Modularizing Crosscutting Concerns in Software

Nalin Saigal
University of South Florida, nsaigal@mail.usf.edu

Follow this and additional works at: http://scholarcommons.usf.edu/etd
Part of the American Studies Commons, and the Computer Sciences Commons

Scholar Commons Citation
http://scholarcommons.usf.edu/etd/3327

This Dissertation is brought to you for free and open access by the Graduate School at Scholar Commons. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact scholarcommons@usf.edu.
Modularizing Crosscutting Concerns in Software

by

Nalin Saigal

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Jay Ligatti, Ph.D.
Adriana Iamnitchi, Ph.D.
Dewey Rundus, Ph.D.
Wilfrido Moreno, Ph.D.
Brendan Nagle, Ph.D.

Date of Approval:
March 22, 2011

Keywords: Software Engineering, Code Maintenance, Aspect-Oriented Programming, Security,
Policy-Specification Languages

Copyright © 2011, Nalin Saigal
DEDICATION

I dedicate my dissertation to my family and close friends. A special thanks goes out to my parents Sumesh and Suman Saigal, who have been extremely supportive, and, to my sister, Shalini, whose encouraging words have always kept me positive.
ACKNOWLEDGEMENTS

I would like to thank my advisor Jay Ligatti, for giving me the opportunity to work with him, and for being an incredible advisor. The duration of my doctoral degree under his guidance has made graduate school a memorable academic experience for me.

Most of the work presented in this dissertation is in collaboration with Jay Ligatti, Adriana Iamnitchi, and Joshua Finnis, and I would like to thank them for their inputs to this research.

I would also like thank the faculty of the Computer Science department at USF for teaching the various courses offered by the department. A big part of the learning experience at USF was through courses, and the faculty at USF made it an extremely enjoyable experience. Also, a special thanks goes out to my committee members, for being part of the group that encouraged this research.

Finally, I would like to acknowledge the National Science Foundation as my research was supported by them under grants CNS-0742736, CNS-0831785, and CNS-0716343.
# TABLE OF CONTENTS

LIST OF TABLES iii  
LIST OF FIGURES iv  
ABSTRACT v  

CHAPTER 1 INTRODUCTION 1  
1.1 Motivation 3  
1.2 Inline Visualization of Concerns 4  
1.2.1 Related Work 6  
1.2.1.1 Aspect-oriented Programming 6  
1.2.1.2 Concern Visualization Tools 7  
1.2.1.3 Programming Languages 9  
1.2.1.4 Feature-oriented Programming 10  
1.3 A Location-based Policy-specification Language 12  
1.3.1 Related Work 13  
1.3.1.1 General Policy-specification Languages 13  
1.3.1.2 Formal Languages 13  
1.3.1.3 Location-based Access-control Languages 14  
1.3.1.4 Android’s Built-in Security Mechanism 15  
1.4 Contributions 16  
1.5 Thesis Organization 17  

CHAPTER 2 AN IMPLEMENTATION OF IVCON 18  
2.1 User Interface 18  
2.1.1 Woven View 18  
2.1.2 Unwoven View 20  
2.1.3 Display of Multi-concern Code 24  
2.2 Implementation Details 25  
2.2.1 Data Structures 25  
2.2.2 Translation Algorithms 26  
2.2.2.1 Unweaving 26  
2.2.2.2 Weaving 27  
2.2.3 Third-party Libraries 27  

CHAPTER 3 MODULARIZING RUNTIME SECURITY POLICIES 29  
3.1 LoPSiL 30  
3.1.1 Core Linguistic Constructs 30  
3.1.2 Example LoPSiL Policy 33  

3.2 A LoPSiL Compiler

3.2.1 Compiler Architecture

CHAPTER 4 VALIDATION OF APPROACH

4.1 IVCon

4.1.1 Case Studies

4.1.1.1 Extending IVCon

4.1.1.2 Extending JHotDraw and Java Scientific Calculator

4.1.2 Experiential Observations

4.1.3 Performance Evaluation

4.2 LoPSiL

4.2.1 Case Studies

4.2.2 Experiential Observations

4.2.3 Performance Evaluation

CHAPTER 5 CONCLUSIONS

5.1 Discussion

5.2 Future Work

LIST OF REFERENCES
LIST OF TABLES

Table 1.1 Comparison of concern-manipulation tools. 7
Table 1.2 Comparison of policy-specification languages. 14
Table 4.1 Implementation effort for IVCon case study. 39
Table 4.2 Implementation effort for JHotDraw and Java Scientific Calculator case studies. 40
Table 4.3 Test-file characteristics. 42
Table 4.4 Performance opening and saving files. 43
Table 4.5 Performance assigning code to concerns. 43
Table 4.6 Performance editing concern colors and removing concerns. 44
Table 4.7 Performance weaving and unweaving code. 44
Table 4.8 Overhead incurred from using LoPSiL 50
LIST OF FIGURES

Figure 1.1 Sample code demonstrating the motivation behind token-level granularity. 6
Figure 2.1 IVCon’s woven view of the same code shown in Figure 2.2. 19
Figure 2.2 IVCon’s unwoven view of the same code shown in Figure 2.1. 20
Figure 2.3 Concern-module formatting in IVCon. 21
Figure 2.4 Unwoven view of the code in Figure 2.5. 22
Figure 2.5 Woven view of the code in Figure 2.4. 23
Figure 3.1 Simple LoPSiL Policy that allow all actions to execute 33
Figure 3.2 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. 34
Figure 3.3 Overview of the LoPSiL compiler. 35
Figure 4.1 LoPSiL policy preventing an application from reading GPS data outside of work hours. 46
Figure 4.2 Abbreviated LoPSiL policy requiring that navigational aid appear when the device’s current location deviates from its expected path. 46
Figure 4.3 Abbreviated LoPSiL policy requiring robot-control software to encrypt outgoing messages when the robot is outside a secure-region perimeter. 47
Figure 4.4 A location-dependent social-networking policy specified in LoPSiL. 48
Figure 5.1 IVCon’s newer interface for reduced concern assignment times. 55
ABSTRACT

Code modularization provides benefits throughout the software life cycle; however, the presence of crosscutting concerns (CCCs) in software hinders its complete modularization. Traditional modularization techniques work well under the assumption that code being modularized is functionally orthogonal to the rest of the code; as a result, software engineers try to separate code segments that are orthogonal in their functionality into distinct modules. However, in practice, software does not decompose neatly into modules with distinct, orthogonal functionality. In this thesis, we investigate the modularization of CCCs in software using two different techniques.

Firstly, we discuss IVCon, a GUI-based tool that provides a novel approach to the modularization of CCCs. We have designed IVCon to capture the multi-concern nature of code. IVCon enables users to create, examine, and modify their code in two different views, the woven view and the unwoven view. The woven view displays program code in colors that indicate which CCCs various code segments implement, while the unwoven view displays code in two panels, one showing the core of the program and the other showing all the code implementing each concern in an isolated module. IVCon aims to provide an easy-to-use interface for conveniently creating, examining, and modifying code in, and translating between, the woven and unwoven views.

Secondly, we discuss LoPSiL, which is a location-based policy-specification language. LoPSiL is Turing-complete and provides users with language constructs that enable them to manipulate location information; hence, LoPSiL can be used to specify and enforce generic policies that might involve location-based constraints. We have implemented a LoPSiL compiler using AspectJ, and we observe and discuss how the use of traditional units of modularization—aspects in this case—help modularize functionally orthogonal CCCs such as security and auditing.
CHAPTER 1

INTRODUCTION

In accordance with good software-development practices, software engineers typically invest time devising the structure of a project before they can start working on the project’s implementation. During the design phase, software engineers attempt to modularize their code; that is, they try to organize their code into distinct software modules. This practice in turn provides multiple benefits to programmers:

- It makes code easier to write. Because programmers have invested time organizing their code into functionally orthogonal modules, different programmers can work on different modules irrespective of the implementation of other modules.

- It makes code easier to understand. By providing APIs and schematic diagrams in which modules and interfaces between modules are clearly depicted, programmers can quickly communicate the structure of their software to other programmers.

- It makes code easier to maintain. Again, because of the design process, programmers know in which file(s) to find relevant code segments when they want to add or edit features in their software.

Ideally, to reap the maximum benefits of modularization, software engineers try to separate code segments that are orthogonal in their functionality into distinct modules. However, in practice, software does not decompose neatly into modules with distinct, orthogonal functionality. For example, code that displays a popup window notifying users about a failed login attempt may be present in a login module, while (partially) implementing various other functional concerns such as security, GUI, and authentication; it may be equally reasonable for the window-popup code to be located in a security, GUI, or authentication module, and at various times it may be more
convenient to write, view, or edit the window-popup code in the context of these other modules. Tarr et al. call this problem the “tyranny of the dominant decomposition” [1]. Although it is useful to modularize the same code segment in various ways throughout the software life cycle, current programming paradigms only allow modularization in fixed and limited ways (e.g., into functions, classes, or aspects).

While one module may implement multiple software concerns (i.e., behaviors required of the software, which are implemented in source code), it is, conversely, common for implementations of concerns to be scattered throughout multiple modules. For example, code implementing a security concern may be scattered throughout login, logout, and network-socket modules. Thus, code segments implementing a concern may crosscut through implementations of other concerns; such code segments implement crosscutting concerns (CCCs). Modularizing a CCC involves collecting and displaying in one place all the scattered code implementing that CCC. Isolating concern code in this way benefits programmers because it relieves them from having to browse through the whole program to find, study, or update a single software concern.

A common example of a CCC is security. Code implementing security often tends to get scattered throughout programs in the form of security checks (e.g., checking array bounds, checking access permissions to a file etc.), and because security is frequently functionally orthogonal to the rest of the program’s functionality, it makes sense to isolate code implementing security into a module that is disjoint from the implementation of the core software.

To isolate security implementations from the core software implementation, security engineers use policy-specification languages. Policy-specification languages such as PoET/PSLang [2], Naccio [3], Polymer [4, 5] etc. are domain-specific programming languages intended to simplify the tasks of specifying and enforcing sound security policies on untrusted (i.e., potentially insecure) software. Modularizing security implementations using policy-specification languages implies that code implementing security in an application, which would otherwise be scattered throughout the application, is now present in one module (file). So, to analyze and/or modify such an application’s security implementation would entail locating and/or modifying only one file, thereby reducing the effort when compared to a system in which a policy-specification language was not being used, in which case, a security engineer would have to locate every instance of a security check throughout
the application (encompassing several files) and make changes in multiple files. We discuss the modularization of runtime security policies further in Chapter 3.

