LoPSiL: A Location-based Policy-specification Language

Jay Ligatti, Billy Rickey*, and Nalin Saigal

Department of Computer Science and Engineering
University of South Florida
`{ligatti,brickey,nsaigal}@cse.usf.edu`

Abstract. This paper describes the design of LoPSiL, a language for specifying location-dependent security and privacy policies. Policy-specification languages like LoPSiL are domain-specific programming languages intended to simplify the tasks of specifying and enforcing sound security policies on untrusted (i.e., potentially insecure) software. As far as we are aware, LoPSiL is the first imperative policy-specification language to provide abstractions specifically tailored to location-dependent policies for mobile-device applications. We have implemented a proof-of-concept compiler that inputs a LoPSiL policy P and a mobile-device application program A and outputs a new application program A' equivalent to A, except that A' contains inlined enforcement code that ensures that A' satisfies P at runtime. We report our experiences using this compiler to design and implement several policies for mobile-device applications.

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

1 Introduction

Policy-specification languages are domain-specific programming languages intended to simplify the tasks of specifying and enforcing sound security policies on untrusted (i.e., potentially insecure) software. There are two common motivations for using policy-specification languages:

1. Users often wish to download and execute third-party software applications but do not (and should not) trust that those applications will behave securely or respect their privacy. Therefore, users (or other parties working on behalf of the users, such as vendors or system administrators) may utilize policy-specification languages to create and enforce customized, flexible constraints (i.e., policies) on the untrusted software. For example, a policy-specification language may be used to specify and enforce that applications downloaded from third-party providers do not delete particular files or read particular regions of memory.

---

* Rickey participated on this project as an NSF-REU (National Science Foundation Research Experience for Undergraduates) student at USF in the summer of 2007.

2. Application developers often wish to enforce security policies on their own code. Typically, application developers implement security and privacy considerations by scattering security checks throughout their code. Policy-specification languages and compilers enable those developers to refactor their code by moving all the scattered security checks from the application implementation and into a separate, isolated policy module (written in a language designed to make it easy to specify the desired policy). 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.

A rich variety of expressive, Turing-complete policy-specification languages and systems has been implemented [16, 10, 9, 11, 12, 21, 5, 13]. 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 into trustworthy applications by inputting a policy P and an application program A and outputting a new application program A' equivalent to A, except that A' contains inlined enforcement code that ensures that A' satisfies P at runtime.

However, as far as we are aware, no expressive policy-specification languages have yet targeted location-dependent policies for securing applications on mobile devices (e.g., roaming laptops, cell phones, robots, PDAs, etc)—though researchers have previously developed declarative (and Turing-incomplete) policy-specification languages limited to location-based access-control policies [7, 2, 1].

The lack of expressive policy-specification languages for mobile-device applications presents a problem—and an opportunity—because:

– The two motivations for policy-specification languages given above are becoming more relevant for mobile-device applications as mobile devices become more powerful, open, and flexible and allow users to download and execute third-party software.
– Security and privacy policies for applications on mobile devices often have to reason about different machine states—*locations* in particular—than policies for applications on immobile devices.

Hence, it is important to consider how to create a convenient policy-specification language specifically for mobile-device applications. This paper proposes such a language, called LoPSiL, whose primary novelty is to provide several abstractions for conveniently accessing and manipulating location information in policy specifications.

Roadmap. We proceed as follows. Section 2 describes the design of LoPSiL, 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 our proof-of-concept implementation of a LoPSiL compiler and experiences we have had designing and implementing LoPSiL policies. Section 4 concludes and describes possible future work.

2 LoPSiL

Due to the popularity of Java, particularly Java ME, as an application programming language for mobile devices [15], 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.

2.1 Core Linguistic Constructs

LoPSiL is built on six core abstractions; we describe each in turn.

Locations In LoPSiL, `Location`s are (possibly abstract) places. They may refer to rooms, chairs, floors, buildings, campuses, GPS coordinates, regions of a network topology, etc. All `Location`s have an identity (e.g., a room or building name, or coordinates in GPS). LoPSiL provides many built-in utility methods for manipulating GPS locations (e.g., to calculate distances between them), as the examples in Section 2.2 demonstrate. However, LoPSiL users are always free to implement custom methods for manipulating locations (e.g., to define a containment relation over locations, useful for testing whether a room is in a building, a building is on a campus, etc).

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 or by inputting location information from a user, a file, another (networked) host, a TLTA device [22], etc. The second abstract method `LocationDevice`s must implement informs LoPSiL policies of the device's *granularity*, that is, with what precision is the device's location information accurate (e.g., accurate within 1 meter, 1 room, 1 road, 1 building, 1 kilometer, etc). LoPSiL policies can require devices to provide location information with particular granularity thresholds.

