a Java library, to which LoPSiL policies may refer (e.g., a LoPSiL policy may refer to the Location class in the LoPSiL library). Although we treat LoPSiL in a Java context in this paper, we have built LoPSiL on six core abstractions that are application-language independent, so we expect LoPSiL to be portable to other languages and platforms.

This section describes our implementation of LoPSiL and its port to Android, and briefly reports on our experiences designing and implementing LoPSiL policies. Our implementation is available online [23].

3.1. Compiler Architecture

A LoPSiL compiler takes as input a LoPSiL policy and an untrusted application, builds a trustworthy application by inserting code into the untrusted application to enforce the input policy, and then outputs the trustworthy application. The standard technique for implementing such a compiler involves inlining policy code into the untrusted application. Several tools exist for inlining code into an application. A convenient tool for our purposes is the AspectJ compiler [24]. AspectJ compilers inline calls to advice at control-flow points specified by point cuts, modifying a target program by inserting new lines of code [25]. In the domain of runtime policy enforcement, advice refers to policy-enforcement code and point cuts refer to the set of security-relevant methods. We wish to interpose and allow policy-enforcement code to execute before and after any security-relevant method invoked by the untrusted application.

LoPSiL users convert an untrusted application into a more trustworthy application as follows:

```
void java.io.PrintStream.println(..)
* javax.swing.JOptionPane.*(..)
java.util.Date.new()
```

Figure 6: Example `.srm` file indicating that the accompanying LoPSiL policy considers security relevant all void-returning `java.io.PrintStream.println` methods, all methods in the `javax.swing.JOptionPane` class, and the parameterless constructor for `java.util.Dates`.

1. The user creates a specification of the desired policy in a `.lopsil` file.
2. The user also creates a listing of all the methods the desired LoPSiL policy considers security relevant. This listing indicates to the compiler which application and library methods it needs to insert policy-enforcement code around. Policies get to interpose and decide whether (and how) all security-relevant methods may execute. The listing of security-relevant methods goes into a `.srm` file, one method signature per line. Figure 6 contains an example and illustrates how wildcards can be used in `.srm` files.
3. The LoPSiL compiler inputs the policy (`.lopsil`) and security-relevant-methods (`.srm`) into a `lopsil2aj` converter, which converts LoPSiL code into AspectJ code. The converter, implemented in 201 lines of Java, begins by converting the LoPSiL policy to Java source (in a `.java` file) by simply inserting three lines of code to import LoPSiL-library classes into the policy.
4. The converter then creates an AspectJ-code file (`.aj`) that defines two things. First, the AspectJ code defines a pointcut based on the declared security-relevant methods. Second, the AspectJ code defines advice to be executed whenever the pointcut gets triggered (i.e., before and after any security-relevant method executes). This advice builds an `Action` object to represent the invoked security-relevant method, passes that `Action` to the LoPSiL policy (now in a `.java` file), obtains the policy's `Reaction` to the

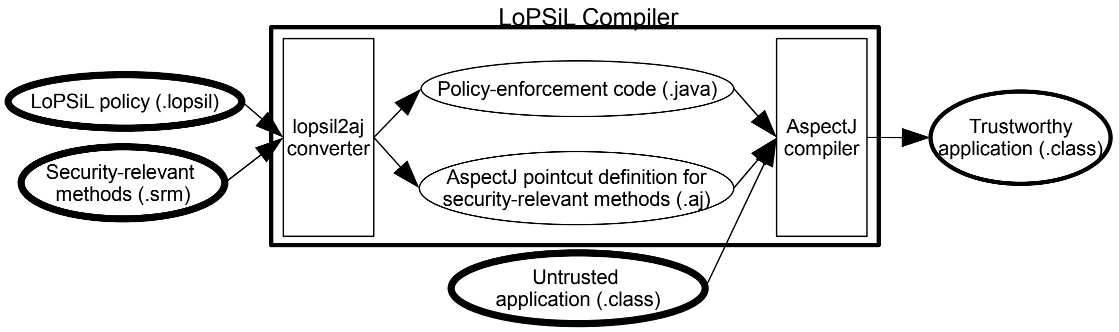
LoPSiL Compiler
LoPSiL policy (.lopsil)
Security-relevant methods (.srm)
lopsil2aj converter
Policy-enforcement code (.java)
AspectJ pointcut definition for security-relevant methods (.aj)
AspectJ compiler
Untrusted application (.class)
Trustworthy application (.class)

Figure 7: Overview of the LoPSiL compiler. The compiler inputs `.lopsil`, `.srm`, and `.class` files and outputs the same `.class` files but with policy-enforcement code inlined before and after all security-relevant methods.