1.1 Motivation

In his book on refactoring techniques, Fowler presents and discusses in detail, a list of 72 techniques for reorganizing code so that code becomes easier to comprehend [6, 7]. While all these techniques help programmers improve the structure and readability of their code, some of them give rise to a new problem. By using refactoring techniques such as Extract into Method, Extract into Class etc. programmers essentially relocate the implementations of their program’s different functionalities into newer files. It would make sense to do so, if the code being extracted into a newer class or newer method were completely orthogonal in functionality to the code surrounding it (such as code implementing security or logging in an application); however, as mentioned earlier code often does not decompose into orthogonally-defined functionalities. As a result, traditional refactoring techniques add to the problem of code-scattering that we discussed earlier by scattering implementations of functionalities across multiple files. This, in turn makes it more difficult for programmers to modify existing applications.

We observed this problem while trying to add a feature to JHotDraw [8], which is a Java-based framework that for developing drawing editors for designing structured graphics. JHotDraw is implemented in approximately 33,000 lines of code, and its implementation is well-structured and easy to comprehend. Most JHotDraw files are small in size, which indicates that JHotDraw developers have used good software development practices by (recursively) modularizing code wherever possible. However, while modifying its source code, we found that by following conventionally-good modularization practices, JHotDraw developers have indeed scattered code implementing a single functionality across multiple files. As a result, we spent a significant amount of time analyzing JHotDraw’s code to locate different files in which the functionalities of our interest were present. Thus, while current modularization practices improve code structure, they can also impede the process of software maintenance.
The problems described above stem from the following two properties of code found in software systems:

- A particular code segment implements multiple functionalities.
- A particular functionality is implemented by code that is scattered across multiple methods and files.

Thus, if we think of concerns and code segments as mathematical sets, we can see that typically in software, there exists a many-to-many relationship between concerns and code segments implementing those concerns; that is, a concern can be implemented by multiple non-contiguous code segments, and a code segment can implement multiple concerns. However, traditional modularization techniques restrict users into defining one-to-many relationships between concerns and code; that is, a concern can be implemented by multiple non-contiguous code segments.

1.2 Inline Visualization of Concerns

To target the behavior of code that traditional modularization techniques do not capture, current modularization techniques need to be augmented by using newer techniques during software development. Correspondingly, we have implemented, in Java, an IDE called IVCon that aims to target these problems by providing users with multiple views of their code so that users can:

- View their complete code, as it will be at runtime. This enables users to debug complete programs; if complete programs are not available to users they could be overlooking an aspect of their programs that might be causing an undesired effect at runtime.

- View their code in the absence of CCCs. This enables users to analyze their code when CCCs are completely orthogonal to the functionality of the core program.

- View CCCs in isolation. In the presence of functionally orthogonal concerns, this enables different programmers to work on different modules, independent of the implementation of other modules (one of the benefits of traditional modularization techniques).

- View and readily identify code that implements multiple concerns. This enables users to understand the different functionalities of their program that will get affected when a code
segment $S$ is modified (all the concerns that $S$ implements get affected when a user modifies $S$).

- View concerns whose implementations are scattered across multiple files readily. This ensures that users do not have to spend time trying to find implementations of particular functionalities.

IVCon permits many-to-many relationships between concerns and code. That is, users can assign scattered code segments to the same concern and can assign a single code segment to multiple, overlapping concerns. We call code that has been assigned to multiple concerns *multi-concern code*. Moreover, IVCon enforces token-level granularity in concern assignments: code assigned to a concern must begin at the beginning of a source-language token and end at the end of a source-language token.

IVCon enforces token-level granularity because tokens are the smallest, atomic units with meaning in a programming language. Allowing finer granularity in concern assignment (e.g., character-level granularity) would be inappropriate because tokens are the core semantic units of programming languages and of concerns implemented in those languages. On the other hand, requiring coarser granularity in concern assignment (e.g., line-level granularity) would be inappropriate as well. Consider the code in Figure 1.1. Token-level granularity enables assignment of just the `System.currentTimeMillis()` code segment to a `SystemCall` concern, while coarser concern-assignment granularities, such as line- or statement-level granularity, lack the precision needed for such a concern assignment. With token-level granularity, a user could even assign just the method name `currentTimeMillis` to the `SystemCall` concern. At the same time, token-level granularity prevents unreasonable concern assignments possible with finer (e.g., character-level) granularities, such as assigning just the ‘i’ in `JOptionPane` to its own concern; this would be unreasonable because if ‘i’ implements a concern $C$, then the rest of the `JOptionPane` token must implement $C$ as well (the atomic `JOptionPane` token has meaning in the programming language, while the ‘i’ alone does not). Thus enforcing token-level granularity enables fine-grained, yet meaningful concern assignments.
JOptionPane.showMessageDialog(mainWindow.frame, "Welcome " + userName + ".
Current time is " + format(System.currentTimeMillis()),
"Welcome", JOptionPane.INFORMATION_MESSAGE);

Figure 1.1. Sample code demonstrating the motivation behind token-level granularity.

1.2.1 Related Work

There are many projects related to IVCon, which we next discuss in detail. Table 1.1 summarizes the discussion.

1.2.1.1 Aspect-oriented Programming

A closely related body of research is Aspect Oriented Programming (AOP) [9]; like IVCon, AOP eases the specification and manipulation of CCCs in software. AOP languages (AOPLs) such as AspectJ [10] and AspectC [11] define a new unit of modularization, the aspect, which is a combination of advice (code that implements a CCC) and joinpoints (points in a program’s control flow where advice gets executed). A complete aspect-oriented program consists of a core program and aspects, and AOPL compilers typically weave advice from user-defined aspects into the core program at the joinpoints specified by those aspects. Roughly speaking, then, IVCon’s unwoven view corresponds to an aspect-oriented view of a program, as code implementing CCCs appear in isolated modules. However, unlike standard AOP tools, IVCon:

- Allows code to be written, examined, and edited in both woven and unwoven views.
- Allows multi-concern code. For example, multi-concern code is impossible in AspectJ because a single code segment $S$ in the woven program cannot appear in multiple AspectJ advices (or in both the program core and some advice)—if it did then $S$ would have to appear multiple times (not once) in the woven program.
- Enforces token-level granularity in concern code.
- Uses a novel interface to aid concern assignment, visualization, weaving, and unweaving.

On the other hand, IVCon is able to provide some of these features only because it disallows joinpoints (called regions in IVCon) from being specified indirectly (i.e., as pointcuts), which AOPLs
typically do allow. IVCon also lacks constructs for encapsulating or interfacing with modularized code, which AOPLs do provide (this limitation is discussed further in Section 5.1).

1.2.1.2 Concern Visualization Tools

Turning our attention to specific AOP tools related to IVCon, Aspect-jEdit [12] plugs into the jEdit [13] text editor and, like IVCon, allows users to view and edit concern code in the context of the core program. Also like IVCon, Aspect-jEdit users assign code to concerns by highlighting code and explicitly assigning it to the concern it implements, and users can assign syntactically different code segments to the same concern. Each aspect (i.e., concern) in Aspect-jEdit is associated with a color, and on assigning code to an aspect, that code’s background color changes to match its aspect color. Aspect-jEdit users can hide one or more aspects and view aspects in isolation, but Aspect-jEdit does not support multi-concern code (which simplifies its interface and text-manipulation algorithms).
Aspect-jEdit displays syntactically equal advice multiple times in an aspect, as opposed to IVCon’s use of subconcerns (described in Section 2.1.2); consequently, Aspect-jEdit users cannot modify all identical concern code at once in a central concern module.

AspectBrowser [14, 15, 16], which is built on ideas from Seesoft [17], lets users specify concerns in the form of regular expressions over characters in the source code. AspectBrowser displays a high-level program map in which colors indicate which lines contain which CCCs. When a program line contains more than one CCC, AspectBrowser colors the corresponding map line red. Also, AspectBrowser users can zoom within the map to obtain a more detailed view and can click on a colored line to view that line’s CCC and its context (i.e., the core code surrounding the CCC). IVCon does not build a high-level program map, though it does graphically identify all code regions implementing each CCC (using colors, flags, tooltips, and a concern-at-current-position panel in the woven view, as well as colors, flags, and explicit region names in the unwoven view). Tools like grep can also be used to identify and display in a single view all the code matching regular expressions [18].

The Visualiser [19] is an Eclipse [20] plugin that helps AspectJ programmers visualize joinpoints in their programs using a high-level program map. After assigning colors to existing aspects, the Visualiser performs static analysis to generate a program map similar to that of AspectBrowser. In the Visualiser map, colored lines represent the locations of joinpoints, and if multiple aspects share a joinpoint then the Visualiser splits that line into the colors of those aspects. Concerns that are implemented in AspectJ and used with the Visualiser can make use of AspectJ’s rich joinpoint language. Due to its reliance on AspectJ, though, the Visualiser inherits AspectJ’s limitations of statement-level granularity in concern assignments and one-to-many relationships between concerns and code.

FEAT [21] and SoQueT [22] are tools available as plugins to Eclipse that help users modularize CCCs in similar ways. In both these tools, users search their code (using regular-expression queries) to retrieve program elements (e.g., classes, data members, methods etc.) that implement concerns. Users can encapsulate these program elements along with other information into concern modules. FEAT and SoQueT differ in the concern modules that they use and the information encapsulated in those concern modules. FEAT uses concern graphs, which store program elements and relations between different program elements, whereas SoQueT uses crosscutting concern sorts, which docu-
ment program elements and the nature of the crosscutting concern. Both tools allow many-to-many relationships between concerns and code and allow users to navigate code implementing concerns in a woven view. Although the tools isolate concerns at a high level, they do not isolate all the code implementing each concern in a single module, so these tools do not support viewing or editing code in a fully unwoven view.

1.2.1.3 Programming Languages

The literate-programming tool **WEB** [23], like IVCon, allows users to document which concerns code implements. Actually, in literate programming, documentation precedes code; users document programs and can write prose abbreviations for code segments before expanding those abbreviations by specifying their code implementations. Hence, **WEB** abbreviations could be viewed as concern names, to which code implementations get assigned. Abbreviations can be nested and scattered through other abbreviations, so in this view literate programming supports many-to-many relationships between concerns and code. A **WEB** application consists of all the documentation and code for that program; when **WEB** compiles the application it produces two sorts of files: human-readable files documenting the program (with each “concern” isolated, as in IVCon’s unwoven view) and source-code/executable files (as in IVCon’s woven view). These two sorts of files present alternate views of the program, but users can not edit one view and then see those changes reflected in the other. Interestingly, the documentation produced by **WEB** cross-references abbreviations, which serves a purpose similar to flags in IVCon (described in Section 2.1.3).

