SECURE SYSTEMS DEVELOPMENT USING
SECURITY-TYPED LANGUAGES.

A Thesis in
Computer Science and Engineering

by

Boniface Patrick Hicks

© 2007 Boniface Patrick Hicks

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2007
The thesis of Boniface Patrick Hicks was reviewed and approved* by the following:

Patrick McDaniel  
Associate Professor of Computer Science and Engineering  
Thesis Advisor, Chair of Committee

John Hannan  
Associate Professor of Computer Science and Engineering

Sencun Zhu  
Assistant Professor of Computer Science and Engineering

Trent Jaeger  
Associate Professor of Computer Science and Engineering

Stephen Simpson  
Professor of Mathematics

Michael Hicks  
Assistant Professor of Computer Science, University of Maryland  
Special Member

Raj Acharya  
Professor of Computer Science and Engineering  
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.
Abstract

To protect sensitive, electronic data against corruption and leakage by computer applications, business, military, medical, and other organizations establish high-level security policies. For these to be effective, however, the policies must be implemented in computer systems. This introduces a semantic gap between high-level, security policy semantics and the low-level, operational semantics of computer systems. Bridging this semantic gap by proving that high-level policy semantics are actually implemented in a computer system is a daunting problem.

Security-typed languages are a new tool for building applications that verifiably implement information flow policies. By joining these language tools with systems principles, we find that it is possible to bridge the semantic gap in security policy. In particular, we focus on the systems principles of modularity, separation of security policy from mechanism, policy expressiveness and portability. When applying these systems principles to existing security-typed languages, additional policy and runtime infrastructure necessary is needed.

In this dissertation, we develop designs and implementations of the runtime and policy infrastructure that is lacking. We also provide systems architectures, formal analyses and software engineering tools, along with multiple applications that use this infrastructure to verifiably implement high-level security policies in systems of applications. We find that, by applying some systems principles with the infrastructure they require, we can use STLs for building secure systems that verifiably enforce high-level security policies.
# Table of Contents

List of Listings .......................................................... viii
List of Tables ............................................................. ix
List of Figures ............................................................. x
Acknowledgments .......................................................... xii
Dedication ................................................................. xv

Chapter 1. Introduction .................................................. 1
  1.1 Information flow policies ........................................... 5
  1.2 Motivating Example ................................................ 8
  1.3 Contributions ....................................................... 10
  1.4 Dissertation Outline .............................................. 12

Chapter 2. Background and Related Work .......................... 15
  2.1 Policy Systems ..................................................... 15
    2.1.1 Improving administration .................................... 16
    2.1.2 Improving expressiveness .................................... 16
    2.1.3 Highly decentralized policy ................................ 18
    2.1.4 Formal analysis with policy languages ................. 19
    2.1.5 Information flow policy .................................... 20
    2.1.6 MLS security model ......................................... 21
  2.2 Systems Security .................................................. 22
    2.2.1 Mandatory access controls .................................. 23
    2.2.2 Application-level MAC ...................................... 24
  2.3 Security-Typed Languages ..................................... 25
    2.3.1 Information flow control .................................... 25
    2.3.2 The Decentralized Label Model (DLM) .................. 27
    2.3.3 Qualities that map well to DLM ......................... 28
    2.3.4 Declassification ............................................ 29
    2.3.5 STL applications ........................................... 31
### Chapter 3. A Real-World Application

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Overview</td>
<td>34</td>
</tr>
<tr>
<td>3.1.1 Security policy</td>
<td>35</td>
</tr>
<tr>
<td>3.2 Building a secure application with Jif</td>
<td>36</td>
</tr>
<tr>
<td>3.2.1 A principal store</td>
<td>37</td>
</tr>
<tr>
<td>3.2.2 System channel policy</td>
<td>38</td>
</tr>
<tr>
<td>3.2.3 Declassification</td>
<td>40</td>
</tr>
<tr>
<td>3.2.4 Relabeling message bodies</td>
<td>41</td>
</tr>
<tr>
<td>3.2.5 Label conflict resolution</td>
<td>43</td>
</tr>
<tr>
<td>3.3 JPmail architecture using a PKI</td>
<td>44</td>
</tr>
<tr>
<td>3.4 Principal Infrastructure</td>
<td>46</td>
</tr>
<tr>
<td>3.4.1 Example policy</td>
<td>47</td>
</tr>
<tr>
<td>3.4.2 Cryptographic principals</td>
<td>48</td>
</tr>
<tr>
<td>3.4.3 Declassifiers</td>
<td>49</td>
</tr>
<tr>
<td>3.5 Security Evaluation for JPmail</td>
<td>49</td>
</tr>
<tr>
<td>3.6 Language Development Tools</td>
<td>51</td>
</tr>
<tr>
<td>3.6.1 Principal determination</td>
<td>52</td>
</tr>
<tr>
<td>3.6.2 Labeling data</td>
<td>53</td>
</tr>
<tr>
<td>3.6.3 Fixing information flows</td>
<td>53</td>
</tr>
<tr>
<td>3.6.3.1 Constraints, declassification, and implicit flows</td>
<td>54</td>
</tr>
<tr>
<td>3.6.3.2 Relaxing security checking</td>
<td>56</td>
</tr>
<tr>
<td>3.7 Resolving Information Flow Conflicts in Jif</td>
<td>57</td>
</tr>
<tr>
<td>3.7.1 The Jif compiler</td>
<td>57</td>
</tr>
<tr>
<td>3.7.2 Fixing information flows</td>
<td>58</td>
</tr>
<tr>
<td>3.8 Jifclipse</td>
<td>61</td>
</tr>
<tr>
<td>3.8.1 Fixing information flows</td>
<td>61</td>
</tr>
<tr>
<td>3.8.1.1 Constraint resolution principle</td>
<td>63</td>
</tr>
<tr>
<td>3.8.1.2 Declassifier inference principle</td>
<td>65</td>
</tr>
<tr>
<td>3.8.1.3 Implicit flow principle</td>
<td>66</td>
</tr>
<tr>
<td>3.8.2 Rapid prototyping principle</td>
<td>67</td>
</tr>
<tr>
<td>3.8.3 Ease-of-Use modifications</td>
<td>67</td>
</tr>
<tr>
<td>3.8.4 Exam room revisited</td>
<td>69</td>
</tr>
</tbody>
</table>

### Chapter 4. Trusted Declassification

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Example</td>
<td>72</td>
</tr>
<tr>
<td>4.2 Semantics and properties of FJifP</td>
<td>75</td>
</tr>
<tr>
<td>4.2.1 Introduction to FJifP</td>
<td>75</td>
</tr>
<tr>
<td>4.2.2 Definitions</td>
<td>76</td>
</tr>
<tr>
<td>4.2.3 Subtyping</td>
<td>79</td>
</tr>
<tr>
<td>4.2.4 Typing Rules</td>
<td>79</td>
</tr>
<tr>
<td>4.2.5 Evaluation</td>
<td>81</td>
</tr>
<tr>
<td>4.2.6 Type System Properties</td>
<td>81</td>
</tr>
<tr>
<td>4.2.7 Noninterference</td>
<td>85</td>
</tr>
<tr>
<td>4.2.8 Related Work</td>
<td>89</td>
</tr>
<tr>
<td>4.3 Policy framework implementation</td>
<td>90</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td>6.5.1 Example</td>
</tr>
<tr>
<td></td>
<td>6.5.2 Applications</td>
</tr>
<tr>
<td>Chapter 7</td>
<td>Policy-Driven Runtime System Infrastructure</td>
</tr>
<tr>
<td></td>
<td>7.1 Security for STL Applications</td>
</tr>
<tr>
<td></td>
<td>7.1.1 Automatic analysis performed by STL compiler</td>
</tr>
<tr>
<td></td>
<td>7.1.2 Runtime system principles</td>
</tr>
<tr>
<td></td>
<td>7.2 Runtime system</td>
</tr>
<tr>
<td></td>
<td>7.2.1 Channels</td>
</tr>
<tr>
<td></td>
<td>7.2.2 Example</td>
</tr>
<tr>
<td></td>
<td>7.2.3 A principled runtime system</td>
</tr>
<tr>
<td></td>
<td>7.3 FlowWall</td>
</tr>
<tr>
<td></td>
<td>7.3.1 An information flow policy for a firewall</td>
</tr>
<tr>
<td></td>
<td>7.3.2 Runtime system: Labeling inputs and outputs</td>
</tr>
<tr>
<td></td>
<td>7.3.3 High-level policy infrastructure</td>
</tr>
<tr>
<td></td>
<td>7.3.3.1 Semantic granularity</td>
</tr>
<tr>
<td></td>
<td>7.3.3.2 Establishing relationships between labels</td>
</tr>
<tr>
<td></td>
<td>7.4 Evaluation</td>
</tr>
<tr>
<td></td>
<td>7.4.1 Analyzing FLOWWALL's security</td>
</tr>
<tr>
<td></td>
<td>7.4.2 Performance</td>
</tr>
<tr>
<td></td>
<td>7.4.3 Evaluating prior applications</td>
</tr>
<tr>
<td></td>
<td>7.5 Summary</td>
</tr>
<tr>
<td>Chapter 8</td>
<td>Conclusion and Future Work</td>
</tr>
<tr>
<td></td>
<td>8.1 Modularity, Expressiveness, Portability, and Separability</td>
</tr>
<tr>
<td></td>
<td>8.2 Bridging the Semantic Gap in Security Policy</td>
</tr>
<tr>
<td></td>
<td>8.3 Future Work</td>
</tr>
<tr>
<td></td>
<td>8.3.1 Improving portability</td>
</tr>
<tr>
<td></td>
<td>8.3.2 Separability</td>
</tr>
<tr>
<td></td>
<td>8.3.3 Improving modularity and practicality</td>
</tr>
<tr>
<td></td>
<td>8.3.4 Expressiveness</td>
</tr>
<tr>
<td></td>
<td>8.4 From Theory to Practice</td>
</tr>
<tr>
<td>References</td>
<td></td>
</tr>
</tbody>
</table>
List of Listings

1.1 Java code for a login protocol using a password dialogue. .......... 8
2.1 Simple example illustrating label creep. .......................... 26
2.2 Code snippet illustrating labels on dynamic labels. ................. 28
3.1 Jif code for method Exam.runExam. .............................. 59
5.1 Jif code with inputs from stdin and a stats file and outputs to stdout and a configuration file. ................................. 101
6.2 Renaming levels in PALMS. ........................................ 137
6.3 A sample run in PALMS. .......................................... 137
6.1 Code for AllFlows ................................................. 138
7.1 Keeping statistics on incoming packets. ............................ 143
7.2 A faulty first attempt at extending the Stats class with a method for printing the statistics. ................................. 144
7.3 A valid approach to extending the Stats class with a method for printing the statistics. ........................................ 145
7.4 Robust approach using Channels. .................................. 145
7.5 Jif signature API for Channel abstraction. .......................... 147
7.6 Policy entries for standard I/O channel. ............................ 150
7.7 Policy class generated automatically from the policy in Listing 7.6. 150
7.8 A simple use of standard I/O. ..................................... 151
7.9 IPTables firewall rules ............................................. 153
7.10 Main loop for getting, processing and accepting or dropping packets. ...................................................... 155
8.1 Problematic dynamic updating. ...................................... 167
### List of Tables

5.1 A list of trusted applications in the SELinux release for Fedora Core 5 using mls-strict policy version 20. ........................................ 99

5.2 Time (ms) to perform one rotation of 50MB of log files in both Jif and C, and time (ms) to start up the Jif process using SIESTA (includes Jif-ness validation and compliance checking). ............................ 115

6.1 BNF grammar for SELinux MLS constraint language. ...................... 124

7.1 Performance measurements. .................................................. 158

7.2 Evaluating prior applications .............................................. 159
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Semantic gap</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Bridging the semantic gap</td>
<td>10</td>
</tr>
<tr>
<td>1.3</td>
<td>Contributions</td>
<td>11</td>
</tr>
<tr>
<td>1.4</td>
<td>Dissertation roadmap</td>
<td>14</td>
</tr>
<tr>
<td>2.1</td>
<td>Military security lattice with category sets</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>Email architecture</td>
<td>35</td>
</tr>
<tr>
<td>3.2</td>
<td>Email diagram</td>
<td>45</td>
</tr>
<tr>
<td>3.3</td>
<td>Research lab security policy</td>
<td>47</td>
</tr>
<tr>
<td>3.4</td>
<td>Error message</td>
<td>60</td>
</tr>
<tr>
<td>3.5</td>
<td>Jifclipse suggests changing the declared label on a variable</td>
<td>61</td>
</tr>
<tr>
<td>3.6</td>
<td>The Outline View, similar to the outline view in Eclipse, quickly provides information about the methods in a file.</td>
<td>64</td>
</tr>
<tr>
<td>3.7</td>
<td>The programmer can use the Constraints View to view all of the constraints generated by a line of code and navigate through related constraints.</td>
<td>65</td>
</tr>
<tr>
<td>3.8</td>
<td>Jifclipse suggests adding a declassifier to resolve a problem</td>
<td>66</td>
</tr>
<tr>
<td>3.9</td>
<td>The PC Label View highlights which lines of the code modify the program counter.</td>
<td>67</td>
</tr>
<tr>
<td>3.10</td>
<td>Screen shot for implicit flow highlighting</td>
<td>68</td>
</tr>
<tr>
<td>4.1</td>
<td>A simple example</td>
<td>73</td>
</tr>
<tr>
<td>4.2</td>
<td>A simple policy</td>
<td>73</td>
</tr>
<tr>
<td>4.3</td>
<td>Example acts-for hierarchy and declassifier context.</td>
<td>73</td>
</tr>
<tr>
<td>4.4</td>
<td>Figure 4.1, rewritten in FJifP</td>
<td>76</td>
</tr>
<tr>
<td>4.5</td>
<td>FJifP Language Syntax</td>
<td>77</td>
</tr>
<tr>
<td>4.6</td>
<td>Security Context Judgements</td>
<td>78</td>
</tr>
<tr>
<td>4.7</td>
<td>Auxiliary Definitions</td>
<td>80</td>
</tr>
<tr>
<td>4.8</td>
<td>Subtyping Rules</td>
<td>81</td>
</tr>
<tr>
<td>4.9</td>
<td>Typing Rules</td>
<td>82</td>
</tr>
<tr>
<td>4.10</td>
<td>Evaluation Rules</td>
<td>83</td>
</tr>
<tr>
<td>4.11</td>
<td>Integrating an external policy into Jif.</td>
<td>90</td>
</tr>
<tr>
<td>4.12</td>
<td>Policy language syntax</td>
<td>91</td>
</tr>
</tbody>
</table>
4.13 A closure for declassifying the cipher text generated by triple DES encryption. The standard constructor is defined, but not displayed. . . . . 92