`Action`, and guides execution appropriately based on that `Reaction`.

5. Finally, the LoPSiL compiler inputs the untrusted mobile-device application (comprised of a set of `.class` files) and the `.java` and `.aj` files created in Step 3 into a standard AspectJ compiler [24]. The AspectJ compiler inlines the advice into the application before and after all security-relevant methods, thus producing an application that is secure with respect to the original LoPSiL policy.

Figure 7 presents an overview of this architecture.

Because LoPSiL uses AspectJ as its application rewriter, LoPSiL inherits AspectJ's limitations. Most importantly, the AspectJ compiler cannot rewrite (i.e., inline code into) methods in standard Java libraries; it can only rewrite application files. Therefore, our LoPSiL compiler can only ensure that policy-enforcement code executes before and after security-relevant methods invoked by the application being monitored. While our implementation allows enforcement mechanisms to interpose and make decisions concerning library methods, the enforcement mechanisms cannot then repeat these decisions upon any library methods being called by the monitored library methods. We could circumvent this limitation by writing our own LoPSiL enforcement-code inliner (e.g., using tools like the Bytecode Engineering Library [26]), as previous work has done [9, 11]. Moreover, to save users from the effort of creating an `.srm` file, we could automatically extract a set of security-relevant methods by performing static analysis on policy code [27, 28, 29]. For the sake of simplicity, we did not include such features in our proof-of-concept compiler.

3.2. Porting LoPSiL to Android

The Android operating system uses a special Java virtual machine, the Dalvik VM, which has its own `.dex` file format rather than normal `.class` files. AspectJ, as described in Section 3.1, can still be used to inline code into applications. Running Java classes on Android requires translating the policy classes and the external LoPSiL and AspectJ libraries into the Dalvik VM's `.dex` format, then packaging the files into Android's `.apk` application format. Conveniently, the Android SDK [21] provides plug-ins for the Eclipse IDE [30] to automate this process, as well as emulators for development and testing.

Our implementation of LoPSiL on Android compiles and inlines LoPSiL policies using the same five-step procedure described, but with one minor addition: the compiler-produced `.aj` file also contains a method that listens for GPS location updates from the Android OS and forwards them to the policy. This extra method uses the `.lopsil` policy's given `FrequencyOfUpdatesAssumption` to specify the frequency with which it expects location updates from the Android OS.

The lifecycle of a mobile application differs from a PC-based application. Consequently, whereas LoPSiL policies on a PC application need to concern themselves with an application starting and ending, LoPSiL policies on Android applications have many more possibilities. An `onResume` function is called when an application is brought to the foreground and `onPause` when it no longer holds the foreground. The two functions that define when an application can be visible and when it ceases being visible are `onStart` and `onStop`. Finally, initialization and completion are controlled by `onCreate` and `onDestroy`. Throughout the lifecycle, an application can lose and regain focus or be stopped and restarted (through `onRestart`) many times before the process is finally destroyed. Mobile applications also have more interrupts with which a policy may be concerned—an incoming phone call or text message, the GPS being turned off, the mobile device having moved out of Wi-Fi range, and so on. LoPSiL policies can interpose on and react to all these events.

Our port of LoPSiL to Android is a proof-of-concept implementation of a location-based policy-specification language on a mobile device. The port proved to be straightforward; only minor changes had to be made to LoPSiL (due to Android's omission of certain standard Java libraries and the way Android applications can display data). The core logic of the policies implemented on Android was unchanged from Section 2.2 other than changes to the graphical interface. We expect the porting of LoPSiL to any platform that runs Java to be a similarly straightforward experience.