The C4 toolkit provides an aspect-oriented approach to system-level programming [24]. As with IVCon, C4 users can examine programs in, and translate between, two code views. In C4, these views are called the AspectC view (in which users define aspects in AspectC) and the C4 view (in which users view advice inlined into the program code); these are analogous to IVCon’s unwoven and woven views. Similar to the Visualiser, C4 inherits AspectC’s lack of support for multi-concern code and its statement-level granularity in concern code.

Hyper/J is a Java-based AOPL that introduces the concept of a hyperspace, an imaginary space consisting of multiple dimensions of concerns [25]. Each dimension, or axis, in the hyperspace groups concerns, while each coordinate on an axis corresponds to a single concern. Hence, a code segment’s position in the hyperspace indicates which concerns it implements (one concern per axis.
in the hyperspace). To build software with Hyper/J, users create a set of text files that specify the set of features to include in the program. Concerns in Hyper/J are defined coarsely at the granularity of declarations (e.g., methods, functions, variables, and classes). If all declarations in a program are assigned to at least one concern, then a Hyper/J user can view any concern in that program in isolation, but doing so involves modifying a textual concern-mapping file to specify that only $c$ should be displayed (i.e., included in the program).

### 1.2.1.4 Feature-oriented Programming

CIDE (Colored Integrated Development Environment) [26, 27] is a tool for feature-oriented programming that was developed concurrently with IVCon. CIDE has many similarities with both Hyper/J and IVCon: as with Hyper/J, CIDE users can build programs by selecting the sets of features (which are analogous to CCCs in Hyper/J and IVCon) to include in those programs; as with IVCon, CIDE users can highlight code to assign it to features being implemented, can define many-to-many relationships between features and code, can assign colors to features, and can view code colored to reflect the features being implemented. However, there are at least four high-level differences between CIDE and IVCon:

- CIDE displays code assigned to a feature $f$ as black text on the background color of $f$. For code that implements multiple features, CIDE displays a background color equal to the chromatic blending of the colors of all features being implemented. This design relies on a user’s ability to decompose any displayed background color $b$ into the feature colors that combined to produce $b$, a challenging task when many feature colors exist (some of which may even be similar to combinations of other feature colors). IVCon attempts to avoid this problem by displaying all multi-concern code in a distinctive but uniform manner; users determine exactly which concerns multi-concern code implements by looking in either a separate panel or a tooltip.

- The granularity of feature assignment in CIDE is coarser than the granularity of concern assignment in IVCon. CIDE allows users to assign concerns at the grammatical level of nodes in abstract syntax trees (ASTs), rather than at the lexical level of tokens. Every AST node is a valid sequence of tokens, but there are an infinite number of token sequences that are
not AST nodes (e.g., standard grammars would not have AST nodes for token sequences like methodName(, methodName(paramExpr1, methodName(paramExpr1, paramExpr2, etc). Token-level granularity is significantly more expressive than AST-node granularity.

• Because CIDE requires that any set of features can be composed to create a legal program, CIDE users can only assign code segments to features when those segments are optional according to the language’s syntax. IVCon lacks the ability to create a valid program by selecting an arbitrary set of concerns to include in the program; therefore, IVCon has no need for restricting concern code to syntactically optional segments. For example, an IVCon user could assign the constant in the statement pi=3.14 to a constants concern (to enable viewing and editing all the program’s constants in a single module in the unwoven view), but such an assignment is impossible in CIDE because the constant is not syntactically optional.

• By specifying a program to contain exactly one feature, CIDE users can view that one feature isolated from the others (but not isolated from the core program code). However, as with Aspect-jEdit, CIDE displays syntactically equal code implementing the same feature multiple times, so CIDE users cannot modify all identical feature code at once in a central module.

Most of these differences arise naturally from the distinct objectives of CIDE and IVCon: CIDE (and related technologies such as Software Plans [28]) focuses on constructing software as a set of features, while IVCon focuses on creating, viewing, and modifying CCCs in isolation (as well as in the regular, woven view of the code).

Finally, we note that although IVCon is closely related to AOPLs due to its emphasis on modularizing and refactoring concern code, IVCon could not be considered an AOPL tool according to Filman and Friedman’s definition of AOPLs [29]. Filman and Friedman specify two necessary properties of AOPLs, obliviousness and quantification. IVCon’s woven view is not oblivious because it requires a programmer to document concerns directly in the body of the code. Also, IVCon does not satisfy the quantification condition because it does not let users define joinpoints (i.e., IVCon regions) as conditions on the program’s control flow. Allowing IVCon users to define joinpoints as conditions on control flow could lead to ambiguous order of execution when weaving concern code into the core; disambiguating concern-code execution ordering would require some mechanism for specifying concern-code precedence, which would complicate IVCon’s design. Nonetheless, IVCon’s
lack of obliviousness and quantification does not necessarily prevent it from being used as a basis for standard AOP technologies, given that previous work has shown how to build an (oblivious and quantified) AOPL on top of an unoblivious and unquantified aspect language [30].

1.3 A Location-based Policy-specification Language

As mentioned in Section 1.1, traditional modularization techniques work well under the assumption that code being modularized implements functionally orthogonal crosscutting concerns such as security, auditing etc. To examine this claim further, we have designed and implemented LoPSiL, which is a policy-specification language for mobile devices [31, 32]. Policy-specification languages are intended to simplify the process of specifying and implementing policies and typically they operate by statically rewriting an application so that the modified application satisfies input policies at runtime.

Ensuring a secure environment on mobile devices has gained a lot of attention with the release of open mobile-computing platforms such as Android, iPhone, and Windows Phone 7. Before their release, phone manufacturers were mostly solely responsible for developing software for phones. However, with the advent of these platforms, a significantly larger set of people are now developing software for phones. Coupled with the fact that phones now have significantly increased computing powers, users need to treat and secure their phones like they would their computers. However, most users tend to ignore this, which in turn, makes it easier for a malicious user to gain unauthorized access to a users phone and sensitive information stored in the phone. Additionally, many programmers developing applications for these open-mobile platforms might not test their applications comprehensively enough to be able to guarantee no (or at least minimal) security holes. Thus, even when a software developers intent may be good, it does not necessarily translate into secure software.

As a result, we have designed and implemented a policy-specification language called LoPSiL that:

- Allows users to specify generic policies: Because our language is Turing-complete, programmers can specify policies that are a strict superset of access-control (safety) policies.
Enables users to conveniently enforce location-based constraints in their software: To ensure that LoPSiL caters to mobile platforms, we have built-in location constructs and included libraries in LoPSiL that enable manipulation of location information (e.g., computing containment within a region, determining a user’s velocity etc.), so that users can more conveniently express location-based conditions in policies.

At its core, LoPSiL uses aspects for policies (although programmers specify policies in the form of Java class files) and AspectJ’s compiler as its rewriter.

1.3.1 Related Work

There are several policy-specification languages and policy frameworks that are related to LoPSiL, which we next discuss. Table 1.2 summarizes this discussion.

1.3.1.1 General Policy-specification Languages

A rich variety of policy-specification languages and systems has been implemented, which enable users to centrally specify security and privacy policies to be enforced on untrusted software at runtime. Ponder [33], XACML [34], PoET/PSLang [2], Naccio [3], Polymer [4], and Deeds [35] 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.

1.3.1.2 Formal Languages

SpatialP is a Turing-complete language in which policies can make decisions about a program’s actions based on the location of the user [36]. 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 4.2 and 4.4, and discussed in Section 4.2.1). Also, as far as we are aware, SpatialP does not have an implementation.
### Table 1.2. Comparison of policy-specification languages.

<table>
<thead>
<tr>
<th>Language</th>
<th>Provides location constructs</th>
<th>Turing complete</th>
<th>Allows flexible manipulation of location information</th>
<th>Provides policy composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ponder</td>
<td>-</td>
<td>√</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>XACML</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PoET/PSLang</td>
<td>-</td>
<td>√</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Naccio</td>
<td>-</td>
<td>√</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Polymer</td>
<td>-</td>
<td>√</td>
<td>-</td>
<td>√</td>
</tr>
<tr>
<td>Deeds</td>
<td>-</td>
<td>√</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SpatialP</td>
<td>√</td>
<td>√</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>OpenAmbient</td>
<td>√</td>
<td>-</td>
<td>√</td>
<td>-</td>
</tr>
<tr>
<td>Geo-RBAC</td>
<td>√</td>
<td>-</td>
<td>√</td>
<td>-</td>
</tr>
<tr>
<td>Android’s built-in Security</td>
<td>√</td>
<td>-</td>
<td>√</td>
<td>-</td>
</tr>
<tr>
<td>LoPSiL</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>-</td>
</tr>
</tbody>
</table>

### 1.3.1.3 Location-based Access-control Languages

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 [37, 38]. 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 [39]. 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 [40]. 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 4.2–4.4) could not be specified with Geo-RBAC.

Ray and Kumar describe a formal, Turing-incomplete model that extends a MAC system with location primitives [41]. 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. However, because their model is an access-control (safety policy) model, users cannot specify policies such as DeviationFromPath and SocialNetworking.

1.3.1.4 Android’s Built-in Security Mechanism

The Android operating system enforces mandatory access-control in Android applications [42, 43]. Users specify, in an XML manifest file, permissions that mediate both incoming and outgoing requests pertaining to the various components of their application (i.e., users have to specify resources that their application will access, and services that can access the users’ application). Users can also specify policies that run arbitrary code to decide whether permission to a resource should be granted or not; this is achieved by modifying a method called checkPermission(). Also, because Android provides users with primitives to manipulate location information, users can utilize these primitives in the checkPermission() method to specify location-based access-control policies. However, the Android security framework does not enable users to specify and enforce runtime policies, so the existing framework in Android can not be used to intercept user actions and decide whether or not to allow those actions to be executed. As a result, Android’s built-in enforcement system is limited to access-control policies and therefore similarly to Geo-RBAC and the model of Ray and Kumar, cannot enforce more general runtime policies such as DeviationFromPath and SocialNetworking.
1.4 Contributions

The first part of this thesis discusses the modularization of crosscutting concerns using IVCon, an IDE that enabling users to flexibly assign concerns in their code by allowing users to explicitly assign different parts of their code to the concern(s) that they implement. IVCon differs from existing concern-management tools by providing a combination of (1) translations between woven and unwoven views (2) many-to-many relationships between concerns and code, (3) isolating concern code in the unwoven view, (4) centrally modifying identical concern code, and (5) token-level granularity in concern assignment.