Our LoPSiL implementation includes concrete implementations of two `LocationDevice`s, but users are always free to implement others. The first `LocationDevice` provided with LoPSiL represents and connects to a Garmin GPS device using Java's communication API and the GPSLib4J library [14]; the second `LocationDevice` represents and connects to a simple GUI with which users can manually select their current location from a list of known locations.

PolicyAssumptions LoPSiL policies may make two important assumptions about `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 10s of the previous update). LoPSiL policies encapsulate these assumptions, 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 assumptions.

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, and 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 to allow security-relevant `Action`s to execute by returning, for every `Action` object, a `Reaction` object. 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 LoPSiL policies incorporate all of the previously described language constructs. There are five parts to a LoPSiL `Policy` object:

1. A policy may declare `PolicyAssumptions` upon which it relies.
2. A policy may define a `handleGranularityViolation` method, which will be invoked whenever all `LocationDevice`s upon which the policy relies violate the policy's location-granularity assumption.
3. A policy may define a `handleFrequencyViolation` method, which will be invoked whenever all `LocationDevice`s upon which the policy relies violate the policy's frequency-of-update assumption. LoPSiL's `PolicyAssumptions` class implements the multithreading needed to test for frequency-of-update violations.
4. 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.
5. 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 five of these components.

```
public class AllowAll extends Policy {
  public LocationDevice[] devices = {new LopsilGPS(LopsilGPS.GARMIN)};
  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");
  }
}
```

Fig. 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.

Existing policy-specification languages, such as Naccio [12], PSLang [11], and Polymer [5, 4], provide constructs similar to our `Actions`, `Reactions`, and `Policy` modules with `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, while the other three are examples of policies that application developers might wish to enforce on their own software. We have enforced and tested versions of all these example policies on Java applications executing on a roaming laptop.

Access-control Policy[2]

   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 can only access the device's

---

[2] We thank Sean Barbeau at USF's Center for Urban Transportation Research for suggesting this policy.

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

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

```
public class ShowNavigation extends Policy {
  public LocationDevice[] devices = {new LopsilGPS(LopsilGPS.GARMIN)};
  public PolicyAssumptions pa =
  ...
  public synchronized void onLocationUpdate() {
    if(devices[0].getLocation().
        distance(getExpectedCurrentLocation(), Units.METERS)>10)
      AppGUI.displayNavigationalAid();
  }
}
```

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

GPS data from 08:00 (8am) to 18:00 (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 employee is not at work (e.g., so the employer does not know where the employee shops, or how much time the employee spends in certain places during the employee's off hours). In fact, providing for the enforcement of such a policy might be the only way the employer could convince the employee to run a work-related application on the employee's mobile device.

Deviation-from-path Policy Our second example policy requires navigational aid to appear when the device's location deviates more than 10m off its expected path. The policy code, shown in Figure 3, invokes a method called `getExpectedCurrentLocation` to determine where the policy currently expects the device to be. Method `getExpectedCurrentLocation` could return a location based on the route being displayed to the user (as in dashboard-mounted GPS systems), on traffic conditions, on the path the user normally travels in this area, etc.

Safe-region Policy [3]

Another interesting sort of 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.

---

[3] We thank Robin Murphy at Texas A&M University for suggesting this policy.

```
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");
  }
}
```

Fig. 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 area. Specifically, the policy requires that if:

– the device has traveled more than 100km over the past 2 hours (i.e., average speed has been more than 50km/hr),
– the device has traveled less than 2km over the past 20 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,

then the policy enforcer 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 20km of the user.

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

3 A LoPSiL Compiler

This section describes our implementation of LoPSiL and briefly reports on our experiences designing and implementing LoPSiL policies. The implementations of LoPSiL's basic `Location`, `LocationDevice`, `Action`, `Reaction`, and `Policy` modules occupy 1588 lines of Java code, while the implementations of our `GarminGpsDevice` and `LopsilWindowDevice` respectively occupy 847 and 107 lines of Java code. Our implementation is available online [20].

```
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(20, Units.MINUTES);
  private Time timeLastInvited = Time.NEVER;
  public PolicyAssumptions pa = ...
  public synchronized void onLocationUpdate() {
    Location currentLoc = devices[0].getLocation();
    longBuf.add(currentLoc);
    shortBuf.add(currentLoc);
    if(longBuf.earliest().distance(currentLoc, Units.KILOMETERS)>100
      && shortBuf.earliest().distance(currentLoc, Units.KILOMETERS)<2
      && timeLastInvited.elapsed(Time.getCurrentTime(),Units.HOURS)>1)
    {
      Location[] friendLocs = getFriendLocations();
      inviteLocalFriends(friendLocs,currentLoc,20,Units.KILOMETERS);
      timeLastInvited = Time.getCurrentTime();
    }
  }
}
```

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

3.1 Compiler Architecture

A LoPSiL compiler needs to input a LoPSiL policy and an untrusted application, build a trustworthy application by inserting code into the untrusted application to enforce the input policy, and then output 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 an AspectJ compiler [3]. AspectJ compilers inline calls to *advice* at control-flow points specified by *point cuts* [17]. In the domain of runtime policy enforcement, *advice* refers to policy-enforcement code and *point cuts* 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 trustworthy application as follows.