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Information flow</td>
<td>100</td>
</tr>
<tr>
<td>5.2</td>
<td>Lattice for logrotate</td>
<td>102</td>
</tr>
<tr>
<td>5.3</td>
<td>Architecture overview</td>
<td>104</td>
</tr>
<tr>
<td>5.4</td>
<td>SIESTA layout</td>
<td>109</td>
</tr>
<tr>
<td>5.5</td>
<td>The compliance testing process.</td>
<td>112</td>
</tr>
<tr>
<td>7.1</td>
<td>Compiler architecture</td>
<td>148</td>
</tr>
<tr>
<td>7.2</td>
<td>Firewall diagram</td>
<td>152</td>
</tr>
<tr>
<td>7.3</td>
<td>FLOWWALL diagram</td>
<td>152</td>
</tr>
<tr>
<td>8.1</td>
<td>Bridging the semantic gap revisited</td>
<td>164</td>
</tr>
</tbody>
</table>
I never had aspirations of getting a Ph.D. in anything. After discovering a whole new dimension of life by coming to know Jesus Christ and trying to live in Him, I had even less interest in doing anything in computer science, let alone getting a Ph.D. This completed dissertation is a testimony to the truth spoken by the Angel Gabriel to Mary, that “all things are possible with God.” It is also a testimony to the great many people through whom He worked and through whom He continually showed me His Face.

The fact that I returned to computer science after entering a monastery and being ordained a priest must be attributed solely to Archabbot Douglas Nowicki. He had the vision to see what I did not want to see and the courage to challenge me to pursue it and the love to send me out with the grace of obedience as my confidence, my strength and my companion. A special mention must also go here to my friend and monastic classmate, Dr. Patrick Doering who originally planted the seed for me to go back to Penn State and continue research in computer science, and who himself preceded me in doctoral studies. Also, Prof. John Hannan and the CSE staff were ever generous and understanding as I renewed my studies after years in absentia.

After returning to Penn State with additional graces of monastic vows and holy orders, I faced greater challenges academically than ever in my life. In these times, the saints sustained me by their prayers and their inspiring witness, most notably St. Benedict and St. Paul of the Cross, and above all our Lady, whose motherhood constantly encouraged me and carried me through the greatest anxieties. Her words to St. Juan Diego, “Am I not here, I who am your mother? Do you need anything else?” echoed repeatedly through my mind as I stared at broken software systems and a jumble of words that were supposed to become a conference submission. Things especially turned around for me when I decided to dedicate my Ph.D. to Servant of God, John Paul II, who went to God in the first year of my studies. He is the inspiration and strength behind the creation of JPmail.

I expected that my Ph.D. would be a daily drudgery for which I would just put in the time and suffer through it. In the end, I could not have been more wrong. The first great blessing was having a class with Prof. Patrick McDaniel. His class project turned into Trusted Declassification, started a very fruitful collaboration with Dave King and sent me in the direction that led to this work. He then invited me into the SIIS Lab, a whole font of blessings in itself. Working with and living in a lab with Patrick Traynor, Kevin Butler, Will Enck, Luke St. Clair, Lisa Johansen and Sandra Rueda was a joy I
will never forget and will always thank God for. It will always be associated in my mind with the taste of MacLanahan’s subs and Starbucks coffee. Among these superstars, Will deserves special mention as my lab neighbor and living man page. He was forever generous in sharing his expansive and detailed knowledge at any time of the day or night. Sandra was a special consolation for me as one with whom I could share both Jif and Sunday Mass.

In addition to bringing me into the SIIS Lab, Patrick McDaniel reflected for me the face of God the Father through being a father to me throughout my Ph.D. He had a high-level vision that never faded, an endless font of enthusiasm by which he repeatedly bolstered me up in the face of failure, and enough love to challenge me when I began to flag in the journey. He was a great mentor, teaching me how to write papers, how to give presentations, how to work with industry on projects and how to be an adviser, both by his example and his conscious reflections on these matters. A special gift he gave me was in showing me to how teach in a way that is more like a homily. He inspired me always to teach the high-level message and look to the real-world application for each concept. He also made the SIIS Lab into a family, and served us always as a loving father, sharing in all the struggles and the joys. He remains an inspiration for me.

Without my family I never would have had the virtue to accept any of the great challenges that led to my Ph.D. They taught me most especially about unconditional love, a Love that sustained me in the hardest and darkest times. They taught me to think for myself, never to give up, to work hard and to follow my dreams. It led me in directions they never expected, but they were loving enough to follow me through it all and supported me in everything. Throughout my graduate studies, it was a special joy to share my research with my dad, whose thirst for knowledge and fearless assimilation of new material inspires me.

The most unexpected gift of my Ph.D. was in the unimaginable blessing to work closely with my brother. He has been my hero my whole life; working with him as a colleague in computer science was more than I hoped for. He was a constant support, always responding quickly and insightfully to my requests for ideas, feedback, and advice. He was a faithful and generous co-worker, inviting me to join him on a project that became my first security publication and adding invaluable quality to much of my own work. He was never afraid to be critical and was tireless in challenging me to improve the quality of my work. At the same time that he tore something down, however, he always worked with me to understand how to improve it and the final product was unquestionably better, with the added blessing that my ego was a little smaller. The sacrifice he made to serve on my committee made a significant contribution to the quality of this work, the quality of my education, and the quality of our friendship. His love has always deepened my longing for and my trust in God’s love.

While I was toiling by day on my Ph.D., I worked tirelessly by night and on Sundays with the countless little ones who inhabit State College. My many spiritual children, who include undergraduate students, graduate students, professors, mothers, and fathers are too many to name, but played an essential role in my education. Sometimes their humanity drove me crazy, but their love truly kept me sane. Serving them in spiritual direction and serving the whole Penn State Catholic community through the Sacramental ministry of the priesthood always brought perspective to the hard times
and accentuated the constant loving presence of Jesus in our lives. They made for the most diverse audience at a computer science Ph.D. defense as they came out in droves to support me by their prayers and their presence. They were always my joy and the face of Jesus shone especially through them during my years of studies. I must make a special mention to Dr. John Dziak who is part of an elite company of those few who actually read this thesis from cover to cover. He was generous to provide extensive comments, even though the work is far outside his field.

The one who the most constant support throughout all the dimensions of my life during these years was my spiritual father, Fr. Tom Acklin. His company and counsel were my continual refuge. His loving words and presence redirected me continually to Jesus. Most importantly, he taught me to love the little ones and to love my own littleness and poverty. He taught me not to fear failure, but to embrace my limitations and to hide myself in the wounds of Christ. He taught me to pray from the lowest place, when the world is collapsing around me, when my brokenness threatens to overwhelm me; not to run, but to meet the Lord in that place. Learning to love poverty and littleness are undoubtedly the greatest fruits I have received throughout the years that produced this work. May all these efforts serve only to glorify the Lord, whose Love endures forever.
Dedication

For Servant of God, John Paul II,
and the Blessed Virgin Mary, to whom he gave everything.
Chapter 1

Introduction

The proliferation of technology has succeeded in expanding opportunities for a significant number of people. It is no longer necessary to travel many miles to a specialized store to buy goods; it can be done from home through the Internet. It is no longer necessary to access complex and extensive cabinets of files or cases of books to find information; it can be accessed readily via the World Wide Web. Education and opportunity are expanded and poverty is diminished through expanding access to technology. The benefit is greater opportunity. The price, however, is greater vulnerability.

True freedom is the freedom to do what is good. Being able to steal or kill does not signify an increase in freedom, but merely an increase in opportunity, or license. What modern technology has achieved is greater opportunity, and in some cases this has produced greater licentiousness. What we would like to ensure, however, is greater freedom. The road to this lies, interestingly, by way of restriction. It is actually necessary to restrict what is possible to safeguard freedom. More than fifteen centuries ago, Saint Benedict of Nursia wrote in his rule of life for monks, “Therefore we intend to establish a school for the Lord’s service. In drawing up its regulations, we hope to set down nothing harsh, nothing burdensome. The good of all concerned, however, may prompt us to a little strictness in order to amend faults and to safeguard love.” ([Fry80], Prologue 45-47) History has borne out the effectiveness of this approach—Saint Benedict’s policy, the “little rule,” has been formative throughout Western civilization for almost fifteen hundred years.

The goal of computer security is to reduce licentiousness in order to increase freedom. Computer security seeks to safeguard love and amend faults, where “love” refers to all those important services that computers may provide, especially to care for all who are weaker and more vulnerable, and “faults” refers to all those ways malicious people or accidental mistakes may cause harm. As more sensitive data is processed, transmitted and stored electronically, there is more incentive to build secure systems. This raises numerous questions, however, starting with the very definition of “security.” What does it mean for a system to be secure? Presuming some security goals can be defined and a system constructed, a second question arises: how can we determine whether the system meets its security goals?
The first question is a matter of policy. Defining the security policy of a system is a challenge that is often handled at various levels: from corporate leadership and lawmakers, who may understand very little about technology, down to programmers, who must handle the concrete concerns of data access, processing and transmission. Diverse policies for protecting sensitive data—financial, medical, personal, military—continue to be developed by lawyers, politicians, corporations, military intelligence agencies, educational institutions and many others. Government standards such as Sarbanes-Oxley (SOX) [Con02], HIPAA [oHotS03], and the Gramm-Leach-Bliley Act [Con99] seek to protect the rights of citizens and organizations against identity theft, fraud and other harmful behavior. The distance between these high-level, natural language specifications and the systems in which they must be enforced introduces a semantic gap between policy specification and implementation, as shown in the left-hand side of Figure 1.1.

A semantic gap exists between high-level, natural language security policy semantics and the operational semantics of an application that is supposed to enforce the policy. Formal policy models and software verification seek to bridge the gap, to show that program operation provably enforces high-level policy.

A cornucopia of policy models have been developed that try to capture these natural-language policies in formal semantics that are closer to the operational semantics of the computer systems in which they must be implemented. Decades of research have produced a particularly strong and expressive policy theory that models the security of data in terms of who can produce it and who can consume it. Said another way, it models the way information can flow through a system—where it can flow from, also called integrity, and where it can flow to, also called confidentiality. Policies that express the security of a system in these terms are called information flow policies. These policies serve to diminish the semantic gap by drawing policy specifications closer to machine implementations.

The second question that must be addressed in determining whether a system meets its security goals is a matter of software verification. Verification of software
may entail testing, evaluation of software engineering practice, and manual analysis of program code. Software verification seeks to prove that a computer program satisfies a high-level specification. In other words, it seeks to close the semantic gap from the bottom up by showing that program operation is faithful to the semantics of high-level policy. Unfortunately, software verification quickly becomes intractable as code size increases. It is simply too difficult for a human to reason about all the combinations of data flows and control flows that can occur when programs process diverse sets of inputs; the results of merely human analysis can be disastrous [KSRW04]. Even automated analyses, such as model checking, are quickly overwhelmed by the proliferation of possibilities an application may generate.

Continuing the theme of restriction, one way to approach software verification is determining what behaviors cannot occur in a program. If it can be proven that programs with bad behaviors (where “bad” means contrary to policy) cannot be created, the contrapositive can be ascertained, that if a program has been created it must not behave badly. Such restriction has, historically, been a theme of programming language development to ensure many important properties, such as memory safety, lexical scoping, compositionality, and exception handling. A theme hailed in the programming language community is “well-typed expressions do not go wrong” [Mil78]. By restricting what programs a programmer can write, the compiler can ensure the executables it produces will behave in good ways.