Thereafter, we discuss the modularization of runtime security policies using LoPSiL, a policy-specification language that uses aspects to modularize security implementations. As far as we are aware, LoPSiL is the first Turing-complete language that provides language constructs that enable users to manipulate location information conveniently.

This thesis derives from our earlier work [44, 31, 45, 46, 47, 48, 32], but ties it together to present a complete treatment to the modularization of CCCs in software. More specifically, this thesis extends previous work by answering the following questions:

- How a dual view of software and the ability for programmers to define multi-concern code aids software comprehension, and how these features help during the process of software maintenance?

- Whether or not, the benefits achieved by using dual views of software and multi-concern code can be achieved within reasonable overheads, in terms of a user’s effort?

- How suitable are traditional modularization constructs (i.e., methods, classes, and aspects) for modularizing code that implements multiple concerns? Additionally, how suitable are the same constructs for modularizing functionally orthogonal concern code?

- What kind of language constructs are useful for policies that reason about a program’s execution based on a system’s location? More specifically, does our language (LoPSiL) provide useful constructs for specifying such policies?
• What kind of policies would be useful to specify and implement in a language that enables users to specify location-based policies? Furthermore, how convenient is it to specify those policies in LoPSiL?

1.5 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 describes the user interface and implementation details of our prototype of IVCon, Chapter 3 discusses the modularization of runtime policies along with implementation details of our proof-of-concept LoPSiL compiler, Chapter 4 discusses the validity of our approach by outlining our experiences working with IVCon and LoPSiL, and an analysis of their performance overheads, and Chapter 5 concludes.
CHAPTER 2

AN IMPLEMENTATION OF IVCON

This chapter describes the user interface and implementation details of our prototype implementation of IVCon.

2.1 User Interface

IVCon displays code in two different but equivalent forms: the woven view (Figure 2.1) and the unwoven view (Figure 2.2). Users can translate their code between the two views simply by selecting the weave or unwave menu options, or by pressing \texttt{<ctrl-w> or <ctrl-u>.

2.1.1 Woven View

In the woven view, shown in Figure 2.1, users can write code as they normally would in a standard text editor or development environment. In addition, users can define concerns, associate a color with each concern, and highlight and explicitly assign code segments to concerns (by right-clicking highlighted code and selecting the concern to which to assign it). Upon assigning a code segment to a concern, IVCon recolors the assigned code to match the concern it implements. IVCon displays code assigned to multiple concerns in white text on a dark background, though users are free to change this multi-concern background color.

By highlighting a contiguous code segment and assigning it to a concern, a user defines a region in the code. Every region starts at the beginning of code assigned to a concern and extends as far as the code does that implements that concern. Multiple concerns can share the same region if code implementing those concerns begins and ends at exactly the same positions in the program file. Although IVCon requires a user-specified name for every unique region a user defines, it always provides a default name for regions based on the name of the concern to which the region is being assigned. Default region names provided by IVCon are of the format \texttt{region_concernname_i},
where \( i \) is the number of regions that have been previously defined for the concern `concernname`. For example, if a user has defined three regions for the `security` concern in a file, the default name that IVCon provides for the next region that gets assigned to the security concern would be `region_security_3`. Also, if a previously defined region gets assigned to a new concern, IVCon simply reuses the existing region name. Specifying names for regions helps users understand where subconcerns are implemented, as discussed in Section 2.1.2.

Figure 2.1 shows the woven-view window divided into three panels: `concerns legend`, `woven body`, and `concerns at current position`.

- The concerns-legend panel lists all the user-defined concerns in the current file. IVCon displays the name of each concern in the color associated with that concern.

- The woven-body panel contains user code displayed in colors that indicate concern assignments.

- The concerns-at-current-position panel lists the concern(s) implemented by the code at the current cursor position.

Apart from creating and modifying code, defining concerns, and assigning code to concerns, users can edit concern names, edit concern colors, remove concerns, de-assign code from concerns,
change the multi-concern background, rename regions, and open, save, and close files in the woven view.

2.1.2 Unwoven View

The unwoven view, shown in Figure 2.2, displays the program core and each concern in isolation. The unwoven-view window is the same as the woven-view window, except that the unwoven-view divides the woven-body panel into two subpanels: unwoven body and unwoven concerns.

- The unwoven-body panel displays the core of the program. Code that has been assigned to one or more concerns is extracted (into the unwoven-concerns panel) and replaced by holes (□) of the same color as the extracted code. Every contiguous code segment that has been assigned
Figure 2.3. Concern-module formatting in IVCon.

to the same set of concerns gets replaced by one hole (explained further in Section 2.2.2.1). Thus, holes in the unwoven-body panel indicate extracted concern code.

- The unwoven-concerns panel displays in isolation each of the program’s concerns (as extracted from the unwoven body). Each concern is divided into subconcerns, which are syntactically different code segments assigned to the same concern. IVCon displays subconcerns in two parts: a list of the regions in which the subconcern appears and then the subconcern code itself. As an example, readers can look at Figure 2.3 to see concern modules for security and audit concerns for the example code from Figure 2.1. On clicking any region name in the unwoven-concerns panel, IVCon automatically focuses the unwoven-body panel to show that region’s location in the context of the program core.
Unwoven Body:

```java
if (buffer.getSize() > 2)
    buffer.truncate(2);
if (getTimeElapsed() > 2)
    JOptionPane.showMessageDialog(frame,
        "Request timed out","Error",
        JOptionPane.ERROR_MESSAGE);
```

Unwoven Concerns:

```java
concern constant
{
    subconcern @ max_buffer_size_0
        @ max_buffer_size_1
            \(\geq\)
            512
            \(<\)
    subconcern @ timeout_ms_0
        \(\geq\)
            2000
            \(<\)
}
```

Figure 2.4. Unwoven view of the code in Figure 2.5.

Code for the security concern in Figure 2.3 indicates the presence of two subconcerns (at regions `before_protected_read` and `after_protected_read`). The code segments beginning with `if (checkCredentials())` and `accessLog.append` implement those subconcerns. Similarly, code for the audit concern indicates the presence of three subconcerns (at regions `file_read_granted`, `file_read_denied`, and `after_protected_read`). The unwoven-concerns panel may also contain constructs called `flags` (e.g., `security` and `security`), which convey information about concern assignment in multi-concern code segments. Section 2.1.3 provides additional explanation of IVCon flags.

Figure 2.3 also demonstrates the usefulness of having descriptive, user-specified names for regions. Descriptive region names help software engineers quickly understand where subconcern code exists in relation to the rest of the program logic. Nonetheless, if a region name provides insufficient contextual information, the user can always click on the name to see that region’s context in the unwoven-body panel.
Woven Body:

```java
if (buffer.getSize() > 512)
    buffer.truncate(512);
if (getTimeElapsed() > 2000)
    JOptionPane.showMessageDialog(frame,
        "Request timed out","Error",
        JOptionPane.ERROR_MESSAGE);
```

Figure 2.5. Woven view of the code in Figure 2.4.

As another example, let us consider Figure 2.4, which shows how IVCon groups into one sub-concern all syntactically equal code assigned to the same concern. Figure 2.5 contains the woven view of the same program. In the woven view, the user has defined a concern named constant and has assigned the two constants 512 and 2000 to that concern (IVCon’s token-level granularity in concern assignment enables such operations). Normally, programmers using standard software-development tools would define these values in memory declared immutable and would always refer to the constants’ values with variables like MAXBUFFERSIZE and TIMEOUTMS. This technique enables the programmers to make a global change to a constant value by modifying just one central definition. However, this benefit comes at the price of not being able to immediately see the values of constants in the source code. In contrast, IVCon’s dual woven and unwoven views provide both benefits: users can update constant values centrally (in the constant concern of the unwoven-concerns panel) and can view the constant values directly in the source code (in the woven-body panel). Actually, IVCon provides the added benefit that users can see (in the unwoven-concerns panel) a region-name reference to every use of every constant and can click any of those region names to see the context of that use in the unwoven-body panel. Of course, this example is just a special case of the general use of IVCon to provide both a global (i.e., woven) view of the code and a concern-specific (i.e., unwoven) view of the same code.

IVCon’s unwoven view allows users to create and edit core-program code (in the unwoven-body panel) and concern code (in the unwoven-concerns panel). To avoid ambiguity in the weaving algorithm (described in Section 2.2.2.2), IVCon does not allow users to delete holes in the unwoven-body panel or to edit non-concern code (e.g., concern and region names) in the unwoven-concerns panel.
2.1.3 Display of Multi-concern Code

The woven view displays multi-concern code in white text over the multi-concern background, while the concerns-at-current-position panel indicates which concern(s) the code at the current cursor position implements. Similarly, the unwoven view displays multi-concern code in the unwoven-body panel as a hole colored white over the multi-concern background, while the concerns-at-current-position panel continues to indicate which concern(s) the hole at the current cursor position implements.

In addition, the unwoven-concerns panel uses flags to convey information about the concerns associated with multi-concern code (as mentioned in Section 2.1.2). Flags serve as a quick reference for visualizing where overlapping concerns begin and end. To illustrate the use of flags, consider the code in Figure 2.2 that implements the security concern at region before.protected_read. In the woven view (Figure 2.1), we assigned this code segment to the security concern, and we assigned the two nested statements beginning with accessLog.append to the audit concern. As a result, the unwoven-concerns panel in Figure 2.2 displays green flags (audit) and red flags (audit) to indicate the nesting of audit-concern code within the security-concern code.

Green and red flags within the unwoven-concerns panel indicate the beginning and ending of overlapping concerns. Green and red flags do not always appear together within a subconcern; depending on the overlap between concern regions, there may be a green flag only, a red flag only, both green and red flags, or no flags at all within subconcern code. Also, multiple red and green flags of the same concern, or multiple flags of various concerns, may be present within a subconcern.