4. Experimental Results

Our proof-of-concept implementation of LoPSiL on an Android mobile phone platform had multiple objectives. First, we wanted to evaluate the expressiveness of the language for real location-specific scenarios. For this, we implemented four location-based policies similar to the ones presented in Section 2.2 in four respective target applications. Second, we wanted to test the practicality of using LoPSiL on a typical, resource-constrained mobile device. Many such constrained devices take great care to limit the strain upon the hardware by imposing restrictions on the operating system (e.g., the iPhone prohibits most forms of multitasking, allowing only a small set of operations like audio or push notifications to run in the background). With these restrictions in mind, we evaluate experimentally the impact of enforcing LoPSiL policies, in terms of execution performance, memory usage, battery usage, and code size, to show that the action of policy enforcement does not impose a significant burden on resource-constrained mobile devices.

4.1. Experimental Setup

We implemented the four location-based policies presented in Section 2.2 on an Android G1. This device, an unlocked HTC Dream, has a 528 MHz processor, 192 MB RAM, 1 GB storage, 1150 mAh lithium ion battery, as well as a 3.2 megapixel camera, GPS, BlueTooth, and Wi-Fi support.

We enforced LoPSiL policies on the following four Android applications:

1. The first application repeats a numerical calculation 1000 times. On this application we enforced the `AllowAll` policy of Figure 1. The policy returns an `OK` reaction for every call made to the security-relevant method `compute`. The purpose of this policy is to demonstrate the amount of overhead the LoPSiL framework produces when enforcing the simplest policy possible. Other policies will have additional overhead based on their complexity.
2. The second application uses the device's GPS to deliver a status update to an employer's server every 5 seconds. The update contains an employee's current location and other work-related information. The `AccessControl` LoPSiL policy we enforced on the application is a version of the `NoGpsOutsideWorkTime` policy shown in Figure 2. It uses a `replace` reaction to omit the employee's location from the message sent to the server outside of work hours.
3. The third application uses the device's GPS and the Google Maps API to display the user's current location on a map and an expected path that the user

will travel. The `DeviationFromPath` LoPSiL policy we enforced on the application matches the policy in Figure 3. The `onLocationUpdate` method in our `DeviationFromPath` policy prints a message if the distance between the user and the expected path exceeds a certain distance

4. The fourth application uses the device's GPS and the Google Maps API to display the user's current location on a map. We enforced the `SocialNetworking` LoPSiL policy (Figure 5), which uses `onLocationUpdate` to determine if the user has stopped traveling (defined as having traveled a distance of 50 km over the past 2 hours). If the user has stopped traveling, the policy sends a request to a server to invite any friends who are nearby, given that no invitations have been sent in the last hour.

We ran each application until either 1000 function calls had been made or 1000 location updates had been received. For the `SocialNetworking` policy, we accelerated the rate at which friend invitations were sent in order to collect sufficient data. To keep the comparison apt, we kept the ratio of policy-enforced friend invitations to location updates equivalent to the same ratio in the original policy. To determine the performance and memory impact of a LoPSiL policy, we ran each application both with and without enforcing their LoPSiL policies.

Policy running times were measured by timing both `react` and `onLocationUpdate` methods. Memory usage was measured by a script running on the device that logs the output of the `meminfo` system service at intervals of 5 seconds. Because the device has a single CPU, `meminfo` and other system services are sometimes switched to run in the middle of a `react` or `onLocationUpdate` method; when this occurs, the resulting performance measurement of the interrupted method is not included in the collected data. To measure battery usage, each application ran for 30 minutes, using a