Recently, programming language technology has been extended to enforce information flow policies. Security-typed languages (STLs) restrict programmers from writing programs that do not behave according to an information flow policy. By automatically verifying that a program behaves in accord with an information flow policy, STLs make an important step towards bridging the semantic gap between high-level security policy and its enforcement in computer systems. Unfortunately, prior to this dissertation, this important technology was essentially untested for building realistic, secure systems.

In this dissertation we bring together two fields, exploring the intersection of secure systems development with the programming language technology of security-typed languages. We find that each area greatly benefits the other: STLs were not designed with systems principles in mind and, in practice, we find they lack policy and runtime infrastructure necessary to develop secure systems; without STLs, secure systems development depends on ad hoc, social assurance methods for establishing system-wide security properties, while STLs can provide verifiable implementations of security policy. We approach this problem by applying systems principles to STLs for application development.

The systems principles we focus on in this thesis are modularity, portability, separation of policy from mechanism and policy expressiveness. We briefly describe these four principles here and explore them in the next section more deeply. Systems are simpler to design and easier to maintain when they are constructed in a modular rather than monolithic manner. When considering the use of STLs in developing systems, it is especially important that STL modules be able to interact with modules that were not built using STLs. The interaction of STL modules with non-STL modules raises many interesting challenges that we address in this thesis. Furthermore, modules should not be limited to a single environment, but should be portable to diverse environments without
needing to re-work the code extensively. The principle of *policy separation*, separating security policy from mechanism, is long-standing in the systems community and plays a pervasive role throughout this thesis. Finally, the separated policy that characterizes the security properties of a system must be sufficiently *expressive* to capture realistic, high-level security policies.

Though they do not arise in the theoretical programming language models (with the possible exception of modularity), these four principles are critical when building real systems. Our initial experiments with STLS show that though they provide verifiable implementation of security policy, they fall short in all these other areas. To remedy these shortcomings, in this thesis we explore the policy and systems infrastructure, along with additional theory and analyses, that are needed to enable STLS to be used for building verifiably secure systems. The infrastructure we develop is a first effort that admits room for further development; at the same time, it serves to sketch out the perimeters of a new field that emerges from the application of a new programming language technology with the more established field of secure systems development.

We answer the following questions. How tractable is it to use this technology for building secure systems? What policies can be verifiably implemented? What additional infrastructure must be incorporated for developing real-world applications in these languages? How do applications, built in these languages, interface with existing systems? How is the software engineering paradigm changed when using these languages?

To address these questions, we put forth the following thesis:

*The application of systems principles to security-typed languages enables the development of realistic, secure systems that bridge the semantic gap between high-level security policy semantics and the operational semantics of computer systems. The systems principles needed for achieving this goal are modularity, portability, separation of security policy from mechanism, and security policy expressiveness.*

We demonstrate this thesis by building some realistic, secure systems in a security-typed language, using these systems principles. By construction, we have high assurance that these systems enforce their high-level security policies. We find that each of the systems principles is necessary for bridging the semantic gap. Conversely, we argue that not following these systems principles inhibits the development of secure systems that bridge the semantic gap between a system’s security policy semantics and its operational semantics. We give a sketch of this reasoning here.

Without the principle of *modularity*, only toy STL programs can be produced, not realistic, secure systems. Realistic systems interact with existing components, including commodity data servers and operating system channels such as files, sockets and other I/O. To build realistic, secure systems, it is important that security-typed components may be constructed that verifiably enforce their security properties, both within the STL applications as well as when data is released over OS channels and into commodity servers. This calls for language support of the systems principle of modular design and construction.
Without the principle of policy separation, it is not clear what semantic gap is being bridged, because there is no high-level policy, only policy-relevant code and annotations buried within applications. Existing STLs have not provided for a separate security policy that can be analyzed and re-configured apart from a particular application. A first step in this process is simply to encode the principal hierarchy in an external policy file.

A policy that simply consists of the principal hierarchy is too limited to handle realistic systems, however. A more expressive policy is needed to capture exceptions to normal information flows between principals (declassifications) and also the labeling of data that enters into or leaves STL modular components. Realistic applications cannot be developed without some access to declassification (even simple operations like sending encrypted data or making a password check require declassification), and this principle must work in tandem with the principle of policy separability.

Finally, it is critical that the modular components constructed as part of secure systems be portable such that they may interact with various operating systems security technologies. Modern systems are most realistic when they are portable to diverse platforms. Diverse platforms support various mechanisms for security, including various authentication mechanisms, cryptographic technologies, and operating system security tools like mandatory access controls. Realistic, secure systems should be portable to interact with diverse security technologies while still enforcing their high-level security policies with high assurance.

We evaluate this thesis by employing methods in systems, programming languages, formal policy analysis and software engineering. We explore and evaluate various approaches through the design and implementation of systems architectures; development of programming language models with proofs of security; formalization of policy systems and proofs of policy compliance; and development and implementation of software engineering principles.

The results of our investigation show that diverse, realistic applications can be built with security-typed languages, in various environments. To achieve this, however, STLs must address systems principles. We find this requires principal infrastructure, runtime infrastructure and policy analysis algorithms and tools, beyond that which existed prior to this work.

For the remainder of this chapter, we present background on information flow policies and STLs, followed by the contributions of our investigation and a roadmap for the remainder of the dissertation. In the next section we briefly describe information flow policies. Following that, we show how they are expressed through STLs. In the process, we show what obstacles must be overcome before these technologies can be used to build secure systems.

1.1 Information flow policies

A common and natural conception of security policy is user-centric, especially focusing on authentication and authorization mechanisms. Data is conceived as an extension of the user; as long as the user authenticates himself, he can access the data and deal with it as he pleases, including releasing it to someone who could not have
gained access to it in the first place. In this view, security is like a lock-box and the data owner must simply provide the right key to unlock the data. This model, often called \textit{discretionary access control} (DAC), is simple and easy to reason about. Any application that handles security-sensitive data must ensure that it stores the data in a lock-box and only provides it to the user who has the key.

Unfortunately, this simplicity gives way to insecurity and complexity when confronted with real-world systems of applications. The heart of the problem is that the user never interacts directly with the lock-box. Rather, the user interacts with an application that interacts with the lock-box. This is like trusting in a credit card charge machine at a restaurant, which may be quite trustworthy, but forgetting that before the credit card is run through the machine, it must be handled by a waiter who could steal the card information en route. Between users and their data lie complex systems that, by malice or by accident, can corrupt or leak sensitive information. Imagine an e-commerce scenario in which a user enters credit card information into a web browser connected with an e-commerce site. The card information is processed by some intervening applications and the transaction is eventually sent to the bank. Any application in this system, from the browser to the e-commerce application to the bank has the potential to leak the card.

We want to ensure that when a user enters sensitive data into a complex system of applications like this, that the data’s confidentiality and integrity is protected from end to end, from user entry to the bank. It is not tractable, however, to analyze the code of all these applications in depth with any degree of assurance. Fortunately, we do not need to know the details of how all this code functions; we are only interested in knowing that some bad behaviors, such as leaking the card information, do not occur. In other words, we want to know some information flow properties of the applications—do they ever allow sensitive data to flow to public places. These policies naturally, but formally, express real, high-level policies such as, credit card information can only flow from the user to the bank.

STLs offer a way to automatically ensure that an application enforces such properties for any possible program execution. STLs enforce noninterference \cite{Goldman82} for lattice-based information flow policies \cite{Dolev77}. In these policies, all program data is tagged with a label and the labels form a lattice, such that if label A \textit{dominates} label B in the lattice, then the application is noninterfering if data flows only from B to A in the application, but not vice versa. For example, if Secret dominates Public (as expected), then data can never flow from Secret to Public; i.e. Secret data is never leaked. Integrity policies may also be enforced, merely by considering the dual of the lattice. High integrity (trusted) data may become low integrity (i.e. flow down the lattice), but not vice versa. The STL compiler automatically restricts the programmer from writing programs that trust low-integrity data or leak secret data. In other words, given an information flow policy, an STL compiler will only compile an application that will always enforce the policy.

One of the powerful properties of information flow policies is that they allow \textit{end-to-end} reasoning about information flows. Without knowing the details of how data flows through an application, just knowing that the application enforces a particular information flow policy is enough to know that certain bad flows do not occur. In our example, it is enough to know that credit cards will be labeled Secret and Secret data
only flows out of channels that lead to the bank. The STL compiler ensures that the data, in part or in whole, is never relabeled to a lower sensitivity level than Secret throughout the code of the application. Moreover, it ensures that the data only exits the application through channels that properly handle Secret information.

STLs are merely tools, however. They are able to take an information flow policy and ensure that a program verifiably enforces it. To use them for building secure systems, it is necessary to apply systems principles. Indeed, these tools are trivialized, if they cannot be used for constructing *modular* applications that enforce *expressive*, high-level policies. On the one hand, determining that an application with no inputs and no outputs enforces an expressive security policy is useless. On the other hand, even if they can be used to construct modular applications, STLs are still useless if the only policies they enforce are so restrictive that they have no correspondance to any high-level, natural language security policy.

The goal of this work is to apply these tools to secure systems development and see whether they can be used to bridge the semantic gap in security policy. To realize this goal, we find it is necessary to apply systems principles, namely, it must be possible to compile applications that interact with real systems; to specify application policy apart from applications; to show policy is expressive enough to implement system security goals; to show policy is portable enough to re-configure applications for use in diverse operating environments and to automatically verify that programs implement their policy. Years of work in STL technology have laid the foundations for verifiability. We find that to implement systems principles of *modularity* and *portability* using *expressive* and *separable* security policies, however, it is necessary to provide further architecture, infrastructure, analysis and software engineering tools for STLs. In so doing we are able to use STLs for developing systems that bridge the semantic gap in security policy.

We now describe each of these systems principles in more detail.

**Modular** We define a *modular* application as one that can fit into an existing system of applications. It must be able to interact with non-security-typed components (such as the operating system) while still properly protecting the security of its inputs and outputs. One possible future is that all applications and operating systems would handle labeled data and so labels could pass easily from one security-typed application to the next. Currently, however, the number of security-typed applications is quite limited and so they must be able to interact with non-security-typed applications to be modular.

**Portable** A modular STL application can interact with one kind of non-STL system or application, but applications should also be portable, meaning that they can be easily re-configured to interact with various operating environments, utilizing whatever diverse security mechanisms are available. Some mandatory access control (MAC) operating systems, for example, enforce information flow policies that could be reconciled with an STL application’s policy. In other cases, the STL might be able to leverage such security mechanisms as encryption or even reduce its security requirements by using DAC protections provided by the operating system. A goal for this work is to ensure that STL applications are portable enough to handle interactions with diverse environments.
Expressive We define an expressive policy to be one that is close to the semantics of high-level security policies. The strength of the information flow policy model can be a limiting factor in constructing practical applications. Experience shows that real systems that enforce information flow policies often need to relax the policies at certain points through declassification. Declassification allows data, in part or in whole, to flow contrary to the information flow policy, from Secret to Public, for example. In the example of handling credit card information, declassification is necessary to reveal to the Public user whether the credit card (a Secret value) was accepted or not. For STLs to be useful in building real systems that bridge the semantic gap between program code and high-level policy specifications, such expressive policies are necessary.

Separable A criterion that pervades this work is policy separation. It must be possible to reason about and control an application’s policy without reference to the program in which it is used. In short, an STL with the principle of policy separability maintains the separation of policy specification from implementation, while still ensuring that a policy is enforced by a given application. All information-flow relevant policy decisions should be captured in a policy separate from the application code. Thus, this criterion shapes the way that expressiveness and portability are handled. Declassifications should not be buried in the code, but defined and controlled through a separate policy. Interactions with diverse security mechanisms should not be hidden away in each application, but expressed and configured through a separate policy. This is a key quality for mapping high-level policies, closer to natural language policies, onto their implementations in computer systems.

We now consider a small motivating example to see how these principles impact application code.

1.2 Motivating Example

Consider some Java code for implementing a login protocol. Presume that stdio connects to some keyboard input and outputs to a video display. Presume that passwdDB connects to a password database matching user names with passwords. This code may certainly be considered modular, as it interacts with non-security-typed applications such as the password database (through passwdDB) and operating system display manager (through stdio). A simple, high-level policy is that this code should display whether the user correctly entered her password without leaking any passwords.

Listing 1.1. Java code for a login protocol using a password dialogue.

```java
stdio.output("Enter user name: ");
String username = stdio.getNextInput();
stdio.output("Enter password: ");
String guess = stdio.getNextInput();
String passwd = passwdDB.getPassword(username);
```
boolean correct = passwd.equals(guess);
if (correct)
    stdio.output("Logged in.");
else
    stdio.output("Wrong password.");

STLs require that all data be labeled with an information flow policy (in practice, label defaults and label inference allow many labels to be left implicit). Unfortunately, the only way to label this code such that it will be allowed to compile in an STL would be to label input from stdio and passwdDB with the same label. The reason is that an explicit dataflow exists from passwdDB to correct and an implicit, control flow exists from correct to stdio. In other words, the user reading stdio will be able to gain some information about data in the password database. Labeling stdio and the passwdDB as the same sensitivity level is non-sensical, however, because if the passwords were already visible to the user, a password check would be unnecessary. More importantly, such a labeling would violate our high-level policy.