The syntax of code in IVCon’s unwoven-concerns panel includes flags, so two subconcerns are syntactically equal if and only if the text of those two subconcerns—including flags within the subconcerns—is the same. Thus, the subconcern accessLog.append(‘‘File read complete.’’) is not syntactically equal to security|accessLog.append(‘‘File read complete.’’)|security. This distinction matters because, as described in Section 2.1.2, syntactically equal subconcerns are grouped together in the unwoven-concerns panel. If IVCon did not consider flags when testing for concern-code equality, it would group both accessLog.append(‘‘File read complete.’’) and security|accessLog. append(‘‘File read complete.’’)|security into a single subconcern, but the presence or absence of security flags in such a subconcern would
be confusing because one of the instances of this subconcern has a nested security region, while the other does not.

2.2 Implementation Details

We have implemented a prototype of IVCon in Java; the source code is online [49]. The core IVCon application consists of 7961 lines of code (18 classes in 15 files), of which 7788 implement the GUI and 173 implement backend data structures. IVCon also uses several third-party libraries (e.g., clipboard and lexical-analysis libraries) for auxiliary functions.

2.2.1 Data Structures

IVCon maintains three key data structures.

- A regionMap is a hash table that maps a numerical region identifier (needed for the RTree implementation described below) to that region’s user-visible name, beginning and ending positions for the region, and a list of the concerns to which the region has been assigned.

- A concernMap is a hash table that maps a unique concern name to that concern’s display color and a list of the regions assigned to that concern.

- A regionTree is an RTree, a dynamic structure for storing data about potentially overlapping regions in space [50, 51]. When queried about a particular region r, an RTree can efficiently return the set of stored regions that overlap r. RTrees are ideal data structures for IVCon because they enable efficient querying to determine which regions are defined at a given character position in the source file (e.g., at the position of the cursor or within a newly defined region). Once IVCon determines which regions are present at a given position, it can use the regionMap to look up all the concerns assigned to those regions. IVCon uses this look-up operation while weaving and unweaving code, and while displaying concerns in the concerns-at-current-position panel.

Together these structures enable reasonable performance in all of IVCon’s core operations.
2.2.2 Translation Algorithms

Given a regionMap, concernMap, and regionTree, it is generally straightforward to implement the translations between woven and unwoven views.

2.2.2.1 Unweaving

Unweaving is the process of translating a woven-view program into an equivalent unwoven-view program.

Unweaving in IVCon begins with lexical analysis to enforce token-level granularity in concern assignment. After lexical analysis, IVCon starts with the woven code in the unwoven-body panel and iterates through all concerns in the concernMap, extracting the code in all of that concern’s regions from the unwoven-body panel to the unwoven-concerns panel. Extracted concern code gets replaced with holes (□) of the appropriate color in the unwoven-body panel. During the extraction process, IVCon groups concern code into syntactically equal subconcerns in the unwoven-concerns panel and displays isolated concerns in the format described in Section 2.1.2.

Although the unweaving algorithm just described is straightforward, one interesting issue arose during implementation. The issue concerns how to display holes in place of overlapping, but unequal, regions. For example, let us reconsider the woven-body code of Figure 2.1. In that figure, a programmer has assigned an entire if statement to the security concern and has nested two audit-concern regions within that security region. There are two reasonable alternatives for unweaving this if statement:

- We could replace the entire if statement with one hole of the multi-concern color to indicate that one region of code, which in total implements multiple concerns, has been extracted.

- We could replace the entire if statement with five holes that alternate between the security-concern color and the multi-concern color, to indicate that the extracted code first contains security-concern code, then some multi-concern code, then more security-concern code, and so on.
Because it provides more precise concern-assignment information, we implemented the second of these alternatives in IVCon. Figure 2.2 shows the resulting five-hole concern extraction in the unwoven-body panel.

2.2.2.2 Weaving

Weaving is the process of translating an unwoven-view program into an equivalent woven-view program. To weave, IVCon builds the woven body from the unwoven body by iterating through all the holes in the unwoven body and filling in each hole with the appropriate concern code.

While designing the weaving algorithm, one interesting issue arose, which relates to filling multi-concern holes with code. Because a code segment that fills in a multi-concern hole has been assigned to multiple concerns, it appears in multiple places in the unwoven-concerns panel. The interesting issue occurs when the user modifies multi-concern code in one place but not another in the unwoven-concern panel. Such a modification leads to code inconsistencies when filling multi-concern holes (i.e., which one of the concern modules should be used for filling a multi-concern hole). To ensure that such code inconsistencies do not occur, we have implemented linked editing in the unwoven-concerns panel [52]. Code segments in the unwoven-concerns panel that fill in the same (multi-concern) hole get linked at runtime, so any changes made in one code location get reflected immediately in all the other code locations that fill in the same hole.

2.2.3 Third-party Libraries

IVCon also uses several libraries and code that we downloaded from third-party sources:

- **RTrees**: IVCon use RTrees (as mentioned earlier in Section 2.2.1) to efficiently determine what concerns are assigned to a particular point in a user’s program. We researched other spatial data structures such as the P-R Tree [51], IBS Tree [53] and skip lists [54]. However, we had a problem finding a correct implementation of the P-R Tree; IBS Tree and skip lists were not suited to our purpose because they do not allow dynamic addition or deletion of intervals, which is important for IVCon as users define regions dynamically. IVCon uses Hadjieleftheriou’s implementation of RTrees [55].
• **JavaCC**: IVCon also uses JavaCC, a Java-based parser generator, which is available online [56]. JavaCC’s downloadable package includes several grammars, including that of Java. By modifying the grammar files for Java, we were able to generate a Java lexer, which we use in our weaving and unwrapping algorithms to verify the lexical validity of user programs and to enforce token-level granularity.

• **Clipboard**: IVCon uses a third-party clipboard implementation to implement the cut, copy, and paste functionalities in all of IVCon’s user-editable panels (i.e., woven-body, unwoven-body, and concerns-at-current-position panels) [57].
CHAPTER 3
MODULARIZING RUNTIME SECURITY POLICIES

In order to prevent malicious activity on computers, programmers have to ensure that their software behaves in ways that it is intended to. Malicious users often exploit security holes in a system’s implementation to gain unauthorized access to computers or to execute malicious code on computers. For example, a malicious user can exploit the weak type-safety of a weakly-typed or type-unsafe language to launch a buffer-overflow attack on a system implemented in that language. However, by incorporating security checks at appropriate places in their code, users can avoid such attacks. For example, to counter the buffer-overflow attack mentioned above, a user would precede every instance of a memory-access with code that ensures that the memory is accessed in safe ways. However, such security checks often become scattered across a software’s implementation, and thereby tedious to maintain.

To ensure that security checks do not get scattered, security engineers modularize such security checks into policies using policy-specification languages. 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. Typically, these languages are 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.

Policy-specification languages enable users to specify policies centrally and provide users with all other modularization benefits (i.e., they make security code easy to read, understand, and maintain). Traditional modularization techniques that do not capture the multi-concern nature of code can be used to modularize security policies because in most applications, security is functionally orthogonal to the rest of the code.
In this chapter, we discuss the modularization of runtime policies using LoPSiL, a policy-specification language that we have implemented, which as far as we know, is the first Turing-complete policy-specification language that provides users with language constructs that help users to conveniently manipulate location information. LoPSiL is short for Location-based Policy-specification Language. We start with a discussion of LoPSiL’s linguistic constructs and use an elementary policy to show how policies are implemented in LoPSiL. Thereafter, we describe the implementation of our proof-of-concept LoPSiL compiler.

3.1 LoPSiL

Due to the popularity of Java, particularly Java ME, as an application programming language for mobile devices [58], 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.

3.1.1 Core Linguistic Constructs

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

- Locations: In LoPSiL, Locations are (possibly abstract) places. They may refer to rooms, chairs, floors, buildings, campuses, GPS coordinates, regions of a network topology, etc. All Locations 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 4.2.1 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 *LocationDevices* 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 [59], etc. The second abstract method *LocationDevices* 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 *LocationDevices*, 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 [60]; 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 *LocationDevices*. 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 *LocationDevices* 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 Actions 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:

  – A policy may declare PolicyAssumptions upon which it relies.
  – A policy may define a handleGranularityViolation method, which will be invoked whenever all LocationDevices upon which the policy relies violate the policy’s location-granularity assumption.
  – A policy may define a handleFrequencyViolation method, which will be invoked whenever all LocationDevices 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.
  – 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.
  – 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.
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");
    }
}

Figure 3.1. Simple LoPSiL Policy that allow all actions to execute

3.1.2 Example LoPSiL Policy

Figure 3.1 contains a simple LoPSiL policy called the AllowAll policy that contains all five of the components illustrated in the previous section. The AllowAll policy 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. We describe several other policies implemented in LoPSiL in Section 4.2.1.

Existing policy-specification languages, such as Naccio [3], PSLang [2], and Polymer [4, 5], 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).

3.2 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,
void java.io.PrintStream.println(..)
* javax.swing.JOptionPane.*(..)
java.util.Date.new()

Figure 3.2. 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.

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 [61].

3.2.1 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 [62]. AspectJ compilers inline calls to advice at control-flow points specified by point cuts. 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.

- The user creates a specification of the desired policy in a .lopsil file.
- 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 3.2 contains an example and illustrates how wildcards can be used in .srm files.
Figure 3.3. Overview of the LoPSiL compiler.

- The LoPSiL compiler inputs the policy (.lopsil) and security-relevant-methods (.srm) into a lopsi2aj 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.

- 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 [62]. 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 3.3 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 have also implemented LoPSiL on the Android platform [42]. The Android implementation is available online [63] along with details about our experiences with porting LoPSiL to the Android platform, and the various overheads induced by using LoPSiL on a mobile system that is memory- and battery-constrained [32].

\footnote{We thank Joshua Finnis for the Android implementation of LoPSiL.}
CHAPTER 4
VALIDATION OF APPROACH

This chapter describes our experiences working with IVCon and LoPSiL, along with evaluating the performance overhead of the two.

4.1 IVCon

To validate our proposed approach to the modularization of CCCs, we performed case studies and performance tests to confirm, 1) to what extent IVCon provides software engineering benefits, and 2) does IVCon perform its various operations in reasonable amounts of time. This section describes those case studies and performance tests.

4.1.1 Case Studies

To improve our understanding of IVCon’s software engineering advantages and disadvantages, we used IVCon to extend several applications, including IVCon itself.

4.1.1.1 Extending IVCon

Our first case study involved extending IVCon with the following features:

- Search for text in code. This serves a feature similar to \(<\text{ctrl-f}>\) in common editors.

- Jump to a particular line number of code specified by the user. This helps users debug, as compilers typically report warnings and errors by line number.