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.

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

Fig. 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.Date`s.
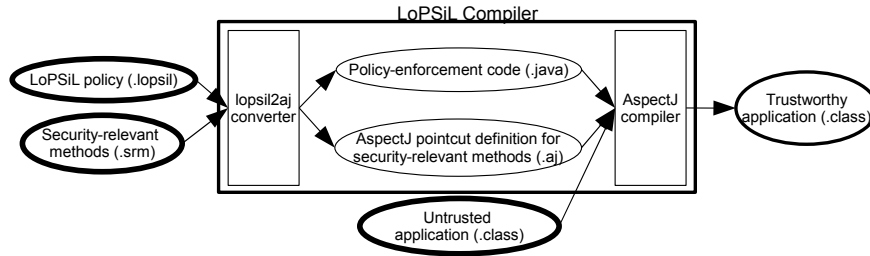
3. The LoPSiL compiler inputs the policy (`.lopsil`) and security-relevant-methods (`.srm`) into a `lopsil2aj` converter, which converts LoPSiL code into AspectJ code. The converter, implemented in 201 lines of Java, begins by converting the LoPSiL policy to Java source (in a `.java` file) by simply inserting three lines of code to import LoPSiL-library classes into the policy. The converter then creates an AspectJ-code file (`.aj`) that defines two things. First, the AspectJ code defines a point cut based on the declared security-relevant methods. Second, the AspectJ code defines advice to be executed whenever the point cut 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 `Action`, and guides execution appropriately based on that `Reaction`.

4. 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 [3]. 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. The important consequence is that our implementation does not allow enforcement mechanisms to interpose and make decisions about the execution of library methods invoked by other library methods. We could circumvent this limitation by writing our own LoPSiL enforcement-code inliner (e.g., using tools like the Bytecode Engineering Library [8]), as previous work has done [11, 5, 4], but at the price of significantly increased implementation complexity.

3.2 Experiential Observations

Having implemented the example policies described in Section 2.2, we believe that the six core constructs underlying LoPSiL serve as good abstractions

Fig. 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.

for specifying location-dependent runtime security policies. This belief stems from the fact that LoPSiL was sufficiently expressive for us to specify every location-dependent policy we considered enforcing. In addition, after implementing LoPSiL, none of the example policies from Section 2.2 took us more than 2 hours to design, specify, and test. Although these results are encouraging, we need more experience with LoPSiL, and feedback from other users, before we can more completely and objectively evaluate its expressiveness and ease of use.

Another interesting outcome of designing the example policies from Section 2.2 is that we have observed some common, recurring uses of location information in security and privacy policies. Our location-dependent 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, we provide several utility methods in LoPSiL for calculating distances, boundaries, velocities, and accelerations between locations. All policies can access these utility methods (cf. Figures 3–5) and can define custom operators on locations when the built-in methods are insufficient.

We have focused our efforts to date on LoPSiL's design, rather than the performance of our proof-of-concept compiler. Although we have not measured the performance overhead induced by LoPSiL-policy enforcement, we refer to previous work to argue that this performance concern is minor in practice when the application code runs on general-purpose, high-power, mobile machines (e.g., laptops), unless the policy itself specifies expensive computations or considers frequently invoked methods to be security relevant [12, 11, 5]. In the future, we would like to explore the performance impact of enforcement systems like LoPSiL on smaller and less powerful mobile devices.

4 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 imperative policy specifications. In our preliminary experiments specifying policies for applications on a mobile laptop, we found these abstractions expressive and convenient. Given the increasing ubiquity of mobile devices, and the unique abstractions needed to conveniently deal with location information in security policies, we believe the research area of location-based policy-specification languages will for some time be an important topic of consideration for the programming-languages, computer-security, and mobile-device-applications research communities.