Rather than eliminate a program such as this, we seek to expand the expressiveness of the policy model to account for such subtle information flows. We can achieve this by adding declassification to the information flow model. Declassification allows small escape hatches to be opened in the code to allow limited information flows contrary to noninterference.

A straight-forward way to fix this problem would be to add a special operation for declassification that allows the leakage to occur.

boolean correct = declassify(passwd.equals(guess));

The problem is that this approach violates the criterion of policy separation. By hard-wiring a policy decision into the code, namely that it is ok to leak a small amount of information about any password through every guess, it is no longer easy to reconfigure this policy without changing the application code. For example, for some principals’ passwords, such as the administrator’s, it may only be desirable to execute this declassification for three failed attempts. A pervasive goal throughout this work is to prevent policy decisions such as this from being buried in the code by lifting them out into a high-level policy separate from the application. To handle this, some policy-driven, principal infrastructure is needed to configure expressive policies.

Similarly, a policy decision is required on how to label the data flowing from the password database and the data flowing into and out of stdio. Note that these policy decisions depend on the systems in which an application runs. Supporting security mechanisms found in different systems environments is crucial for building applications that are portable. It should not be necessary to rewrite the code in each new environment. Rather, to increase portability, we seek to support pluggable runtime infrastructure that can be swapped in and out and configured by an external policy for use in different systems.

Consider the labeling of input from stdio. In a UNIX system, some security mechanisms are available for authenticating the user who is running the program against
the system’s user database. In this case, the policy decision may be to label the data from standard input with the user’s login principal. As another example, in a mandatory access control (MAC) operating system, data from different terminal windows may carry different labels. These system labels must be translated into application labels that can be monitored by the application’s information flow policy. This translation involves policy decisions. For the sake of portability, these policy decisions must not be hard-wired in the code, but configurable through an external policy.

Furthermore, the policy decisions made for an application do not occur in a vacuum, but must correspond to overall security goals for a system. In order to ensure that an application faithfully implements system policy, some policy analysis is needed for comparing application-level policies to system-level policies.

This example has highlighted the need for policy-based principal infrastructure and runtime infrastructure as well as policy analysis when constructing modular, portable applications that verifiably enforce separable and expressive security policies.

1.3 Contributions

![Fig. 1.2. Verifiable implementation of lattice policies (abbreviated “IF latt.” here) is insufficient for bridging the semantic gap between security policy semantics and operational semantics. Adding principal (JP) infrastructure and runtime (SP) infrastructure supports systems principles of modularity, portability, policy expressiveness and policy separation to bridge the semantic gap.](image)

The contributions in this work are illustrated broadly in Figure 1.2 and summarized concisely in Figure 1.3. Namely, prior to this work, STL technology provided the ability to automatically verify enforcement of information flows in applications, but with limited or no communication to outside systems and with limited policy. To use this technology to bridge the semantic gap between high-level policy specification and its implementation in a computer system, it is necessary to add infrastructure for making the technology modular and portable with expressive, separable policy. We provide the
<table>
<thead>
<tr>
<th>General scientific advances</th>
<th>Tools and Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trusted Declassification</td>
<td>First application: JPmail</td>
</tr>
<tr>
<td>Architecture for integration with MAC operating systems</td>
<td>JPolicy system</td>
</tr>
<tr>
<td>Design principles for STL IDEs</td>
<td>SIESTA SELinux tool</td>
</tr>
<tr>
<td>Design principles for STL Runtime Systems</td>
<td>Jif system utility (logrotate)</td>
</tr>
<tr>
<td>Information flow policy compliance analysis</td>
<td>Jifclipse plugin</td>
</tr>
<tr>
<td>SELinux MLS formalism</td>
<td>Channels; SPolicy system</td>
</tr>
<tr>
<td></td>
<td>Jif system security tool (FlowWall)</td>
</tr>
<tr>
<td></td>
<td>Compliance analysis tool (PALMS)</td>
</tr>
</tbody>
</table>

Fig. 1.3. Concise listing of contributions, consisting of both general scientific advances with applicability to STLs generally, as well as tools specific to Jif and the operating systems we study in this thesis.

shaded infrastructure illustrated in the figure, along with underlying theory and analysis that captures its security properties, and use it for building some practical applications. Namely, we provide trusted declassifiers (part of the JP principal infrastructure) and channels (part of the SP infrastructure) whose policies are separated out and controlled through a high-level application policy (JPolicy and SPolicy, respectively). The application policy can be analyzed against system policy for compliance.

We identify the following contributions:

- **Practical STL applications.** The JPmail email client was the first real-world application developed with an STL. It represents a milestone in moving STL technology from theory to practice, and exposes many practical challenges when using STLs to build real systems. We discuss this further in Chapter 3. Additionally, we employ systems methodologies to construct system utilities (see Chapter 5) and security infrastructure (see Chapter 7) that improve the modularity of STLs, enabling the construction of applications that function as secure components in heterogeneous systems. Finally, our investigations motivated the development of principles for practical software engineering tools for assisting in the construction of practical applications. These principles and our implementation of them for Jif are described in Chapter 3.

- **Trusted declassification: an expressive, separable policy.** Because the standard security property enforced by STLs is too strong for many applications, expressiveness requires that this policy be relaxed through declassification. Although declassification mechanisms already exist, none could be controlled through an external policy. In order to relax the policy while still maintaining a separable policy system, we developed a new mechanism for declassification, called trusted declassification that marks some methods as trusted declassifiers and defers the declassification decision to high-level policy. We formally prove the security properties enforced by trusted declassification. This is discussed in Chapter 4.
• **Integration of STL applications with diverse systems.** In this systems work, we implement applications that utilize diverse security mechanisms provided by different operating environments. While JPmail was originally implemented with a public key infrastructure (described in Chapter 3), we demonstrate portability by re-implementing it to interact with a different set of systems security protections provided by a mandatory access control (MAC) operating system. MAC operating systems label all resources and processes and use a reference monitor to govern all interactions through a mandatory, system-wide policy. They cannot control the internal behavior of applications, however. We extend MAC system security goals into applications through APIs and services that integrate STLs and MAC OS’s. We discuss this result in Chapter 5.

• **Policy compliance analysis.** Separating policy specifications out from program code makes it possible to reason about policy apart from the applications in which it is used. This is especially helpful when seeking to understand how an application’s policy interacts with system policy. To this end, we develop a formal specification of MLS policy in the commodity operating system, SELinux, and formally define an algorithm for determining that an STL application policy is compliant with a system information flow policy. We implement our approach as a tool, called PALMS, that carries out this analysis for SELinux and Jif. We describe this in Chapter 6.

• **Channels: portable runtime infrastructure with separable policy.** Channels connecting application data to system resources are a critical policy decision point in STL applications, and can have subtle security properties. While portability demands that interactions with diverse operating systems environments be possible, separability demands that these policy decisions be captured in an external policy. To meet this need, we develop a general systems architecture for dynamic labeling of I/O in STL applications. We provide principles for developing portable runtime infrastructure with separable policy along with a concrete implementation for Jif, a Channel abstraction that can be controlled through high-level policy. More information is given in Chapter 7.

### 1.4 Dissertation Outline

Our goals of producing diverse, practical applications that enforce diverse, expressive policies in a way that is portable to diverse environments provide useful categories for organizing the remainder of this work. Throughout each category, we seek to maintain the separability of policy. We break down the problems into three main categories, shown diagrammatically in the roadmap in Figure 1.4, 1) supporting diverse policies, 2) supporting diverse environments, and 3) reconciling policies. In each case, we are guided by practical application development. As we move through the roadmap, we explore the extent to which STLs can be used to develop modular, portable applications that verifiably enforce separable, expressive security policies.

We begin in Chapter 3 with a realistic application, an email client, implemented for a particular environment, one with PKI-based principals. This application serves to
explore the dimensions of modularity and policy expressiveness. The principle of separability motivates drawing out the principal declarations and relationships from this application and leads to the development of two new technologies, a principal infrastructure we call JPolicy and a mechanism for declassification, trusted declassification. We describe the details of the principal infrastructure and the security properties of trusted declassification in Chapter 4. Moreover, experience building realistic applications provided principles for developing practical software development tools. We describe these principles and Jifclipse, an implementation of them for Jif, in Chapter 3.

In Chapter 5, we explore the dimension of portability by studying the interactions of STLs with a new environment, a mandatory access control (MAC) operating system. For this, we implement a new application, a system utility for rotating log files, and we also revisit our email client for continuity, recasting it in the new setting. Once again, our focus is to separate policy specification from implementation. In this case, we discover the necessity of being able to analyze an application’s high-level policy to compare it with a system security policy. This led to the development of a policy compliance algorithm and a tool called PALMS to carry out this analysis automatically, described in Chapter 6. Furthermore, in order for an operating system to validate its trust in an application’s security, it must check the application before execution. We provide a MAC/STL architectural design and implementation of an OS service, SIESTA, for this purpose.

In our final experiment, we explore the diversity of policies that can be enforced with STLs, especially by configuring the policy to use principals and principal relationships in different ways. For this experiment, we provide another new application, an information flow-controlled firewall, called FlowWall. Once again seeking to separate policy specification from implementation, this experiment motivated the factoring out of channel labeling policy. The system policy infrastructure, SPolicy, provides key support for portable and separable STL application policy. This is described in Chapter 7.
Fig. 1.4. Roadmap: the dissertation covers each of the listed subjects, broken down into three main categories.
Chapter 2

Background and Related Work

Security-typed languages grow out of decades of research in information flow policies going back to work by Bell and La Padula [BL73b, BL75]. Bell and La Padula developed a model for multi-level security (MLS) for an implementation in the MULTICS operating system. From the beginning, information flow policies were developed for the sake of enforcement in real systems [SCSW78, Org72]. Security-typed languages developed from the insight by Volpano, Smith, and Irvine [VSI96] that information flow policies could be encoded as annotations on types and determining a program’s security could be reduced to determining type soundness. Much of the work that followed has been theoretical, but some work has led to practical languages. These have been used for developing only a few applications.

There are three main areas of relevance. The first area regards security policies, particularly as they relate to information flow policies. Secondly, we consider how information flow policies are enforced in real systems, with a special consideration on the place of SELinux, a commodity operating system we use in this work. Finally, we look at security-typed languages, with a particular focus on Jif. In the section on STLs we provide some details about the Jif language that will be important for the remaining chapters.

2.1 Policy Systems

The need for having an explicit policy to govern the security of networked systems and electronic data is becoming more widely accepted. From protection of healthcare data [oHotS03] to financial records [How03, Kme03] to a University’s institutional data [PSU04, PSU02] to web privacy [CLM+02] to national security [Dep03, Dep05, Che05], many and diverse organizations are seeking to specify and enforce various security policies. Such high-level specifications of security policy for systems architecture are certainly important [CMS94, Lan81]. At the heart of these security policies is a description of what is needed by an organization in order to deem their software systems trustworthy to handle their sensitive data. In particular, in order for a software system to be deemed trustworthy, three things must be clear: what the system should do, what it should not do and what it actually does.
From the massive tangle of policy models, one model has remained an active area of research and development throughout, the *information flow policy model*. Recently this model has experienced a bit of a renaissance as it finds a home in commodity MAC operating systems [SSL+99, Fou, VMD07] and is implemented in emerging security-typed languages. In this section, we explore the design space of policy models and culminate in a look at information flow policy.

One of the most basic policies is the Access Control Matrix proposed by Lampson [Lam71]. In this policy, a matrix is constructed using all possible objects and all possible subjects (or principals) and the values stored in the matrix are the rights of that subject over that object (e.g. read, write, execute). This can be implemented more efficiently using Access Control Lists (ACLs) by attaching lists of subjects and rights to objects or conversely using capability lists, essentially attaching objects and rights to subjects. This approach is simple, clear and effective and is still used in many commodity operating systems, especially in file systems.

### 2.1.1 Improving administration

A major development in policy languages came in the introduction of role-based access control (RBAC). RBAC was introduced by Ferraiolo and Kuhn [FK92] as an alternative to Mandatory Access Controls (MAC) and Discretionary Access Controls (DAC) for use in the civilian sector. A key semantic development in RBAC is the attachment of permissions to roles rather than individuals. Recognizing that individuals change roles more often than roles change rights, this improvement makes the system more flexible and improves administration. RBAC is a hierarchical model which aids in management of security policies, especially in naturally hierarchical domains such as financial, health care and military settings. The role hierarchy in RBAC is a partial order meaning that if a role $r_1 \sqsubseteq r_2$ then $r_2$ has all the permissions of $r_1$ in addition to its own permissions. RBAC is flexible and robust, able to be centrally managed and also able to implement a wide variety of security models [San93, San96, OSM00].

RBAC has found broad acceptance in the security community. Since the seminal work, RBAC model has been explored extensively and extended with features that improve power and expressibility. The proposed standard [FSG+01] was recently accepted by ANSI.

RBAC has been used in a wide variety of contexts. It has been implemented for use with the Web [FBK99], has been used to implement mediation policy for multi-domain environments [JBBG04], has been studied in healthcare environments [BBB97, WFSM02, SDA05], and has been used in banking [Mar02]. The model has been extended with rules for enterprise environments in a system called ERBAC [Ker02, KKKR04, KW05]. It is best suited to centralized organizations, since the meaning of roles may not transfer correctly between multiple, cooperating organizations. For example, a *manager* in one company may not have analogous permissions in another company.