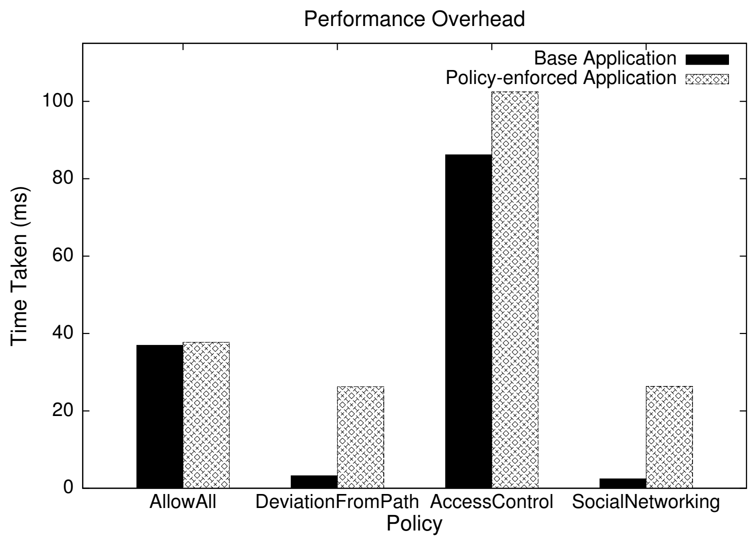

Figure 8: Performance overhead of the implemented Android applications with and without policy-enforcement.

mechanism to prevent the device from turning its screen off and going to sleep. Without this mechanism, it would have been impossible to determine whether any battery drain was related to the policy or to other processes running on the mobile device. It must be noted, however, that preventing the device from going to sleep is a worst-case scenario. Mobile devices are typically used interactively; when not in current use, the devices go to sleep. The battery usage information was recovered from an API method that has a granularity of one percent. The battery experiments were repeated eight times for consistency.

4.2. Results

Figure 8 shows that the performance overhead of the `AllowAll` policy is a little over 2%, or, in absolute terms, 0.74 ms for every monitored function call. To assess the significance of this overhead, we ran a two-sample t-test in which we compared the time taken for the `compute` method to run with and without the `AllowAll` policy. We obtained a t-value of 1.61, thereby confirming that the overhead induced was

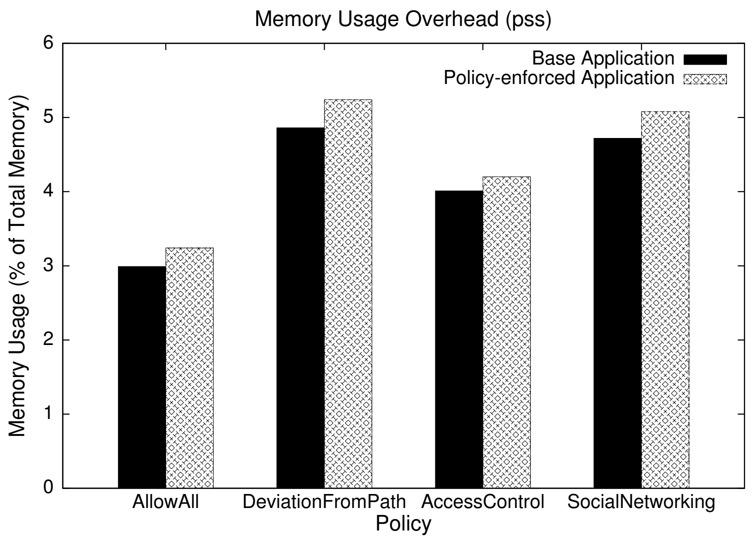

Figure 9: Memory overhead of the implemented Android applications with and without policy-enforcement.

statistically insignificant at the 1% confidence levels. The base overhead of the framework when invoking a policy therefore does not significantly affect performance—an important factor for CPU-limited mobile devices. Longer-running enforcement code induces greater running-time overhead; the `DeviationFromPath`, `AccessControl`, and `SocialNetworking` policies add about 20 ms of overhead per location update and function call to the application. Across all four policies, we find an average performance overhead per monitored function call or location update of 8.6 ms, or 27.4% (with the 95% confidence interval between 23.4% and 31.3%).

Android has two metrics available for memory usage of an application: the amount of private memory in use (unique set size or uss) and the amount of total memory in use, which includes shared pages (proportional set size or pss). Figure 9 shows that the memory usage (pss) of the policy-enforced application very closely matches the memory usage of the application run by itself. The y-axis represents the percentage of total memory (including shared pages) that the application uses. On average, the use