- Open multiple files simultaneously. This enables users to build projects that span multiple files, with the ability for concerns to crosscut multiple files.
• Show in a tool-tip the concerns implemented by the code that the mouse pointer points to. This enables users to see which concerns a code segment $S$ implements without actually moving the cursor to $S$.

• Integrate linked editing in the unwoven-concerns panel. This ensures that there are no code inconsistencies when filling in multi-concern holes during weaving (as mentioned in Section 2.2.2.2).

• View flags in the woven view. This provides users with the same clarity of concern assignment in the woven view that currently exists in the unwoven view. Because concern assignments also serve as code documentation, this feature can improve users’ comprehension of woven-view code. In addition, because flags stand out well from the rest of the code, this feature helps users quickly locate code assigned to particular concerns.

• Jump to a specified concern module in the unwoven-concerns panel. This enables users to easily navigate in the unwoven-concerns panel.

• Compile and execute code directly from IVCon. This saves users from having to switch to external tools for compilation and execution.

After implementing each of these features, we recompiled IVCon and worked in the newer version to implement the next feature. This bootstrapping approach ensured that we could draw the benefits of each new feature while implementing subsequent features.

As shown in Figure 4.1, implementing the case study entailed adding 1591 lines of code to our initial IVCon prototype. The total time spent implementing the case-study features was 45 hours and 13 minutes, out of which we spent 27 minutes and 31 seconds defining and assigning code to concerns. Hence, 1.03% of the total design and implementation time involved performing overhead actions necessary to reap IVCon’s benefits (of creating, viewing, and editing code in both woven and unwoven views). The code assigned to concerns in the case study occupied 125 regions, which we assigned to 43 concerns. Taken together, these data show that our case study was a practical, medium-sized implementation effort (about one work week of time), and only a small portion (about one percent) of the implementation time had to be spent defining and assigning code to concerns.
Table 4.1. Implementation effort for IVCon case study.

<table>
<thead>
<tr>
<th>Feature implemented</th>
<th>Lines of code added</th>
<th>Time spent implementing (hr:min)</th>
<th>Number of times used defining and assigning code to concerns</th>
<th>Time spent unweaving (hr:min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find text</td>
<td>57</td>
<td>03:14</td>
<td>11</td>
<td>00:07</td>
</tr>
<tr>
<td>Goto line</td>
<td>41</td>
<td>00:48</td>
<td>7</td>
<td>00:00</td>
</tr>
<tr>
<td>Multiple files (projects)</td>
<td>446</td>
<td>10:09</td>
<td>79</td>
<td>00:03</td>
</tr>
<tr>
<td>Tool tips</td>
<td>53</td>
<td>01:44</td>
<td>8</td>
<td>00:05</td>
</tr>
<tr>
<td>Linked editing</td>
<td>364</td>
<td>15:39</td>
<td>93</td>
<td>00:09</td>
</tr>
<tr>
<td>Flags in the woven view</td>
<td>346</td>
<td>09:43</td>
<td>108</td>
<td>00:01</td>
</tr>
<tr>
<td>Jump to a concern</td>
<td>81</td>
<td>00:43</td>
<td>4</td>
<td>00:00</td>
</tr>
<tr>
<td>Compile and execute</td>
<td>203</td>
<td>03:41</td>
<td>15</td>
<td>00:03</td>
</tr>
<tr>
<td>Total</td>
<td>1591</td>
<td>45:41</td>
<td>325</td>
<td>00:28</td>
</tr>
</tbody>
</table>

4.1.1.2 Extending JHotDraw and Java Scientific Calculator

We also used IVCon to extend other programmers’ projects: JHotDraw [8] and the Java Scientific Calculator [64]. JHotDraw is a framework, implemented in approximately 33,000 lines of Java code, for developing graphics editors. The Java Scientific Calculator is implemented in approximately 25,000 lines of code.

We added the following features:

- To JHotDraw we added a feature for asking users, before exiting the program without saving changes to open files, whether they wish to save those changes.

- To the Java Scientific Calculator we added:
  - Three extra memory slots to the calculator, which in its original version had only one memory slot.
  - Grades as a unit for measuring angles (to complement degrees and radians).
  - A button that, when pressed, displays the modulus of a complex number.

Table 4.2 summarizes our implementation effort for these additional case studies. Defining and assigning code to concerns consumed about 1.59% of the total implementation time, up from the 1.03% overhead we found in the first case study. We attribute the increased overhead to the extra
Table 4.2. Implementation effort for JHotDraw and Java Scientific Calculator case studies.

<table>
<thead>
<tr>
<th>Feature implemented</th>
<th>Lines of code added</th>
<th>Time spent implementing (hr:min)</th>
<th>Number of times used defining and assigning code to concerns</th>
<th>Time spent unweaving (hr:min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JHotDraw</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ask to save files</td>
<td>256</td>
<td>07:42</td>
<td>42</td>
<td>00:07</td>
</tr>
<tr>
<td>Java Scientific Calculator</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Additional memory slots</td>
<td>542</td>
<td>04:41</td>
<td>18</td>
<td>00:05</td>
</tr>
<tr>
<td>Grade angle unit</td>
<td>10</td>
<td>00:54</td>
<td>4</td>
<td>00:01</td>
</tr>
<tr>
<td>Mod of complex numbers</td>
<td>43</td>
<td>01:25</td>
<td>12</td>
<td>00:01</td>
</tr>
<tr>
<td>Total</td>
<td>851</td>
<td>14:42</td>
<td>76</td>
<td>00:14</td>
</tr>
</tbody>
</table>

The effort required for figuring out which concerns are being implemented in code written by another programmer. The 1.59% overhead is still low and was achieved while defining and assigning code to several concerns; we found it useful to define 8 concerns in 18 regions throughout 5 JHotDraw files, and 10 concerns in 27 regions throughout 19 Java Scientific Calculator files.

4.1.2 Experiential Observations

While adding new features to IVCon, IVCon aided code production in several ways, by providing all the benefits outlined in Sections 1.4 and 2.1, as well as the bootstrapped benefits listed in Section 4.1.1.1. Assigning all code implementing each of the case-study features to a feature-specific concern was a significant aid when locating code for each concern as it was being implemented and tested, particularly because some of the concerns (e.g., MultiFile) crosscut several classes and methods (e.g., FileUtilities.newFile(), FileUtilities.openIvcFile(), Windows.stateChanged(), etc). It was similarly helpful to be able to unweave other concerns during the case study, as we frequently needed to refer to their implementations; for instance, we also defined OpenFile, SaveFile, LexerFunctions, and RTreeFunctions concerns. Many of the concerns we defined overlapped (e.g., MultiFile and OpenFile), so IVCon’s ability to handle multi-concern code was helpful.

It was also useful during the case study to assign syntactically equal code segments, which crosscut various functions, to the same concern, so we could update those code segments centrally.
For example, while implementing the MultiFile concern, we made 9 centralized updates to 3 instances of a line of code used for switching between per-file data structures; these 9 updates would have required 27 updates in standard code editors.

As evidence that the crosscutting and overlapping concerns we defined were natural and not enabled by poor code organization, we ran IVCon’s woven code through Eclipse’s automatic class-extraction refactoring tool, which only suggested a minor refactoring unrelated to our concern assignments (i.e., to move all data fields into separate data-field-only classes). While automated refactoring tools work with less effort than IVCon, they cannot in general predict all the code a programmer might want assigned to concerns.

To add the one feature to JHotDraw, we defined concerns for several operations related to the feature, and most of the concerns we defined were scattered in one way or the other. For instance, concerns such as OpenFile and SaveFile (pertaining to implementations of saving and opening files respectively) were scattered across multiple files, whereas concerns such as EndApp (code that gets executed when a JHotDraw application is terminated) and PanelOperations (code that gets executed when windows within the application are opened or closed) were scattered within the same file. Other concerns that we defined included code that implemented the new feature (AskUsersForSave concern) and code that imported various packages (Import). Code implementing these two concerns was not scattered, but assigning these code segments to a concern proved to be useful because we were updating these code segments frequently and upon unweaving, it became easier to locate them.

Similarly to the JHotDraw case study, most of the concerns in the Java Scientific Calculator case study were scattered. However, contrary to JHotDraw, they were mostly scattered within the same file itself. Again, a majority of the concerns that we defined in this case study were feature-specific, and defining these concerns enabled us to readily locate the code that we were frequently updating. For example, to implement the first feature (i.e., additional memory slots), we defined concerns named NewMemoryButton (for adding new buttons to the calculator), ButtonBackend (for handling the backend operations of buttons e.g., initializing buttons, button listener etc.), and MemoryBackend (for initializing and accessing memory locations for the calculator’s memory buttons). The only concern that was scattered across multiple files in the Java Scientific Calculator
Table 4.3. Test-file characteristics.

<table>
<thead>
<tr>
<th>File Name</th>
<th>File Size (LoC)</th>
<th>Total No. of Concerns</th>
<th>Total No. of Regions</th>
<th>Avg. Region Size (chars)</th>
<th>Max. Region Size (chars)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IVCON.java</td>
<td>61</td>
<td>5</td>
<td>7</td>
<td>135.9</td>
<td>337</td>
</tr>
<tr>
<td>Windows.java</td>
<td>2,849</td>
<td>20</td>
<td>84</td>
<td>914.4</td>
<td>24,902</td>
</tr>
<tr>
<td>StressTest.java</td>
<td>100,000</td>
<td>1,000</td>
<td>5,000</td>
<td>998.0</td>
<td>3,794</td>
</tr>
</tbody>
</table>

was the feature-specific concern pertaining to the addition of the grade unit for angle measurement, the GradeAngle concern.

IVCon helped us in the same way for this case study as it did for the first case study. Additionally, because of the larger code bases that we working with during this case study, we encountered multiple cases of a concern crosscutting through several files. For such concerns, being able to open multiple files as part of a project proved to be useful; we saved all files that a concern crosscut as part of a project, which in turn ensured quick access to every instance of the relevant concern. Moreover, we observed that for the JHotDraw case study, being able to define multi-concern code proved to be useful. The feature-specific concern (AskUsersForSave) that we were implementing overlapped with other concerns such as OpenFile, SaveFile, PanelOperations etc., and flags in the AskUsersForSave concern-module served as a quick aid to locate the position within the concern-module where we wanted to make changes.