### 2.1.2 Improving expressiveness

In addition to mere access controls, various other properties are desirable in different settings. For example, in some settings it is important to indicate that a subject
is obliged to act in a certain way. A policy might state that on an application startup, a certain series of initializations must take place. At a higher level, a bank teller might be obliged to submit a receipt for each transaction. For this setting, the notion of obligations was developed [Ort98]. Obligations indicate that a certain action must take place. They express a richer security policy than merely optional permissions. In fact, Schneider has shown that such policies are too rich to be enforced with runtime monitors [Sch00], which cannot foresee future execution paths. Ortalo formalized the notion of obligations by using deontic logic. He developed an expressive policy model for handling existing security policies, such as for a small bank. Cholvy, et al. analyzed the combination of policies involving obligations and showed safe ways to reconcile conflicts [CCSC98].

Ribeiro, et. al. [RZFG01b, dCR02] developed the first language which not only expresses, but also enforces [RZFG01a] obligations in security policy. In accord with Schneider’s result showing the theoretical limitations of capturing obligations using security monitors, however, they agree that they can only capture a limited form of obligations by using transactions. They use their framework to express and enforce other powerful policies such as DAC and MAC, but they are only able to capture a limited form of information-flow policy. In their research, they took a top-down approach, beginning from the policy language and then developing an enforcement mechanism for the language.

More recently, Damianou, et. al. developed Ponder [DDLS01, SL02], a robust language, building on many features of these previous languages. They attempt to reverse the trend towards a multiplication of policy languages by developing a universal policy language. They claim that their language is able to express all the desirable features of system-level policy in a way that is scalable, extensible and easily analyzable. They developed Ponder using a declarative language that is composable and easy to analyze and can scale up to millions of principals by using conglomeration and composition. Ponder is able to capture complex policies, including obligations and delegations. It is rule-based and stored separately from the services it governs, thus allowing it to be more easily adapted to different contexts. It expresses constraints, such as separation of duties, through meta policies which can be composed. It enforces policies by compiling them into applications and services, using different backends for applications, such as firewalls, Windows security templates and Java applications. The authors of this work were also trying to address the semantic gap, but took a fundamentally different approach by trying to develop a universal language that could be compiled for various, specific settings. This approach is also more limited in that it cannot capture the very fine-grained information-flow policies enforceable by security-typed languages.

Other features that have been built into policy languages include temporal constraints [JSGB03, BJBG04], quality of service (QoS) constraints, separation-of-duties (SoD) constraints [JSGB03, GGF98], context constraints [GMPT01, McD03, WFSM02], temporal constraints [BJBG04, JSGB03, Jos03] and workflow constraints [BFA99]. All of these add expressibility for policy which is needed in different contexts (including health care, financial, military, etc.).

Kern et al. have developed a system for handling the kinds of expressiveness that are useful for applications security [KKKR04, KW05], in contrast to file system and networking security. Applications require more expressive permissions than merely
read, write and execute. Thus, Kern’s system uses rule-based policies such as that a
particular teller has a limit of a withdrawal of $20,000. By using rules, they are able to
parameterize permissions and make the allowance or denial more fine-grained ($20,000
could be changed to $40,000, e.g.) They dub this rule-based RBAC (RB-RBAC). In later
work, they combine it with Enterprise RBAC (ERBAC) [Ker02] so that it can handle
decentralized security policies and be used in an enterprise environment. The expressivity
of the policies and the granularity they use for processes are both qualities that make
their work applicable to ours. Their parameterization of permissions is especially useful
in our own setting and something we plan to investigate more deeply.

2.1.3 Highly decentralized policy

Many of the above systems take a monolithic approach, having a centralized policy
database which can be polled directly from all points within a system. This may not
be possible in a highly decentralized environment, such as the Internet. This has led
to the development of decentralized policy models such as public-key infrastructures,
which provide certificates of authority to widely distributed principals. In this way the
certificates can be vetted asynchronously and more decentralized environments can be
supported.

This was formalized into a security policy model through trust management (TM)
systems. TM provides a framework for building security checks into applications. By
placing hooks in applications, credentials can be checked before authorizing certain op-
erations. The credentials may be chains of delegated authority that must ultimately be
certified by some main authority. One advantage of TM is that it provides a security
infrastructure for checking authorizations in a distributed setting. The TM functions
are able to perform the necessary credential chain verifications to ensure that proper
authority has been granted to the acting principal. On the other hand, TM requires the
programmer to do the hard work of figuring out where to place security checks within
applications. The application developer must have sufficient awareness of security con-
cerns to place policies (or pose security questions) explicitly in the code. TM also places
a great deal of importance in a centralized or a small group of centralized authorities,
called certification authorities (CA).

PolicyMaker was the first system designed for this [BFL96]. It included a policy
language and Application Programming Interface (API) which allowed its integration
into applications. It also included a system to make decisions about policy and enforce
it at runtime by checking credentials. This system was improved and developed into the
KeyNote Trust-Management System [BFIK99] which was standardized and stabilized
in RFC 2704. This system was used to build two robust applications—one for handling
policy in IPsec [BIK02] and another for handling policy in an Apache web server [BIK03].

The SD3 Trust Management System [Jim01] uses a formal, logic-based language
to make policy decisions. A key feature of SD3 is that the proofs generated for pol-
icy decisions can be automatically verified by a proof checker, improving confidence of
correctness and reducing the Trusted Computing Base. This system is as robust as pre-
ceding systems, demonstrated by its ability to implement the necessary security checks
for a version of DNSsec.
Another recent line of research is in the RT family of role-based, trust-management languages [LM03]. These languages capitalize on extensive research in RBAC and bring that expressivity to bear in the domain of trust management. They express policies in a logical format that allows for efficient management of policies and checking of queries [LWM03]. RT policies can be checked efficiently for compliance with constraints such as SoD [LBT04]. Furthermore, Li and Tripunitara have shown [TL04] that a variant of RT is more powerful than the most common RBAC language under the most common administration model, ARBAC97 [SBM99].

Although Zhang et al. [ZAC02, ZAC03] do not identify themselves under the category of trust management, their policy definition and enforcement system bears many resemblances to RT. They are distinguished from KeyNote because they use role-based techniques (like a role hierarchy), but this is very similar to RT. Like KeyNote, they have implemented a system which can answer policy questions according to policies defined in a declarative, rule-based language. Their language is RDM2000. They have used their system to implement prototypes for a healthcare environment as well as a law enforcement environment.

2.1.4 Formal analysis with policy languages

One major advantage of having a high-level policy language with a clearly defined semantics is that it thus becomes subject to automated reasoning. We can construct tools to make automatic inference about certain properties being held or not held in a particular high-level policy. This is particularly powerful in logic-based policy languages [Aba03] such as SD3 [Jim01], RT [LM03], RDM2000 [ZAC02] and Binder [DeT02].

Li et al. have developed a security analysis for RT [LMW] and RBAC [LT04] using logic programming. Drouineaud et al. perform automated formal verification for an RBAC policy use Linear Temporal Logic which they embedded in the Isabelle/HOL theorem prover. These languages can also be analyzed with regard to certain properties, e.g. separation of duties. Li et al. show some theoretical limits on the efficiency of automated algorithms for such analyses [LWM03]. Another area of formal analysis is with privacy policy languages. Barth and Mitchell used modal logic to analyze privacy preference and policy languages. They implement their analysis with a tool that can determine when one policy enforces another [BM05]. Finally, McDaniel et al. axiomatized a group policy reconciliation language called Ismene [MP00], and developed the Antigone system to enforce it [McD01]. With this, they were able to prove the intractibility of general group policy reconciliation.

Analyzing SELinux policy  SELinux plays a particularly important role in this work as an exemplary MAC operating system with a particularly rich policy language. Because the policy language is so expressive, a great number of analyses have been developed to understand whether basic security properties are enforced by a particular policy. We give a brief overview of these here. We provide our own analysis for SELinux MLS policy in Chapter 6.

Some frameworks developed to help in the analysis of SELinux security policies include Gokyo [JEZ02, JEZ03], SLAT [GHRS05], PAL [SSS04], APOL [Tec] and
SELAC [ZM04]. Gokyo assesses access control policies based on Access Control Spaces; such spaces define sets of assigned permissions (prohibited, permissible, and unknown spaces). Their approach was used to evaluate the integrity of the Apache web server in the context of the entire SELinux policy. More precisely, they determined whether low integrity subjects (subjects outside Apache who are not high security trusted subjects) were allowed to write data that the Apache administrator would read.

Another framework is SLAT (Security Enhanced Linux Analysis Tool) [GHRS05]. It provides a systematic scheme for defining OS security goals. They define that a system’s security goals depend on the configuration of the system and the interaction between the system and trusted pieces of software with their goal being that only a small set of programs should be granted such privileges. SLAT also contains an implementation, using model checking.

Sarna-Sota and Stoller [SSS04] used the information flow model defined in SLAT to implement another framework for analyzing configuration policies in SELinux; it is called PAL (Policy Analysis using Logic Programming). PAL creates a logic program based on an SELinux policy that make it possible to run queries to analyze the policy. PAL is implemented in XSB [xsb], a logic-programming system based on tabled resolution. They find that using queries based on logic programming makes the system more flexible and easy to use than the system build in SLAT.

APOL [Tec] is a tool developed by Tresys Technology to analyze SELinux configuration policies. Among its multiple features it includes forward and reverse domain transition analyses, direct and transitive information flow analysis, relabel analysis and type relationship analysis.

Zanin and Mancini [ZM04] define a formal framework called SELAC (Security Enhanced Linux Access Control) for analyzing an SELinux policy configuration. They define semantics for the various SELinux policy rules, including how they interact to allow or prevent operations in a live system. Using their semantics, they define accessibility spaces [JEZ02, JEZ03] and use these to describe an algorithm for finding what subject-object accesses are legitimate for a given policy.

None of these approaches handle the SELinux MLS labels in any way. Each of the analyses above deals only with Type Enforcement (TE) policies, and the effects these policies have on the system. MLS in the context of SELinux MAC enforcement may or may not conform to the formal descriptions and properties (∗-property, simple security condition, etc.) historically given in literature. Consequently, we develop a methodology for analyzing SELinux, and extend it to form a good functional analysis of SELinux MLS policies. In doing so, we seek to establish a practical method of validating the information flows present in a given SELinux policy, not simply a workable formalism. We present this in Chapter 6.

2.1.5 Information flow policy

The concept of information-flow control is well established. Despite the power and robustness of the systems and languages described above, all of them lack the expressive power available to information-flow policies. Information-flow policies govern not only information release (like access control), but also control information propagation. An
information-flow policy will only release sensitive information to a process which is able to maintain the same information flow policy. This leads to the powerful end-to-end property of noninterference. To recall, noninterference is the property that highly confidential data never influences low confidentiality outputs (i.e., that no sensitive data is ever leaked explicitly or implicitly). Dually, low integrity inputs cannot influence high integrity outputs. As an example of the practical importance of such policies, the theft of more than 100 Top Secret documents was recently reported. This theft was carried out by Leandro Aragoncillo, a White House official with Top Secret authorization. While his access to the secret information was legitimate, he never should have been allowed to propagate that information to unauthorized parties, especially over an insecure channel like Email. Yet, that is exactly what he did [RE05].

2.1.6 MLS security model

Multi-level security was formalized by Bell and LaPadula [BL75] in order to control how information is allowed to flow between subjects in a system. These subjects are given a sensitivity level, or security clearance, and objects are also given a similar security classification. MLS policies attempt to restrict how information may flow between designated sensitivities. As an example, consider a military application with 4 sensitivities, ordered from least to most sensitive: Unclassified (UC), Confidential (CO), Secret (S), and Top Secret (TS). In this case, TS dominates S. Note that in this example the sensitivities form a total ordering; each sensitivity is either higher, lower, or equal to another. This is not always the case.

Typically, MLS defines information flow policies, based on two properties: the simple security condition and the ⋆-property. The simple security property, sometimes described as “no read up” requires a subject S to dominate an object O to have read rights, meaning the subject’s security clearance dominates the object’s security classification. The ⋆-property, described as “no write down” requires the object’s classification to dominate the subject’s clearance for the subject to have write rights.

To allow finer granularity of information control than just a few sensitivity levels, the MLS model was expanded by adding categories to the security level. These categories serve to group information of the same kind so that access may only be granted to subjects on a need-to-know basis. Categories provide a way to allow access to certain types of data, while staying within the confines of the sensitivity restrictions. A subject must then have a superset of the object’s categories to dominate the object. To illustrate this, let us take subject S with sensitivity Secret and categories {Nuclear, Military, Domestic}, and object O with sensitivity Confidential and {Military, Domestic} as categories. Since S has a higher sensitivity and a superset of O’s categories, it is said to dominate O, and O is said to be dominated by S. In nearly every practical MLS policy, this would equate to subject S being able to read from object O. Note that if S did not have Domestic as a category, it would no longer dominate O; the two would be incomparable.