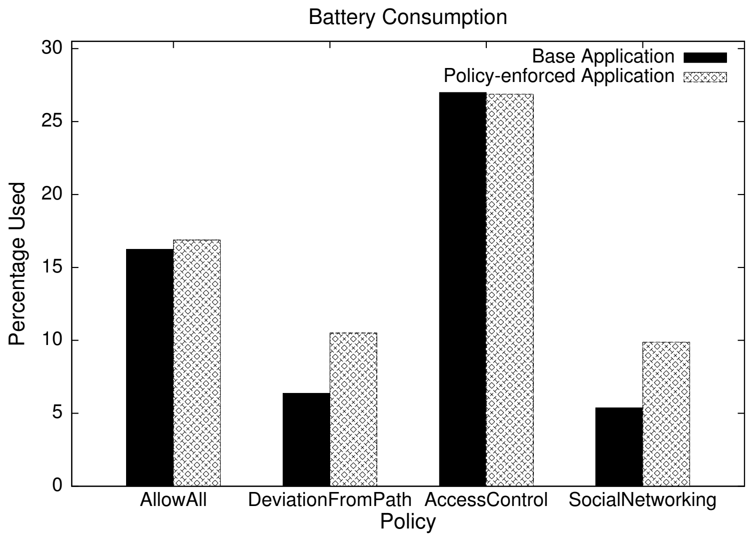

Figure 10: Measured battery consumption of an Android application vs. the policy-enforced application.

of a LoPSiL policy took an extra 291 bytes of memory, or 6.6% more memory (with a 95% confidence interval from 5.8% to 7.4%), compared to the memory usage of the application without a policy, less than 0.3% of the total memory available on the G1 device.

Battery results in Android are reported at the granularity of 1%, so results should be considered accurate within 1%. Figure 10 shows that the `AllowAll` and `AccessControl` policies have no significant impact on battery overhead, as the numbers reported are within 1% of each other. The `DeviationFromPath` and `SocialNetworking` policies use up a little more than 4% extra battery over a span of 30 minutes of execution. It should be noted that the `SocialNetworking` policy is simulated at an accelerated pace, sending messages to a server once every 5 seconds instead of once every hour. Overall, the battery overhead incurred from using LoPSiL was on average 16.6% (the 95% confidence interval is between -3.2% to 36.4%).

The code size is particularly important in smartphones and other mobile devices

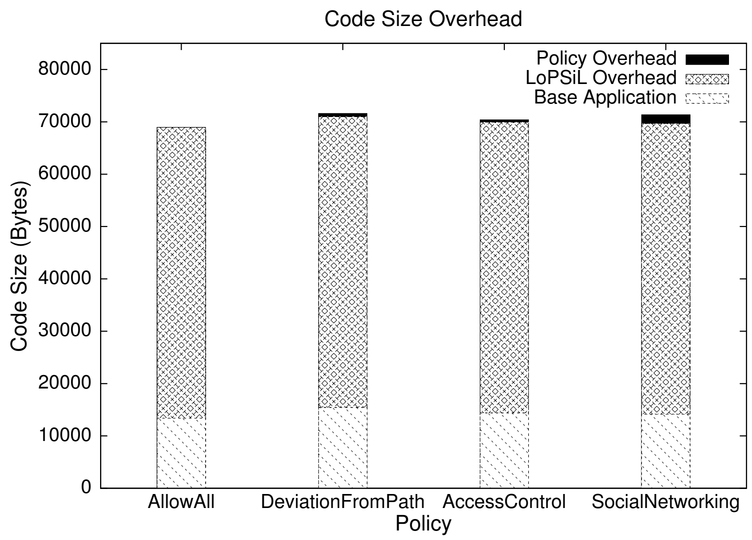

Figure 11: Code size of the implemented Android application with and without policy-enforcement. The LoPSiL overhead represents the constant space required for the LoPSiL and AspectJ libraries, while the policy overhead refers to the variable space used by the policy file.

with limited hard drive capacity. The minimum amount of extra space needed, as seen in Figure 11, was 55.5 KB for the `AllowAll` policy. As this policy implements only the minimal constructs (the aspect file and a mostly empty policy), the `AllowAll` policy can viewed as the constant, minimum overhead of the AspectJ and LoPSiL libraries. The average extra space required for our policies was 56.2 KB (or less than one percent of the total available memory on the G1), indicating that the more complex policies do not incur much overhead in terms of code size. Though the overhead needed for the policies is greater than the average code size for the applications in our examples, it is important to note that the base applications we used were all simple and far smaller than actual Android applications commonly in use; a survey of the top 9 featured applications in the Android market[1], for example, shows an average size of 1.96 MB, compared to an average size of 13 KB for the

[1]Top 9 free and paid applications as of March 10, 2010 according to `http://android.com/market`.

example applications. The code size required by the example policies would be much less significant in comparison to these actual Android applications than they appeared in comparison to the example policies.