There are many opportunities for future work. One unresolved question is: how tolerable are the performance degradations that result from enforcing application-level runtime policies on weakly powered, inexpensive, and/or computationally constrained mobile devices? As Section 3.2 mentioned, for simplicity we have only enforced LoPSiL policies on a powerful roaming laptop. In the future we would like to perform a larger case study on less powerful mobile devices, such as cell phones; this would provide a more realistic measurement of enforcement overhead and give us more experience programming in LoPSiL.

It would also be interesting to investigate how to incorporate technologies, such as sophisticated static analyses [6] or policy structures [5], to simplify the task of specifying complex policies in LoPSiL. Such an investigation might lead to technologies for specifying complex location-dependent policies more conveniently as compositions of simpler subpolicy modules.

Finally, previous work has shown that policy-specification languages like LoPSiL, which allow monitoring mechanisms to store a complete trace of the application-program execution being monitored, enable specification and enforcement of all safety policies (such as the access-control policy in Figure 2) and some liveness policies (such as the social-networking policy in Figure 5) [18, 19]. In the future we would like to include location information, mobile applications, and mobile monitoring mechanisms in formal models of policy enforcement, in order to better understand the precise space of policies enforceable with LoPSiL.

References

1. Anisetti, M., Ardagna, C., Bellandi, V., Damiani, E.: Openambient: A Pervasive Access Control Architecture. In: Schmidt, A., Kreutzer, M., Accorsi, R. (eds.) Long-Term and Dynamical Aspects of Information Security: Emerging Trends in Information and Communication Security. Nova Science Publisher (2007)

2. Ardagna, C., Cremonini, M., Damiani, E., di Vimercati, S., Samarati, P.: Supporting Location-based Conditions in Access Control Policies. In: Symposium on Information, Computer and Communications Security. (2006)
3. The AspectJ Project, `http://www.eclipse.org/aspectj/`
4. Bauer, L., Ligatti, J., Walker, D.: Composing Expressive Run-time Security Policies. ACM Transactions on Software Engineering and Methodology. (To appear)
5. Bauer, L., Ligatti, J., Walker, D.: Composing Security Policies with Polymer. In: ACM Conference on Programming Language Design and Implementation. (2005)
6. Bauer, L., Ligatti, J., Walker, D.: Types and Effects for Non-interfering Program Monitors. In: Okada, M., Pierce, B., Scedrov, A., Tokuda, H., Yonezawa, A. (eds.) Software Security—Theories and Systems. Springer (2003)
7. Bhatti, R., Damiani, M., Bettis, D., Bertino, E.: Policy Mapper: Administering Location-based Access-control Policies. IEEE Internet Computing, 12(2), 38–45. (2008)
8. Byte Code Engineering Library, `http://jakarta.apache.org/bcel/`
9. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder Policy Specification Language. Lecture Notes in Computer Science, 1995, 18–39. Springer (2001)
10. Edjlali, G., Acharya, A., Chaudhary, V.: History-based Access Control for Mobile Code. In: ACM Conference on Computer and Communications Security. (1998)
11. Erlingsson, Ú., Schneider, F.: IRM Enforcement of Java Stack Inspection. In: IEEE Symposium on Security and Privacy. (2000)
12. Evans, D., Twyman, A.: Flexible Policy-directed Code Safety. In: IEEE Symposium on Security and Privacy. (1999)
13. eXtensible Access Control Markup Language (XACML) version 2.0, `http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf`
14. GPSLib4J v0.1, `http://gpslib4j.sourceforge.net/`
15. The Java ME Platform - the Most Ubiquitous Application Platform for Mobile Devices, `http://java.sun.com/javame/index.jsp`
16. Jeffery, C., Zhou, W., Templer, K., Brazell, M.: A Lightweight Architecture for Program Execution Monitoring. In: Program Analysis for Software Tools and Engineering (PASTE), pp. 67–74. ACM Press (1998)
17. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. An Overview of AspectJ. In: European Conference on Object-oriented Programming. Springer-Verlag (2001)
18. Ligatti, J., Bauer, L., Walker, D.: Enforcing Non-safety Security Policies with Program Monitors. In: 10th European Symposium on Research in Computer Security. (2005)
19. Ligatti, J., Bauer, L., Walker, D.: Run-time Enforcement of Nonsafety Policies. ACM Transactions on Information and System Security, 12(3), 1–41. (2009)
20. LoPSiL Implementation, `http://www.cse.usf.edu/~ligatti/projects/runtime/LoPSiL.zip`
21. Robinson, W.: Monitoring Software Requirements Using Instrumented Code. In: Proceedings of the 35th Annual Hawaii International Conference on System Sciences, p. 276.2. (2002)
22. Schmidt, A., Kuntze, N., Abendroth, J.: Trust for Location-based Authorisation. In: Wireless Communications and Networking Conference, pp. 3163–3168. (2008)