4.1.3 Performance Evaluation

To evaluate the practicality of our design, we tested\footnote{The tests were performed on a Dell Latitude D830 with dual Intel Core2 2.2 GHz CPUs and 2 GB of RAM, running Windows XP. The times represent real time at low average load. We performed each test in sets of 100; the results shown are the averages of those sets. During stress testing, we had to increase the virtual machine’s heap size to 1.5 GB.} IVCon by assigning code to concerns in two of our own IVCon source-code files: IVCON.java and Windows.java, respectively containing 61 and 2849 lines of Java code (in the woven view) and 7 and 84 regions assigned to 5 and 20 total concerns. We also created an impractically large file of 100,000 lines, each containing 20 randomly generated single-character tokens, in order to stress test IVCon’s performance. Figure 4.3 summarizes our test-file characteristics. We emphasize that StressTest.java would be an unreasonably large (2-
Table 4.4. Performance opening and saving files.

<table>
<thead>
<tr>
<th>File Size (LoC)</th>
<th>Type of File</th>
<th>File-open Time (ms)</th>
<th>File-save Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td>.java file</td>
<td>12.96</td>
<td>9.09</td>
</tr>
<tr>
<td></td>
<td>.ivc file (no concerns)</td>
<td>30.78</td>
<td>12.60</td>
</tr>
<tr>
<td></td>
<td>.ivc file (5 concerns)</td>
<td>33.59</td>
<td>15.53</td>
</tr>
<tr>
<td>2,849</td>
<td>.java file</td>
<td>66.72</td>
<td>42.56</td>
</tr>
<tr>
<td></td>
<td>.ivc file (no concerns)</td>
<td>367.81</td>
<td>216.29</td>
</tr>
<tr>
<td></td>
<td>.ivc file (20 concerns)</td>
<td>413.22</td>
<td>250.14</td>
</tr>
<tr>
<td>100,000</td>
<td>.java file</td>
<td>11,582.80</td>
<td>474.36</td>
</tr>
<tr>
<td></td>
<td>.ivc file (no concerns)</td>
<td>16,162.50</td>
<td>4,756.17</td>
</tr>
<tr>
<td></td>
<td>.ivc file (1000 concerns)</td>
<td>16,374.81</td>
<td>5,693.11</td>
</tr>
</tbody>
</table>

Table 4.5. Performance assigning code to concerns.

<table>
<thead>
<tr>
<th>File Size (LoC)</th>
<th>Region Size (LoC)</th>
<th>Number of Nested Regions</th>
<th>Concern-assignment Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td>61</td>
<td>7</td>
<td>23.78</td>
</tr>
<tr>
<td>2,849</td>
<td>500</td>
<td>21</td>
<td>73.86</td>
</tr>
<tr>
<td></td>
<td>2,849</td>
<td>84</td>
<td>295.51</td>
</tr>
<tr>
<td>100,000</td>
<td>1,000</td>
<td>53</td>
<td>760.95</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>525</td>
<td>10,246.56</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>5,000</td>
<td>104,976.56</td>
</tr>
</tbody>
</table>

million-token) source-code file in practice; we included it in our test suite to better understand IVCon’s performance limitations.

IVCon allows users to open and save Java source-code (.java) files and IVCon (.ivc) files. IVCon files contain several serialized objects: the Java source-code string in the woven view (which is stored independently of any .java files), plus IVCon’s regionMap, concernMap, and regionTree for that program. We measured IVCon’s performance opening and saving our test files as .java files, .ivc files with no concerns defined, and .ivc files with all concerns defined. Table 4.4 displays the results. IVCon opens and saves .java files more quickly than .ivc files (which contain several potentially large, serialized data structures). As expected, IVCon’s file-open and file-save times are proportional to the length of the Java code being processed and the sizes of objects being serialized. The file-save times in Table 4.4 do not include the time taken to close the files, but file-closing times were negligible (i.e., unobservably small for all but the StressTest files, which took about 83ms to close).
Table 4.6. Performance editing concern colors and removing concerns.

<table>
<thead>
<tr>
<th>File Size (LoC)</th>
<th>Number of Characters Assigned to Concern</th>
<th>Concern-edit Time (ms)</th>
<th>Concern-removal Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td>332</td>
<td>7.34</td>
<td>9.41</td>
</tr>
<tr>
<td>2,849</td>
<td>1,531</td>
<td>14.80</td>
<td>27.87</td>
</tr>
<tr>
<td></td>
<td>1,789</td>
<td>22.95</td>
<td>40.02</td>
</tr>
<tr>
<td></td>
<td>3,386</td>
<td>30.15</td>
<td>98.40</td>
</tr>
<tr>
<td>100,000</td>
<td>23,335</td>
<td>367.29</td>
<td>512.38</td>
</tr>
<tr>
<td></td>
<td>38,462</td>
<td>419.38</td>
<td>685.00</td>
</tr>
<tr>
<td></td>
<td>85,069</td>
<td>810.26</td>
<td>1,050.18</td>
</tr>
</tbody>
</table>

Table 4.7. Performance weaving and unweaving code.

<table>
<thead>
<tr>
<th>Lines of Woven Code</th>
<th>Number of Concerns</th>
<th>Holes in Unwoven Body</th>
<th>Weaving Time (ms)</th>
<th>Unweaving Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td>0</td>
<td>0</td>
<td>3.27</td>
<td>2.66</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>7</td>
<td>7.03</td>
<td>18.76</td>
</tr>
<tr>
<td>2,849</td>
<td>0</td>
<td>0</td>
<td>66.45</td>
<td>57.36</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>127</td>
<td>943.22</td>
<td>625.45</td>
</tr>
<tr>
<td>100,000</td>
<td>0</td>
<td>0</td>
<td>3,527.19</td>
<td>3,445.31</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>7,760</td>
<td>537,959.40</td>
<td>89,534.50</td>
</tr>
</tbody>
</table>

Although creating a new concern in IVCon takes only a constant amount of time and is a fast (0ms to 16ms) operation, we next describe IVCon’s performance during three heavier-weight concern-manipulation operations. Tables 4.5 and 4.6 show IVCon’s performance during concern assignment, editing, and removal. IVCon’s performance when assigning code to a concern (Table 4.5) depends on the size of the region \( r \) being assigned to the concern and the number of regions nested within \( r \); higher values for these two parameters imply more considerations of code-color changes and therefore more time to complete the concern-assignment operation. Because editing concern names takes a negligible amount of time (0ms to 16ms), the biggest factor in editing a concern is actually changing its color (Table 4.6), which again depends on the amount of text (i.e., the number and size of regions) assigned to the concern being recolored. Similarly, when a user completely removes a concern in the woven view, most of the time IVCon spends completing this operation involves recoloring the code that had been assigned to that concern; hence, concerns assigned to more and larger regions take more time to remove than concerns assigned to fewer and smaller regions.
Finally, we measured IVCon's performance weaving and unweaving code, as presented in Table 4.7. Based on the descriptions of these algorithms in Section 2.2.2, we would expect that the biggest factors determining weaving and unweaving times are the number of concerns being woven or unwoven, the number of holes being filled in or created, and the number of characters filling in or being extracted from holes. The results in Table 4.7 match these expectations. Completely weaving and unweaving code in all the reasonable-sized test files took less than one second.

In summary, our performance results demonstrate that the prototype implementation is efficient when operating on reasonably sized source-code files. However, the implementation would need additional optimizations to perform tolerably efficiently on extremely large source-code files, such as those containing millions of source-language tokens.

4.2 LoPSiL

This section describes our experiences working with LoPSiL and summarizes the performance overhead incurred from using LoPSiL on a mobile device.

4.2.1 Case Studies

To test the usefulness of the location constructs that we have defined for LoPSiL, we implemented and tested several small location-based policies using LoPSiL. The first of these 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: 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 4.1, requires that monitored applications can only access the device’s 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

---

2For a detailed analysis of our performance tests, we refer readers to our journal article [32].
3We thank Sean Barbeau 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");
    }
}

Figure 4.1. 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();
    }
}

Figure 4.2. Abbreviated LoPSiL policy requiring that navigational aid appear when the device’s current location deviates from its expected path.

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 4.2, 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.
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 4.3. Abbreviated LoPSiL policy requiring robot-control software to encrypt outgoing messages when the robot is outside a secure-region perimeter.

- Safe-region Policy\(^4\): Another interesting sort of policy expressible in LoPSiL is shown in Figure 4.3. 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.

- 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

\(^4\)We thank Robin Murphy for suggesting this policy.
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();
        }
    }
}

Figure 4.4. A location-dependent social-networking policy specified in LoPSiL.

– 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 4.4.

4.2.2 Experiential Observations

Having implemented the example policies described in above, we believe that the six core constructs underlying LoPSiL serve as good abstractions 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 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 these example policies 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 (Figures 4.2–4.4) and can define custom operators on locations when the built-in methods are insufficient.

### 4.2.3 Performance Evaluation

To evaluate the practicality of our proof-of-concept LoPSiL compiler, we measured the overhead incurred from using four of our LoPSiL policies, **AllowAll** (Figure 3.1), **AccessControl** (Figure 4.1), **DeviationFromPath** (Figure 4.2), and **SocialNetworking** (Figure 4.4). We recorded the following metrics for the original application and for the rewritten application that enforces our policies:

- Time taken for the application to finish execution.
- Memory usage of the application.

---

5*Tests were conducted on an unlocked HTC Dream, that had a 528 MHz processor, 192 MB RAM, and 1 GB storage. We thank Joshua Finnis for performing the various performance tests.*
Table 4.8. Overhead incurred from using LoPSiL

<table>
<thead>
<tr>
<th>Policy</th>
<th>Time taken (ms)</th>
<th>Memory usage (Bytes)</th>
<th>Battery usage (%age of total battery)</th>
<th>Code size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AllowAll</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base</td>
<td>36.981</td>
<td>2,967</td>
<td>16.25</td>
<td>13,379</td>
</tr>
<tr>
<td>With Policy</td>
<td>37.725</td>
<td>3,211</td>
<td>16.88</td>
<td>68,939</td>
</tr>
<tr>
<td>Overhead</td>
<td>0.744</td>
<td>244</td>
<td>0.63</td>
<td>55,560</td>
</tr>
<tr>
<td><strong>AccessControl</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base</td>
<td>86.203</td>
<td>3,976</td>
<td>27.00</td>
<td>14,386</td>
</tr>
<tr>
<td>With Policy</td>
<td>102.458</td>
<td>4,168</td>
<td>26.88</td>
<td>70,394</td>
</tr>
<tr>
<td>Overhead</td>
<td>16.255</td>
<td>192</td>
<td>-0.12</td>
<td>56,008</td>
</tr>
<tr>
<td><strong>DeviationFromPath</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base</td>
<td>3.258</td>
<td>4,821</td>
<td>6.38</td>
<td>15,419</td>
</tr>
<tr>
<td>With Policy</td>
<td>26.254</td>
<td>5,196</td>
<td>10.50</td>
<td>71,602</td>
</tr>
<tr>
<td>Overhead</td>
<td>22.996</td>
<td>375</td>
<td>4.12</td>
<td>56,183</td>
</tr>
<tr>
<td><strong>SocialNetworking</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base</td>
<td>2.450</td>
<td>4,679</td>
<td>5.38</td>
<td>14,142</td>
</tr>
<tr>
<td>With Policy</td>
<td>82.860</td>
<td>5,035</td>
<td>9.88</td>
<td>71,357</td>
</tr>
<tr>
<td>Overhead</td>
<td>80.410</td>
<td>356</td>
<td>4.50</td>
<td>57,215</td>
</tr>
</tbody>
</table>

- Battery usage of the application.
- Code size of the application.