Security lattices Denning and Denning [DD77] introduced lattice-based information flow policies, as a generalization of the Bell-LaPadula model. These models arrange
labels on data as a lattice of principals, sometimes called a principal hierarchy. The
traditional model [BL75, Den76] allows data only to flow up the lattice (i.e. data can
become more secure, but not less secure). If, for some reason, data must flow down the
lattice, a declassification must take place. These policy violations should be infrequent
or non-existent and if occurring at all should be carefully regulated. Filters for regulating
declassification are called declassifiers.

Lattices may have a variety of principals and structures. A standard military
lattice is simply a vertical line containing five levels: unclassified, classified, confidential,
secret and top secret, with top secret placed on the top of the lattice and unclassified
on the bottom. While unclassified data can be written to classified files, the opposite
is not true. This is also expanded with category sets that capture the notion of “need-
to-know.” Although an individual may have secret clearance, there may be no need for
a nuclear power engineer to know about secret, domestic information. This is shown in
Figure 2.1.

![Security Lattice Diagram](image)

**Fig. 2.1.** A security lattice for two need-to-know category sets. Nuclear secrets may not flow
to domestic secret compartments, for example, but nuclear classified data may flow to nuclear secret compartments.

### 2.2 Systems Security

The first information flow policy was designed to capture the security model used
in government as it should be implemented in a strongly secure operating system. From
the beginning, the policy was designed to be usable and corresponding enforcement mechan-
isms were built in the target operating system, MULTICS. As theory met practice,
successive versions of the work [BL73b, BL73a, BL75] included practical developments
(such as trusted subjects) for increased expressiveness and flexibility.

In this section we survey information flow policies as they have been implemented
in real systems. This primarily includes systems with MAC, which have been the most
common mechanism for implementing information-flow control.
2.2.1 Mandatory access controls

MAC security utilizes the reference monitor concept given by Anderson [And72], which requires that all subjects and objects be labeled and all security-sensitive operations are hooked with runtime checks. These checks query a previously configured security policy to determine whether the operation is allowed, based on the subject and object labels. Security in the MULTICS operating system [Org72, SCSW78, KS02] was built around this reference monitor concept.

Recently, MAC has made its way into commodity operating systems. An NSA project integrated the Flask architecture [SSL+99] into the Linux operating system through the Linux Security Module (LSM), giving Security Enhanced (SE) Linux [SVS01]. This is now being shipped as part of the mainstream kernel in the 2.6 series and turned on by default in Redhat distributions since Fedora Core 5. Other work in operating systems with MAC security include Trusted Solaris [Micb], Solaris Trusted Extensions [Mica], TrustedBSD [Fou] and SEDarwin [VMD07].

MAC policy is not restricted to expressing information flow policies; it is often used to enforce least privilege. The goal there is to limit applications to the least set of rights necessary to properly carry out their function [LSM+98].

MAC policies have been used to implement various high-level information flow properties across a distributed system. SELinux policy achieves this through detailed security labels that capture information about the user, role, type and MLS level on particular processes and operating system resources. Furthermore, the policies are quite granular, regulating access to data not just at the level of read, write and execute, but also getting attributes, setting attributes, renaming files, opening sockets, listening to sockets, deleting files, etc. Recent research has shown that this basic mediation (called Type Enforcement or TE) can be used to enforce integrity constraints on data [JEZ03]. More recently added multi-level security (MLS) labels can be used to enforce confidentiality [Han06].

MAC makes it possible to “sand-box” applications and prevent them from doing anything malicious. Thus, a web server can be limited to accessing its own files or protected from reading in malicious data on low-security sockets. A system resource such as the password database can be protected from being accessed by any program other than the passwd program. logrotate can be prevented from leaking logs to user accounts.

A major limitation in all this work is that OS-level MAC cannot peer into an application. As soon as it allows the application to handle multiple levels of resources, it cannot validate the information flows that pass through that application. For example, the passwd program is allowed to read from the password database and write to stdout. The operating system is unable to prevent it from writing the passwords to stdout. Likewise, although logrotate cannot write logs to user accounts, nothing prevents it from mailing them to the user. We address this limitation using STLs in Chapter 5.

Label-based approaches The Common Criteria [CC05] system for evaluating security identifies the strong security benefits of label-based approaches.
A privacy system of note is PCASSO [BBB97, BMJB99, MBBC02]. It is remarkable for a few reasons. One is that it was implemented on a large scale in the UCSD medical system and so it is able to give usability reports based on years of practical experience. Another is that it is the first system to allow patients to view their own medical records via a web interface. Finally, it is notable for using some Department of Defense strong assurance mechanisms to achieve a high degree of security. In particular, it uses a label-based approach to achieve a kind of multi-level security. Some strong separations of data are enforced between different groups of patients and doctors.

More recently, the operating system Asbestos [EKV+05] was built to handle exceptional situations, such as applications that handle multiple levels of data and applications that perform downgrading by using a MAC system, with labels similar to those used in the STL, Jif. Declassifications are monitored in a more decentralized fashion, similar to Jif. Though still in preliminary stages, this work provides promise for future integration with STLs.

2.2.2 Application-level MAC

Because operating system-based reference monitors can only enforce policies at the level of system calls, a technology was developed for pushing reference monitors inside applications. Inlined reference monitors [ES99, ES00] utilize the same concepts as system-level reference monitors, wrapping security sensitive operations with guards and forcing policy checks to be made each time such operations are called.

The idea was further developed by Bauer et al. with the Polymer security mechanism [BLW05]. This mechanism not only allows very robust security policies, but also implements policies as first-class objections which can be queried or utilized within higher level policies. The authors develop a concept of edit automata [LBW05] which allow an additional expressiveness for the security monitor. In addition to being able to accept and deny actions, the monitor can also perform its own action, perhaps editing the request, auditing the operation, raising an alarm or allowing only part of the request.

Edit automata are provably more expressive than normal reference monitors, but still are not able to express the full range of information-flow policies [BLW02]. The methodology used by this system, however, is moving in the direction of our own methodology. By making policies into first-class objects and allowing them to be composed, this facilitates the task of developing complex policies at the application level. In contrast to our approach, however, these policies are not able to express the fine granularity of information flows captured by security-typed languages. Furthermore, they have not tried to express higher, system-level policies. This work will undoubtedly provide further insight for our own application-level research, however.

A major deficiency in this work, as in implementations of the reference monitor concept at the OS-level, is the lack of automated tools for determining whether a reference monitor implementation actually has the property of complete mediation. This is complicated, in part, by the fact that “security-sensitive operation” may be defined differently in different contexts. STLs excel in this regard, because security-sensitive operations are defined as information flow-related operations and because the type checker ensures that all these operations are checked.
This deficiency applies also to recent work that seeks to extend MAC into applications through userspace object managers [Wal07, Car07], as well as work to retrofit legacy applications, such as the X-Windows Server, with reference monitor hooks [GJJ06, GKJ07].

2.3 Security-Typed Languages

Volpano, Smith and Irvine [VSI96] were the first to make the connection between types and information flow and to propose a security-type system. They explored a small, imperative language with \texttt{while}-loops and conditionals and showed how a proof of noninterference could be made isomorphic to a proof of type soundness. The field of language-based security has since expanded in many ways.

Jif [MZZ+] and its predecessor J-Flow [Mye99a] extend the Java language [GJSB05] with security typing. They also provide language features needed to implement real systems, such as a decentralized label model (DLM), label polymorphism, and runtime principals. To date, the analysis of implicit flows in Jif has been restricted to control flows, leaving timing [Koc96, BB03] and termination [VS97] flow analysis to future work. Existing almost exclusively within the domain of formal programming languages, ongoing investigations are exploring security-typed variants of Caml [Sim03a, PS02], assembly language [BCM05], Ada [KHCP00, WDMR04, Dav05] and web scripting [LZ05b], as well as language features for interactive programs [OCC06], multiple threads of execution [SV98], functional programming [HR98, Zda02, PS02], and distributed systems [Man00, Man02, MS01, MS03, SM02]. Other work has studied integrity information flow [Bib77, CM06] in the context of replication and partitioning [ZZNM02, ZCMZ03], downgrading [LZ05c], and in web scripting [LZ05b]. Much of this is documented in the survey by Sabelfeld and Myers [SM03]. More recently, the language Sj was developed for handling dynamic policies [Tse07]. Its practicality is demonstrated through using it to build a proof-of-concept medical database application that implements many of the principles of Anderson’s medical system security model [And96, And00].

Sabelfeld and Myers identify the following major areas of development so far: language expressiveness, concurrency, covert channels, and security policies. They note the lack of practical experience. One of the relevant areas in which they indicate the need for future work is “practical issues”. Here they state,

\begin{quote}
Efforts spent on accommodating richer languages and modeling elaborate attacks should be supported by the investigation of the impact on the restrictiveness for a programmer. The question is whether it is, in the first place, possible to write efficient secure programs that do not violate security requirements. (p. 9)
\end{quote}

This is, of course, precisely what we set out to investigate in this research.

2.3.1 Information flow control

The first full-featured system that enforces information-flow control using language-based techniques is Jif [MZZ+]. Jif is based on the decentralized label model (DLM)
developed by Myers and Liskov [ML00]. The DLM is a lattice-based model which can simulate many other security policies. Because it is decentralized, it is also particularly suited for distributed systems. In this section, we will first consider information-flow policies, motivating their importance and power of expression. Then we will consider the DLM, especially focusing on how well it could serve for simulating other high-level policies.

Dynamic methods of enforcing information flow policies suffer in size and speed. Additionally, they are unable to monitor more subtle leaks of information, such as implicit flows. Implicit flows are leakages of information through control structures. These leaks can be caught using static analysis, however. Static analysis tools have access to the program code and can make inference about all possible program execution paths. Schneider proved this using automata to formally analyze the expressive power (and limitations) of enforcement monitors (EMs) [Sch00]. Because they are unable to see into the future, EMs are fundamentally limited. Information flow policies, on the other hand, depend on static analysis for which all possible program paths can be considered. Although it must be conservative in tracking implicit flows, such a policy is much less conservative than a runtime mechanism (such as an EM) must be.

Consider the following simple example:

Listing 2.1. Simple example illustrating label creep.

```
1  if (secret\textsubscript{high} == true)
2      then pub\textsubscript{low} = true; // implicit flow
3      else pub\textsubscript{low} = false;
4  print\textsubscript{low}(pub\textsubscript{low}); // leak
5  pub\textsubscript{low} = true; // no leak, but dynamic checking disallows this
```

In this code, the variable secret is labeled as high sensitivity and the variable pub is labeled as low sensitivity. Although there is no direct leak from secret to pub, there is an implicit leak in which the secret, high-sensitivity value can be determined from the public, low-sensitivity value by checking the value of pub that is printed out. In order to protect against this leak, all the values in the body of the conditional must be treated as high sensitivity in order to protect the high-sensitivity guard. This is overly conservative, since there need not be information loss in every case. The advantage of the static analysis comes into play, however, in realizing that things return to normal after the conditional. Thus, the final statement on line 5 should be a valid assignment, because it is no longer able to leak information implicitly about the guard. This cannot be determined at runtime, however, because it is not clear, without the source code, where the conditional ends. To be careful, a runtime system would have to use a process sensitivity label [LB73] to keep track of the implicit flow. The problem then is that the label grows monotonically until all values are considered high security. This phenomenon, called “label creep”, makes such systems extremely impractical.

Thus, the strong, end-to-end security guarantees promised by information-flow policies can best be enforced by building them into a system that performs static analysis. In this light, security-typed languages, which are able to implement this static analysis through type checking, are a particularly appropriate mechanism for enforcing
information-flow control. By tagging a variable’s type with a security level, the type system can ensure that there is no interaction between a high-security variable and a low-security variable. In this way, the type system can guarantee noninterference in a very efficient way at compile-time.

### 2.3.2 The Decentralized Label Model (DLM)

The DLM was developed by Myers and Liskov [ML97, ML98, Mye99b, ML00] as a model for information-flow control. The DLM protects confidentiality and integrity (additional labelings for integrity and availability were later proposed, but have not yet been implemented [Mye99b, LMZ03, LZ05c, ZM05]) by labeling variables with policies that express the security concerns of the principal(s) which own the data. The principals are arranged in a principal hierarchy that is structured by acts-for relations between principals. This allows for delegation of authority and arrangement of principals into groups and roles.

**Principals** Principals can be individuals such as Alice or Bob or they can be roles, such as Teller or Auditor or groups such as Student or Professor. In the context of availability policies, principals can represent certain conditions under which data will be available. For example, host1 could be a principal meaning that host 1 remains live. Another principal, puzzle might indicate that so long as foreign machines are unable to break a rate-limiting puzzle, then a distributed denial of service (DDoS) attack cannot compromise data availability.

**Labels and Jif features** We describe here some Jif syntax which will be used throughout. Jif uses the decentralized label model (DLM) [ML98] to specify security labels. Labels in Jif consist of confidentiality policies and integrity policies and form a lattice. In the DLM, the join of two labels L1 and L2 is written \{L1 ; L2\}. If P is a principal, then the label with confidentiality policy \{P;\} indicates data that can only flow to other data labeled L where \{P;\} \leq L, while the label with integrity policy \{P!:\} indicates data that can only flow to data labeled with L where \{P!:\} \leq L. Principals can be arranged in a hierarchy which induces an ordering on labels, such that \{P;\} \leq \{Q;\} if P \leq Q. The least confidential (i.e. public) principal is _ and the least confidential label is \{_;_;\}, while the most confidential principal is *. The DLM also allows for reader and writer lists on labels, but they were never needed for the applications developed in our work.