Summary of Experimental Results. LoPSiL overhead is highly dependent on the specific LoPSiL policy. The purpose of the `AllowAll` policy was to show what overhead would be incurred by adopting the LoPSiL framework. The results—0.74 ms per monitored function call, 244 bytes of runtime (non-code) memory, 55.5 kB of code memory, and a negligible impact on battery life—show that the LoPSiL framework induces quite tolerable overhead on mobile devices such as the G1.

However, the performance, memory, battery, and code size metrics are highly dependent on the actual policy. A policy that enforces that an application must frequently check in with a server will incur much more overhead than a policy that infrequently checks a variable in memory, for example. We presented the `SocialNetworking`, `DeviationFromPath`, and `AccessControl` policies to show that realistic policies can be enforced without having a significantly detrimental effect on the target application. Although these policies were invoked frequently—every 5 seconds for `AccessControl`, 1 second for `DeviationFromPath` and `SocialNetworking`, and many times a second for `AllowAll`—many other policies, like those concerned with whether a user is at a certain location or whether certain conditions hold at the time of `onStart` and `onResume` functions, may invoke the policy methods much less frequently and therefore induce overheads closer to our `AllowAll` policy.

5. Related Work

A rich variety of policy-specification languages and systems have been implemented, which enable users to centrally specify security and privacy policies to be enforced on untrusted software at runtime. Ponder [7], XACML [8], PoET/PSLang [9], Naccio [10], Polymer [11], and Deeds [12] are examples of expressive (i.e., Turing-complete) policy-specification languages. However, none of these languages provide users with built-in constructs for manipulating location information.

SpatialP is a Turing-complete language in which policies can make decisions about a program's actions based on the location of the user [31]. However, users can specify only equality and containment conditions on locations, so SpatialP users can manipulate location information in only limited ways. For example, users cannot specify conditions that capture their being within a certain distance of a fixed point. Thus, SpatialP cannot be used to specify policies such as the `DeviationFromPath` and `SocialNetworking` (shown in Figures 3 and 5). Also, as far as we are aware, SpatialP does not have an implementation.

There are other policy-specification languages that do have primitives for manipulating location information, but as far as we are aware, none of these languages are Turing-complete, which prevents specification of arbitrary policies. For example, OpenAmbient, an access-control architecture for web services, provides a policy-specification language in which policies can reason about location information (as part of the system's ambient state) but cannot maintain a state of their own [13, 14]. Not being able to maintain state prevents policies from (1) basing decisions on previous policy actions and (2) executing arbitrary code in response to application actions. For these reasons, it would be impossible to specify the

`DeviationFromPath` and `SocialNetworking` policies using OpenAmbient.

Another location-based but Turing-incomplete policy-specification language is Geo-RBAC [15]. The Turing-incompleteness of Geo-RBAC derives from its targeting only RBAC (role-based access-control) policies, which are safety policies and therefore a proper subset of the policies enforceable at runtime [32]. In addition, role assignment and location information are fixed per session in Geo-RBAC, so policies on systems with dynamically changing roles or locations (e.g., the policies in Figures 3–5) could not be specified with Geo-RBAC.

Ray and Kumar describe a formal, Turing-incomplete model that extends a MAC system with location primitives [16]. They describe how the location of a subject and an object can be used to make decisions about granting subjects access to objects, while keeping the locations of subjects and objects private from each other. Because their model is an access-control (safety-policy) model, its users cannot specify the `DeviationFromPath` and `SocialNetworking` policies.

The Android operating system also lets users specify and enforce location-based access-control policies [21, 34]. These policies get specified in the form of permissions in XML manifest files and optional implementations of `checkPermission()` methods. Additionally, Ongtang et al. have implemented a system called Saint that extends Android's security model and enables users to specify fine-grained permissions [33]. However, similarly to Geo-RBAC and the model of Ray and Kumar, Saint and Android's built-in enforcement system are limited to access-control policies and therefore cannot enforce more general runtime policies such as `DeviationFromPath` and `SocialNetworking`.