The results from our experiments are summarized in Table 4.8, in which we mention for each policy and metric, the recorded value for the application without the policy enforced (mentioned as *Base*), the recorded value with the policy enforced (mentioned as *With Policy*), and the difference between the two values (mentioned as *Overhead*).

We observe that in terms of the running time of an application, the base overhead incurred from using LoPSiL was approximately 2%. We measure the base overhead of LoPSiL as the overhead from the *AllowAll* policy, because the *AllowAll* policy allows all actions and does not perform any additional computations. On the other hand, all the other policies perform additional computations to determine the result of a security-relevant method, and those additional computations increase significantly the running time of the application with those policies enforced. Additionally, on average, the memory usage overhead was recorded to be 6.5%, which is seemingly reasonable. The average battery usage overhead was 16.5% (the 95% confidence interval is between -3.2% to 36.4%). Battery usage in Android is reported with a granularity of 1% of total battery usage, which explains the negative overhead of -0.12% in the *AccessControl* policy. Another interesting overhead to observe is the increased code size of the application. This overhead ranges approximately between
55 kB and 57 kB, and most of the increased code size (55 kB, as can be observed from the AllowAll base policy) is attributed to the LoPSiL libraries.
CHAPTER 5

CONCLUSIONS

We have discussed in this thesis, two tools that enable users to modularize their code: We have discussed IVCon, a GUI-based tool for conveniently creating, examining, and modifying code in, and translating between, woven and unwoven views of code. IVCon differs from existing aspect-visualization tools by providing a combination of (1) translations between woven and unwoven views (2) many-to-many relationships between concerns and code, (3) token-level granularity in concern assignment, and (4) a GUI designed to make all of this convenient. We have also discussed LoPSiL, which enables users to modularize security code using traditional modularization units (aspects). LoPSiL differs from existing policy-specification languages because, as far as we are aware, it is the first Turing-complete policy-specification language that provides users with language constructs for conveniently specifying location-based constraints.

This chapter concludes this thesis by discussing some of the lessons that we have learned about code modularization and providing several extensions to this work.

5.1 Discussion

Encapsulating code segments into methods and replacing them with calls to those methods produces a well-structured program, and provided that chosen method names are descriptive enough, it aids program comprehension. However, such practices adversely affect the code-maintenance process. We observed these problems while working on the JHotDraw case study described in Section 4.1.1.2. During the case study, although we could understand how JHotDraw was implemented, we had to browse through several files to locate the exact place in the code where we had to make changes to implement the desired functionality in JHotDraw. This, we observed, happens because traditional modularization techniques (such as Extract into Method, Extract into Class etc.) ultimately lead to the scattering of concern implementations. IVCon targets this
scattering problem by displaying concern implementations in the context of the rest of the program (i.e., the complete program) in one view (i.e., the woven view), and provides the standard benefits of code modularization by isolating concern implementations from the program core in the other view (i.e., the unwoven view).

While working on the case studies described in Section 4.1.1, we gained further insights into the benefits of modularizing code using IVCon, including how IVCon would be best used in a practical setting. To reap maximum benefits from IVCon, we propose that programmers should use IVCon as part of the development process; programmers should assign code to concerns as they are writing it. We observed (while creating and implementing feature-specific concerns for new features) that assigning code to concerns while writing it incurs very little overhead. We expect this overhead would be much less than the overheads of 1.03% and 1.59% that we reported in our case studies (Sections 4.1.1.1 and 4.1.1.2, respectively). Also, we could further reduce this overhead by implementing an interface that is more conducive to concern assignment than IVCon's current interface, which is primarily targeted towards providing woven and unwoven views of user code and conveying the benefit of IVCon's concern modules.

While IVCon provides multiple software-engineering benefits by modularizing concern code, it provides no constructs for encapsulating or interfacing with that modularized code. That is, programmers may modify unwoven concern code arbitrarily, as long as the resulting woven program is (lexically, syntactically, and semantically) valid. The woven and unwoven views are equivalent, but only the woven view is checked for validity—during compilation IVCon checks that the woven program is valid Java, and after weaving (and before unweaving) IVCon checks that all declared regions begin at token beginnings and end at token endings.

For example, it is possible in the unwoven view to declare a new variable for one concern that another concern will then implicitly have access to. As long as the woven program is valid, such modifications are allowed, and IVCon's interface currently provides no indication of which variables are in scope at which parts of unwoven code (though one could imagine extending IVCon with such a feature). For now, IVCon users can only see which variables are in scope in the unwoven view by switching to, and examining code in, the woven view. In general, we expect that there will always be some tasks that are easier to perform in the woven view (e.g., understanding which variables are in scope at given positions), while others are easier to perform in the unwoven view (e.g., changing
the messages output in an audit concern); after all, these differences are what motivated IVCon and its ability to work in and translate between the woven and unwoven views.

In the presence of functionally orthogonal code segments, IVCon’s concern modules provide the same benefits to users as traditional modularization techniques. For example, if we use IVCon to isolate code implementing security in a program, we can view all security code in one place in the unwoven-concerns panel (in the unwoven view), which offers program comprehension benefits similar to those offered by policy-specification languages.

Also, we have derived from our experiences working with LoPSiL that traditional modularization techniques prove to be a good solution for modularizing CCCs that are functionally orthogonal to the functionality of the rest of the program. This is why we used AspectJ, a traditional modularization technique, to implement LoPSiL. Our belief is further confirmed by the fact that aspect-oriented languages are typically used to implement the logging functionality (which is typically a functionally-orthogonal concern) in many applications.

5.2 Future Work

Several opportunities exist for improving upon work that we have discussed in this thesis. Some of the directions that this work could take on in the future are enumerated as follows:

- We would like to collect data from other users to measure IVCon’s effectiveness at improving the software development process. Moreover, we would like to further evaluate IVCon’s empirical performance to determine whether and how the weaving and unweaving operations could be optimized, especially for larger source-code files.

- It would also be useful for IVCon to let users select a subset of concerns to export to an external file. This would provide a benefit similar to Hyper/J’s on-demand remodularization, with which users can specify a set of features to include in their program [25]. We expect that implementing this capability would require only minor changes to IVCon’s unweaving algorithm. A more challenging problem, however, would be to display in one location, concern code from CCCs that span multiple files in a project. This would enable users to independently reason about and edit CCCs from various files in one location.
Figure 5.1. IVCon’s newer interface for reduced concern assignment times.

- Although our prototype version of IVCon is aimed towards users programming in a Java environment, we believe that the principles underlying IVCon apply to software in any language. To validate our belief, we would like to add support for other languages to IVCon. Again, we expect that implementing this capability would entail making minor changes to the interface (to allow for users to select the environment they wish to work in), and adding JavaCC grammars of other languages to IVCon’s lexer (to lexically validate user code and enforce token-level granularity).

- We could reduce the overhead incurred from the concern-assignment process by making minor changes to IVCon’s interface. One of the features that would reduce this overhead would be to enable users to import concerns (concern names and colors) from other IVCon files; doing so would reduce the time taken to define concerns by eliminating the need to specify a name and a color for concerns. Additionally, it would ensure consistency in the choice of concern colors across different files. Another feature that would reduce the concern-assignment overhead would be the use of toggle buttons for concern assignments. In such an interface, users would...
toggle a concern button (one for every concern) to “ON” to either (1) assign selected code to the concern, (2) de-assign the concern from the selected code, or (3) assign to the concern, text that the user will type. Moreover, users would not be prompted to specify names for regions; instead, the new interface would consist of an activity panel that would inform users that a region with the name “concernname_region_i” has been defined, and would provide a link to allow users to change the name of the region. Figure 5.1 illustrates this proposed interface.

- We would like to model our IVCon language in a formal calculus and prove that (1) our translation algorithms are inverses of each other, (2) our translations are semantics preserving (i.e., translation soundness), and (3) we can unweave any concern in any woven program and weave any concern into any unwoven program (i.e., translation completeness).

- Concern information obtained by users explicitly assigning code to concerns can tell us how concerns are related to each other. More specifically, it can help tell us whether a concern is functionally orthogonal to another concern or not. Using this information we can suggest to users whether or not it makes sense to refactor code segments into different classes or not.

- Instead of using a GUI-based tool to define concern assignments in code, we could alternatively implement a language that allows users to “code” the concern assignments. This could be made possible by using Java annotations in order to mark the beginning and end of regions. Having such a language would possibly reduce the overhead of concern assignment that we have described in our case studies (by eliminating the time that a user takes to select code before assigning it to a concern).

- We could extend LoPSiL to allow users to specify and enforce multiple policies on their programs. As previous work has shown, this would increase a user’s effort while specifying policies because users will have to ensure that their policies are effectless [5, 4]. Moreover, we would have to provide mechanisms within LoPSiL that can combine decisions from various policies.
• As mentioned in Section 3.2.1, as a result of using AspectJ as our application-rewriter, our implementation does not make decisions about the execution of library methods invoked by other library methods. As a result LoPSiL does not provide complete mediation of security-relevant methods. We could circumvent this limitation by writing our own LoPSiL enforcement-code inliner (e.g., using tools like the Bytecode Engineering Library [65]), as previous work has done [2, 4, 5], but at the price of significantly increased implementation complexity.

• Finally, we would like to model LoPSiL in a formal language so that we have a complete and implementation-independent specification of our language.
LIST OF REFERENCES