Jif allows classes to be parameterized with a principal or a label to provide additional static checking on mutable fields. For example, a Student object containing secret fields annotated with \{Alice;\} is instantiated as Student[Alice]. Another important note for our examples is that arrays of parameterized classes will have two pairs of square braces and two labels. The first pair of square braces signifies the a security annotation parameter and the second indicates the type is an array. Consider an array of Question’s, Question[Examiner]{Examiner:}{Examiner;}[]. The structure of this array is public (i.e. anyone can see the maximum length of the array), but each element is parameterized with Examiner and the existence of each array element (i.e., that it is not null) is considered \{Examiner;\}-level information.
Jif supports features for dynamic labels. Labels can be constructed dynamically from principals with `new label` statements such as `label l = new label{p;}`. Labels can also be compared with the `<=` operator which depends in turn on the ordering of principals, which can be compared with the `actsfor` operator. The statement `l1 <= l2` returns true only when `l2` is more secret than `l1` and `l1` has higher integrity than `l2`. Label comparisons release information about the labels themselves and so labels must be labeled with meta-policies. Special syntax accesses the label on the label.

Listing 2.2. Code snippet illustrating labels on dynamic labels.

```java
label{_:} lb1 = new label{*:}; // a high security label guarded with a public label
int{*lb1} i = 10; // i is given the label {*:}
int{lb1} j = i; // error; j is given the label {_:}, and {_:} <= {*:}
```

Declassification is a necessary part of any practical information-flow control system. Noninterference is too strict a policy for real applications. For example, a password check releases a very small amount of information about the confidential password (whether or not it is equal to the guessed password). Encryption also releases information by revealing the ciphertext (which obviously has some relation to the confidential plaintext). Also in an auction, the bids must remain secret, but when the auction is finished, the winning bid should be made known. All of these examples involve some form of declassification.

The DLM allows for relabelings of data via declassification, so long as the declassifying procedure has the authority of the owner(s) of the data. This is known as selective declassification. Jif introduces the primitive `declassify(datum, fromLabel, toLabel)` for this purpose. Jif also implements an authority model similar to Java’s stack inspection, although the authority in Jif can be established statically. We severely limit this feature by requiring all authority to be rooted in some authentication and allowing declassifications only in special Closures. We consider declassification in more detail in Section 2.3.4 and we introduce our own mechanism called trusted declassification in Chapter 4.

2.3.3 Qualities that map well to DLM

**Role hierarchy**  Clearly the DLM is well suited to modeling a role hierarchy. As described above, this is especially effective in naturally hierarchical settings such as corporations, medical institutions, military structures, etc. In the DLM, as in RBAC, the principal hierarchy forms a partial order and all permissions are delegated.

**Permissions**  Currently, the DLM hierarchy differs from RBAC’s in that permissions are not directly attached to roles. More precisely, the DLM doesn’t have a clear concept of permissions as does RBAC. Rather, the main permission in DLM is declassification. A principal has permission to declassify by modifying his own policy labels on data. Our extension of trusted declassification serves to make declassification more fine-grained. This may help in modeling the more fine-grained permissions of RBAC. The DLM hierarchy corresponds more naturally to $RT_0$ in which permissions are just named sets attached to roles.
Confidentiality, integrity and availability

The most basic policy which the DLM provides is confidentiality. The confidentiality labels described above can be used to implement very fine-grained confidentiality policies. Research has also been conducted in expanding this to include integrity [ZZNM02, LMZ03, LZ05c] and availability policies [ZM05].

Integrity is the dual of confidentiality [Bib77]. While confidentiality restricts where data can flow to, integrity restricts where the data can flow from. Thus, in the DLM, an integrity label \{Alice\} indicates that the data has only been modified by Alice. If that data is added to data labeled with \{Bob\}, the label becomes \{Alice;Bob\}. If a procedure requires data that can only have been touched by Alice, then the data labeled \{Alice;Bob\} would be rejected as an illegal flow.

Availability policies are the least well-developed information-flow policies for the DLM. Zheng and Myers propose availability labels on data as a collection of principals which represent different kinds of system requirements. If a certain bit of data will be available so long as either host h1 or host h2 is live, then this data could be labeled as \{h1∧h2\}. In the same way as other DLM labels, if this data is added to data that is available so long as there is enough power, then the combination of policies guarantees data availability only when h1 and h2 remain live and there is sufficient power in the system. Others principals they describe are DDoS1000 and puzzle. When data is labeled with these principals, then the availability of this data is guaranteed so long as the system is not afflicted by a DDoS attack involving 1000 machines, or as long as an attacking system is not able to break a puzzle which limits access rates, respectively.

Constraints

Policy constraints, such as separation of duties (SoD) constraints, have some natural implementations in the DLM. This requires analyzing the principal hierarchy to ensure that two separate roles are not populated by the same users. Furthermore, the data must be properly labeled so that it cannot be shared by two separated roles. For example, if the Teller should not be able to modify audit information, then the audit report must be properly labeled so that it is protected from the Teller. This would not be the case if the report were labeled public, for example. Furthermore, if there are any declassifiers which can pass the data between two separated roles, then those declassifiers must limited. For example, passing information from the Teller to the Auditor should require a declassification. Since certain information from receipts must be accessible to the auditor, to verify accounts, that information must be declassified at some point so the Auditor can read it. Special, trusted declassifiers which strip some information from receipts and pass them to the auditor may be explicitly allowed, knowing that they do not violate separation of duties. Other policy features, such as temporal constraints and obligations remain open areas.

2.3.4 Declassification

A major theme in STLS relates to the regular, infrequent need to weaken the strong property of noninterference enforced by STLS through declassification (see the survey by Sabelfeld and Sands [SS05]). Declassification introduces “escape hatches” by which security labels are weakened. It is needed for such common operations as password
checks and encryption, because both of these operations release a small amount of secret information. For this reason, declassification is the main vehicle by which information flow policies are made more expressive.

As described earlier, Jif’s selective declassification [Mye99b] is the only declassification mechanism currently implemented in a full-scale language. We restrict this mechanism in order to lift out authorization into an external, global policy. In this way, we are able to prove the security property of noninterference modulo trusted methods (Theorem 4.2.8, stated and proved in Chapter 4).

Much work has recently been done on declassification, as described in a recent survey [SS05]. In this survey, the authors loosely divide declassification schemata into four categories: who, what, where and when. Our model does not fit nicely into any of these categories. It corresponds mostly to “where” declassification may occur (in explicitly identified declassifiers). “Who”, “when” and “what” is declassified may be gleaned from analyzing the policy and the declassifiers themselves. Our system could naturally be strengthened by quantifying exactly what may be leaked by declassifiers. For example, our system facilitating knowing that Alice’s data can only be leaked by a password check by merely examining the external policy. Analyzing this declassifier, it could be determined that only one bit of information is leaked per call. A further analysis could ensure that it is not called a sufficient number of times to leak more than a certain amount of information.

Broberg and Sands recently introduced the notion of flow locks [BS06] for describing temporal policies. This work is similar to ours in that spots of declassification are limited to explicitly identified regions. We can imagine placing appropriate flow locks around our declassifiers. While this mechanism is very general, it is also very localized. Our policies are more global and more separable.

Chong and Myers introduce a mechanism for downgrading until conditions [CM04]. This model allows downgrading only in the presence of externally verified conditions. It is similar to ours in that we both check an externally verified condition. They open new flows, which are not subsequently closed, while our mechanism limits declassification to the bodies of declassifiers. Furthermore, they provide some possibilities for conditions, but they provide no external policy or actual implementation.

Ana Matos and Boudol’s non-disclosure policies [MB05] are also related to our approach. They have locally induced, transitive policies. Their system makes an important contribution in handling concurrency, but they do not have an implementation; we accepted the limitations of Jif (no concurrency) in order to facilitate an implementation. They also do not allow declassifications to be expressed as a global policy.

Another well-studied declassification mechanism related to ours is robust declassification [MSZ06] which is implemented in Jif [CM06]. The key to this mechanism is in the use of integrity. It uses integrity to ensure that low integrity flows do not influence high confidentiality data that will later be declassified.

Tse and Zdancewic propose a decentralized, certificate-based mechanism for declassification [TZ05]. Like us, they use subtyping to describe declassifications. They prove a noninterference theorem that says that so long as no declassifications are visible to the observer, noninterference is maintained. Furthermore, they are able to justify
all declassifications based on externally validated certificates. They implement this approach in the very recently developed language Sj [Tse07]. A merging of our approach and theirs may yield a very useful architecture for building distributed applications.

Declassification is a way that critical security policy decisions can be hidden in the midst of application code. This violates the characteristic of separable policy we seek to meet in this work. This problem can be addressed most effectively by encoding declassification permissions in a high-level policy and deferring to this policy at each runtime declassification point. This is essentially what we provide with the mechanism, called trusted declassification (Chapter 4). Trusted declassification follows the spirit of intransitive noninterference [MS04, RG99], which Sabelfeld and Sands identify as belonging to the “where” category of declassification. This work introduces lattice levels that principals must pass through when transitioning from one level to another. We expand this concept by encoding these policy decisions in an separate, external policy.

Li and Zdancewic [LZ05a] propose an alternative version of noninterference called relaxed noninterference that uses downgrading policies that embed the lambda code of declassifiers directly into the type annotations. Their approach is related to ours in parameterizing the type system with specific declassifiers, but what they strive to gain in precision by placing lambda code in types is unwieldy. The approach we take of merely naming declassifiers is more practical.

2.3.5 STL applications

Much of the previous work in STLs has focused on laying theoretical foundations. Only two projects have generated realistic programming languages, Flow Caml [Sim03b] and Jif [MZZ+], and now possibly Sj [Tse07], but only Jif has been used to build realistic applications. Other than our own applications, presented in Chapters 3, 5, and 7, the number of applications is still very small. They include only a toy version of battleship [MZZ+], mental poker [AS05], and only recently a more expressive framework supporting the development of web servlets along with two demonstrative servlets for instant messaging and shared calendar management [CVM07].

In the area of future work [SM03], entitled system-wide security, Sabelfeld and Myers note that a system is only as secure as its least-restricted component. This is a problem that MAC operating systems security tries to solve through severely limiting the trust it has in any application components. The problem is that such policies are overly restrictive in practice and restrictions must be loosened. This increases the vulnerability of the whole system, however, because there is no basis for certifying that applications will not be compromised when restrictions are relaxed. This is the basic motivation for the work we discuss in Chapter 5.

Another area of work in using security-typed languages for system-wide security takes the opposite approach [ZZNM01, ZZNIM02, ZZNIM03, CLM+07]. In this work, a program places practically no trust in the operating system. Then security-type labels are used to split the program into a distributed system, deploying parts of the application to various remote hosts. By duplicating components and using voting protocols, these systems seek to ensure integrity of data in the presence of mutually distrusting
hosts. They build on the idea of security by construction through constructing an entire distributed system from verifiably secure components.

Our work contrasts with this body of work in that we wish to compose security-typed applications with existing security mechanisms. Rather than developing entire systems, we wish to provide provably secure components that fit into existing systems and adapt existing security mechanisms according to their security policy. In this way, we are able to interface with existing systems and supply provably secure applications that interact with existing commodity applications. It is not clear how the split approach would handle interactions like those between an email client and server, for example.
Chapter 3

A Real-World Application

Developer tools and programming experience have not evolved in concert with language features for STLs. At the time of this work, there were only two significant language implementations, Flow Caml [Sim03b] and Jif [MZZ+] and prior to this work there were only two applications [AS05, MZZ+], both written in Jif. The literature frequently postulates on practical, distributed applications with many principals and complex policy models such as tax preparation [Myc99b], medical databases [SHTZ06] and banking systems [TW04]. However, the only completed applications were both “toy” applications with only two principals within a simplistic distributed environment. For this reason, many language features such as dynamic principals and declassification, as well as integration with conventional security mechanisms such as cryptography, certificates, certificate authorities, mandatory access controls and network authentication protocols were yet to be explored.

A central theme throughout this thesis is to explore the practicality of these language tools by building realistic applications. We seek to discover whether this tool for secure programming can hold up to its promise of delivering real-world applications with strong security guarantees. Two key criteria we used for defining “real-world” were that 1) they be modular, i.e., the application should interact with other non-security-typed, networked components while still maintaining the security policy of its data and 2) the security policy should be separable and flexible, i.e. easily re-configurable, such that the application could be of general use (not just in a military, MLS setting, but also in a corporate setting, for example). We conduct this experiment by implementing an email system in Jif. Based on this practical experience, we identify the benefits and shortcomings of Jif when building modular applications with separable policies. This chapter provides an outline for the remainder of the thesis.