6. Conclusions and Future Work

We have presented LoPSiL, a language for specifying location-dependent runtime security policies. LoPSiL's novelties are its abstractions for accessing and reasoning about location information in expressive policy specifications; existing policy-specification languages up to this point either did not target location-dependent policies or lacked expressiveness (by being confined to access-control policies). In our experiments specifying location-dependent policies for mobile-device applications, we found LoPSiL expressive and efficient. Through experimentation, we have also noticed some common, recurring uses of location information in security and privacy policies. Our location-dependent security policies consistently based policy decisions on:

- The current absolute location of the device (e.g., whether the device is in the user's office);
- The geographic relationship of the device's current location with another location (e.g., whether the device is north of or within 1km of another location);
- The geographic relationship of the device's current location with a region of locations (e.g., whether the device is in an area of trusted terrain or within 10m of an expected path);
- The velocity or acceleration of the device.

Because location-dependent policies consistently use location information in these ways, LoPSiL provides core linguistic constructs, as well as utility methods, designed to make it easy for policies to calculate and keep track of distances, boundaries, velocities, and accelerations between locations, all in a centralized policy module.

Future work could improve the usability of the process of policy creation and enforcement. Presently, users have to create and compile their own LoPSiL policies before being able to run an application with inlined policy-enforcement code. One helpful extension would be to create a centralized LoPSiL policy repository, where like-minded LoPSiL developers could share policies that either restrict or enhance the functionality of applications. Such a repository, in conjunction with an easy-to-use interface, could provide non-technically inclined users the ability to select and apply LoPSiL policies to enhance their mobile-device security. A user-friendly "Security Center" application could provide the mechanism for downloading and automatically enforcing LoPSiL policies in such a repository. By increasing the ease of use of locating and managing LoPSiL policies, we hope to foster a community of policy creation and sharing as a bastion against the current untamed state of mobile applications.

Acknowledgments

We would like to thank Billy Rickey for helping implement the original version of LoPSiL. This research was funded in part by National Science Foundation grants CNS-0716343 and CNS-0831785. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] GigaOM, The apple app store economy (Feb. 2010).
URL `http://gigaom.com/2010/01/12/the-apple-app-store-economy/`

[2] Apple, Apple's app store downloads top three billion (Feb. 2010).
URL `http://www.apple.com/pr/library/2010/01/05appstore.html`

[3] R. Wauters, Google: Android market now serving 30,000 apps (Mar. 2010).

URL http://www.mobilecrunch.com/2010/03/16/google-android-market-now-serving-30000-apps

[4] MobileBits.net, Mobile app stores to register purchases worth of $6.2 billion in 2010 (Feb. 2010).
URL http://www.mobilebits.net/items/view/4599

[5] Qualcomm, Qualcomm products and services - the Snapdragon platform (Feb. 2010).
URL
http://www.qualcomm.com/products_services/chipsets/snapdragon.html

[6] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, A. N. Sheth, Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones, in: To appear at the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10), 2010.

[7] N. Damianou, N. Dulay, E. Lupu, M. Sloman, The Ponder policy specification language, Lecture Notes in Computer Science 1995 (2001) 18–39.

[8] OASIS, eXtensible Access Control Markup Language (XACML) version 2.0 (2005).

[9] Ú. Erlingsson, F. B. Schneider, IRM enforcement of Java stack inspection, in: IEEE Symposium on Security and Privacy, Oakland, CA, 2000.

[10] D. Evans, A. Twyman, Flexible policy-directed code safety, in: IEEE Security and Privacy, Oakland, CA, 1999.

[11] L. Bauer, J. Ligatti, D. Walker, Composing expressive runtime security policies, ACM Transactions on Software Engineering and Methodology 18 (3) (2009) 1–43.

[12] G. Edjlali, A. Acharya, V. Chaudhary, History-based access control for mobile code, in: ACM Conference on Computer and Communications Security, 1998.

[13] M. Anisetti, C. Ardagna, V. Bellandi, E. Damiani, OpenAmbient: A pervasive access control architecture, in: A. Schmidt, M. Kreutzer, R. Accorsi (Eds.), Long-Term and Dynamical Aspects of Information Security: Emerging Trends in Information and Communication Security, Nova Science Publisher, Inc., 2007.

[14] C. A. Ardagna, M. Cremonini, E. Damiani, S. D. C. di Vimercati, P. Samarati, Supporting location-based conditions in access control policies, in: Symposium on Information, computer and communications security, 2006.

[15] E. Bertino, B. Catania, M. L. Damiani, P. Perlasca, Geo-rbac: a spatially aware rbac, in: SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies, 2005, pp. 29–37.

[16] I. Ray, M. Kumar, Towards a location-based mandatory access control model, Computers and Security 25 (1) (2006) 36–44.

[17] J. Ligatti, B. Rickey, N. Saigal, LoPSiL: A location-based policy-specification language, in: International ICST Conference on Security and Privacy in Mobile Information and Communication Systems (MobiSec), 2009.

[18] M. Hamblen, Symbian, Android will be top smartphone OSes in '12, Gartner reiterates (Feb. 2010).
URL `http://www.computerworld.com/s/article/9139301/Symbian_Android_will_be_top_smartphone_OSes_in_12_Gartner_reiterates`

[19] H. Miller, Google's Android apps may soon top 150,000 (Feb. 2010).
URL `http://www.mercurynews.com/news/ci_14058873?source=rss`

[20] SourceForge.net, GPSLib4J v0.1 (2009).
URL `http://gpslib4j.sourceforge.net/`

[21] Google, Android Developers (Feb. 2010).
URL `http://developer.android.com/`

[22] Sun Microsystems, The Java ME Platform - the Most Ubiquitous Application Platform for Mobile Devices (2009).
URL `http://java.sun.com/javame/index.jsp`

[23] J. Finnis, B. Rickey, N. Saigal, A. Iamnitchi, J. Ligatti, LoPSiL Implementation (2009).
URL
`http://www.cse.usf.edu/~ligatti/projects/runtime/LopsilAndroid.zip`

[24] A. Coyler, A. Clement, W. Isberg, M. Kersten, A. Vasseur, M. Webster, The AspectJ Project (2009).
URL `http://www.eclipse.org/aspectj/`

[25] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, An overview of AspectJ, in: European Conference on Object-oriented Programming, Springer-Verlag, 2001.

[26] Apache, Byte Code Engineering Library, Apache Software Foundation (2003).
URL `http://jakarta.apache.org/bcel/`

[27] X. Zhang, A. Edwards, T. Jaeger, Using CQUAL for static analysis of authorization hook placement, in: Proceedings of the 11th USENIX Security Symposium, USENIX Association, Berkeley, CA, USA, 2002.

[28] V. Ganapathy, T. Jaeger, S. Jha, Automatic placement of authorization hooks in the linux security modules framework, in: CCS '05: Proceedings of the 12th ACM Conference on Computer and Communications Security, ACM, New York, NY, USA, 2005.

[29] V. Ganapathy, T. Jaeger, S. Jha, Retrofitting legacy code for authorization policy enforcement, in: SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy, IEEE Computer Society, Washington, DC, USA, 2006.

[30] Eclipse, Eclipse.org home (Feb. 2010).
URL `http://www.eclipse.org/`

[31] D. Scott, A. Beresford, A. Mycroft, Spatial security policies for mobile agents in a sentient computing environment, in: 6th Fundamental Approaches to Software Engineering (FASE), volume LNCS 2621, Springer-Verlag, 2003, pp. 102–117.

[32] J. Ligatti, L. Bauer, D. Walker, Run-time enforcement of nonsafety policies, ACM Transactions on Information and System Security 12 (3) (2009) 1–41.

[33] M. Ongtang, S. McLaughlin, W. Enck, P. McDaniel, Semantically rich application-centric security in android, in: ACSAC '09: Proceedings of the 2009 Annual Computer Security Applications Conference, IEEE Computer Society, Washington, DC, USA, 2009.

[34] W. Enck, M. Ongtang, P. McDaniel, Understanding android security, IEEE Security and Privacy 7 (1) (2009) 50–57.

List of Figures