A principal result of this experiment is that we succeeded in developing the first realistic STL application for which we can easily assess that there is no information leakage beyond what is allowed by a clear, user-defined, high-level policy. In other words, we are able to bridge the semantic gap between a simple, high-level policy and the operational semantics of the email system. We reflect on the security provided by our system at the end of the chapter.
To accomplish this, however, we find that the information flow policy must be made more expressive and be separated from the application. We also require runtime infrastructure. To solve the policy problems, we provide new principal infrastructure, JPolicy, which expands the expressiveness of information flow policy through trusted declassification and improves the separability of policy by compiling JPolicy into code. We describe JPolicy more extensively in Chapter 4. We only provide limited runtime infrastructure here; we make this infrastructure more flexible and portable in Chapters 5 and 7. Finally, our practical experience motivates the importance of software development tools for resolving information flow conflicts that inevitably arise in application development.

Some of our work uses concepts (PKIs, email encryption, etc.) explored more extensively in such systems as OpenPGP. Our purpose in developing the JPmail client, however, was not to replace the state-of-the-art secure mail clients (for a survey, see [GNM+05]), nor to replace extensive secure email infrastructures such as OpenPGP [CDFT98], but rather to investigate the interaction of security-typed programming with real-world security tools, such as certificates, symmetric and asymmetric encryption, etc.

The remainder of this chapter is organized as follows. We begin in the next section by providing a sketch of an email system, the threats it faces and the kinds of security policies it requires. Section 3.2 discusses the security that can be provided by Jif, the limitations of Jif and some solutions to these limitations. Section 3.3 concisely describes the architecture of our JPmail system. We describe the principal infrastructure tools in more detail in Chapter 4, but we give an overview here, focusing on infrastructure that is particular to JPmail, such as cryptographic principals. Section 3.5 provides a limited security evaluation of our email client and discusses our experience with Jif. Section 3.6 provides software engineering principles discovered through practical experience and our implementation of these principles in an STL integrated development environment (IDE).

3.1 Overview

An email system is particularly useful for the study of application development in security-typed languages. This is not only because email is ubiquitous, but also because it has been a frequent avenue for security leaks [RE05, Fed02]. Moreover, email has a wide variety of security policies that it might need to enforce: including policies from military multi-level security (MLS) [BL73b] to organizational hierarchies [FSG+01]. Finally, email policy is naturally distributed, with unique principals interacting across potentially distant clients. We seek to support policies that involve these diverse and dynamic principals.

Illustrated in Figure 3.1, the JPmail system (JP = Jif/Policy) consists of three main components: JPmail clients, the Internet and public mail servers. Written in Jif, the JPmail client (or just JPmail throughout) is a functional email client implementing a subset of the MIME protocol. The JPmail client software consists of three software components: a POP3-based mail reader, an SMTP-based mail sender and a policy store. The client provably enforces security policy from end to end (sender to recipient). Policy
is defined with respect to a principal hierarchy. Each environment defines principal hierarchies representative of their organizational rights structure.

### 3.1.1 Security policy

The single real-world security policy we defined at the outset of this work was seemingly simple:

_The body of an email should be visible only to the authorized senders and receivers._

However, provably realizing this policy was more complex than it would initially appear. We make two clarifications about this policy. Firstly, in this work, we are only concerned with privacy (confidentiality). Secondly, our email client is not inherently limited to sending email only to authorized receivers. The way JPmail handles unauthorized recipients depends on the user-defined policy (see Section 3.4.2).

Consider some dangers in email. 1) In the case of a malicious insider, email was used to leak classified documents [RE05]. 2) In another case, a programmer mistake led to a privacy violation for a list of patients using anti-depressant medication [Fed02]. 3) An email application also handles passwords for logging into remote servers and could leak a password by sending it to the server as plaintext (a protocol that some servers use, in fact). 4) An email client that uses PGP or other systems could accidentally or maliciously leak keys.

Given these threats, which involve both malice and mistakes (on the part of both the programmer and the user), how can one be sure that an email client is safe to use? The answer to this critical question lies in two realms: the proper configuration of an
email policy and the application’s faithful, verifiable implementation of that policy. How should this verification be done? It is not unreasonable to verify some of this by hand, but the parts that are verified by hand should be small and straight-forward. It would be desirable to be able to verify the remainder of these complex systems automatically.

Jif provides the basis for this in performing automated verification of information flows through security-annotated type-checking. For this reason, it promises to be a powerful, tool for developing secure applications. At the same time, however, Jif has practical limitations when it is being used to build components in a distributed system. In the next section, we explore these features and limitations.

3.2 Building a secure application with Jif

Jif is an object-oriented, strongly-typed language based\(^1\) on Java. In Jif, the programmer labels types with security annotations according to the decentralized label model (DLM) [ML00]. The compiler uses these annotations during type-checking to ensure noninterference. For example, assuming \texttt{alice} and \texttt{bob} are principals, \{alice:\} is a DLM-label in Jif syntax indicating that a particular value is owned and readable only by alice. Thus, the following code would produce a type error, because it attempts an illegal flow of information from a sensitive string owned by alice to a string owned by bob.

\begin{verbatim}
String{alice:} password = "1fh2;zg";
String{bob:} leak = password; // causes error
\end{verbatim}

This provides the starting point for implementing a secure email client in Jif. It suggests that if we properly label the data we want to keep secret (the bodies of emails, passwords and keys), then Jif will handle the rest. Jif implements a single, strong, information-flow policy—noninterference—parameterized by principals and delegations. One of the advantages of noninterference is that it is an end-to-end policy (the same policy applies for the whole lifetime of data—from its creation to its destruction). Consider the following code for an email data structure:

\begin{verbatim}
public class Email {
    String{} toAddress;
    String{} fromAddress;
    String{this} body;
    public Email(String{} to, String{} from, String{this} body) { ... }
}
\end{verbatim}

If Alice wants to send an email to Bob, she could use the following declaration:

\begin{verbatim}
email policy and the application’s faithful, verifiable implementation of that policy. How should this verification be done? It is not unreasonable to verify some of this by hand, but the parts that are verified by hand should be small and straight-forward. It would be desirable to be able to verify the remainder of these complex systems automatically.

Jif provides the basis for this in performing automated verification of information flows through security-annotated type-checking. For this reason, it promises to be a powerful, tool for developing secure applications. At the same time, however, Jif has practical limitations when it is being used to build components in a distributed system. In the next section, we explore these features and limitations.

3.2 Building a secure application with Jif

Jif is an object-oriented, strongly-typed language based\(^1\) on Java. In Jif, the programmer labels types with security annotations according to the decentralized label model (DLM) [ML00]. The compiler uses these annotations during type-checking to ensure noninterference. For example, assuming \texttt{alice} and \texttt{bob} are principals, \{alice:\} is a DLM-label in Jif syntax indicating that a particular value is owned and readable only by alice. Thus, the following code would produce a type error, because it attempts an illegal flow of information from a sensitive string owned by alice to a string owned by bob.

\begin{verbatim}
String{alice:} password = "1fh2;zg";
String{bob:} leak = password; // causes error
\end{verbatim}

This provides the starting point for implementing a secure email client in Jif. It suggests that if we properly label the data we want to keep secret (the bodies of emails, passwords and keys), then Jif will handle the rest. Jif implements a single, strong, information-flow policy—noninterference—parameterized by principals and delegations. One of the advantages of noninterference is that it is an end-to-end policy (the same policy applies for the whole lifetime of data—from its creation to its destruction). Consider the following code for an email data structure:

\begin{verbatim}
public class Email {
    String{} toAddress;
    String{} fromAddress;
    String{this} body;
    public Email(String{} to, String{} from, String{this} body) { ... }
}
\end{verbatim}

If Alice wants to send an email to Bob, she could use the following declaration:

\begin{verbatim}

\end{verbatim}

1Jif does not provide support for inner classes or threads, because of the ways they complicate information flow analysis. Jif is described most completely by Myers [Mye99a], has online documentation at \url{www.cs.cornell.edu/jif/} and a helpful, practical overview, along with expository examples, is given by Askarov and Sabelfeld [AS05].
Here, the email headers are public ({} is the Jif syntax for a public label) and the body of the email will be labeled {bob;} (since the {this} label in the class definition is always replaced with whatever label is used when an instance of this class is created). Suppose that a delegation also exists from Bob to his wife Charlotte. Under a strict noninterference policy, we could be certain, based solely on this declaration, that no one but Bob (and Charlotte, to whom he delegates) could ever read the body of this email. Furthermore, Jif prevents the programmer from leaking information through email. For example, the following code would generate an error, because password is labeled as {alice;} while the constructor for Email requires that the body be labeled {this} (which is {bob;} in this case):

Finally, observe one more important property of Jif: compositionality. Jif requires that a method’s information flows be accurately indicated on the method header and then verifies that the header and the information flows in the body are consistent. After that, the body never needs to be examined again by the type checker. These analyses are used in the later evaluation of calling functions. Thus, we recursively build upon smaller analyses toward a total view of system information flow.

Detailed below, Jif presents several challenges as a tool for system development.

### 3.2.1 A principal store

One challenge is in managing principals beyond the limited domain of a single Jif program execution. Principals need to be defined explicitly in the program, along with the policies they enforce (whether they allow certain declassifiers, e.g.). Furthermore, for our email client to be useful in practice, the principals persist beyond a single execution so that labeled data may leave one Jif application (through a network socket, for example) and later re-enter another one. Intuitively, these principals should be anchored in principals in the real-world, e.g., users of the parent operating system.

This is a problem that prior, simple Jif demonstration applications [AS05, MZZ+ ] did not face, because they did not communicate with the non-Jif world and they used trivially simple policies. Being merely games, they only had to define principals “me” and “opponent” and those principals only needed to have meaning for the duration of one execution of the program. In our application, to the contrary, it is necessary to utilize principals with persistent meaning across multiple applications. Additionally, for these principals to be robust, it should not be possible for the user to impersonate a principal illegally; i.e. we need some way to authenticate the user of a program as a particular principal.

One solution would be to map operating system principals one-to-one onto Jif principals and leverage the existing technologies of distributed systems. It is desirable,
however, that there be a one-to-many or many-to-one mapping between our application’s principals and operating system principals, in order that our application can be sufficiently general. For example, in an MLS setting, it might be desirable for many different users to be able to send “secret” mail—this is a many-to-one mapping. Conversely, a single user should be able to take on the role of “classified” and of “top secret”—this is a one-to-many mapping. Furthermore, defining a distinct principal-space for a Jif application frees the application from being tied to a particular operating system instance. This brings us to the following technical challenge:

**Problem 1** Jif principals must be developed which are persistent across multiple executions of an application and consistent across multiple applications and operating systems. Jif principals must also be unique so that they cannot be impersonated.

**Observation 1** A public-key infrastructure (PKI) provides this uniqueness, persistence and consistency. Other environments such as a distributed MAC system can also provide such qualities.

**Solution 1a** By mapping Jif principals to public/private key pairs and leveraging existing PKI technologies for creation of certificate authority and public key certificates, we describe a principal-space with the desired properties. Furthermore, we prevent illegal impersonation of a principal by requiring that a user have access to the principal’s private key before taking on that role. We describe this in more detail in Section 3.4.2.

**Solution 1b** By mapping Jif principals to users in a distributed MAC system, we are also able to ensure uniqueness, persistence and consistency across a distributed system. We can handle authentication through whatever authentication mechanism is native to the platform and interface it through PAM. We describe this in more detail in Section 5.4.

### 3.2.2 System channel policy

Returning to the above code, let us see how Jif prevents illegal information leaks. Consider this code fragment in which `msgToBob` is sent out on a socket:

```java
JifSocketFactory socketFactory =
    new JifSocketFactory();
Socket[] outchannel =
    socketFactory.createSocket(mailhost.mailport);
outchannel.write(msgToBob);  // causes error
```

In general, a `Socket` could be trusted to keep a certain level of data confidential (using IPsec to connect with a trusted operating system, e.g.) and so the `Socket` class is parameterized by a label (class parameterization is indicated with []'s). Labeling sockets, files and other I/O channels must be handled with care, because these labels express a critical dimension of security policy. All the ways that an application interacts with
its operating system environment impact the security guarantees of the application and must be handled through policy decisions.

Runtime infrastructure is needed in STLs to mediate interactions between STL application data and operating system resources. Though finding a general solution for handling all possible runtime environments is unlikely, we provide principles in Chapter 7 for developers to write specialized runtime environments in specialized settings. Furthermore, because critical security policy decisions are implemented in runtime infrastructure, we wish to control these through high-level policy.

Labeling standard I/O is another example of this. Who decides how input from the terminal should be labeled? A reasonable policy (Jif’s default policy) is that all data read from the terminal should be labeled with the principal corresponding to the user who originally executed the application. In other words, the data that the user enters should be protected according to her security policy.

The default Jif runtime system implements this by first adding a special native principal as an input parameter to main (the method called when the application is executed). It then requires that standard input be labeled at least as confidential as this principal. We again see the need for authentication (as described in Problem 1) when connecting the native principal with the email principals in our policy.

```java
static void main(String[] args) {
    jif.runtime.Runtime runtime = null;
    try {
        runtime = jif.runtime.Runtime.getRuntime();
    } catch (SecurityException e) {
    }
    InputStreamReader inS = null;
    try {
        inS = new InputStreamReader(new label {
            runtime.stdin(new label {
                runtime = null;
            });
        } catch (SecurityException ex) {
            }
    } catch (NullPointerException e) {
    }
    ...
}
```

**Problem 2** When STL applications must interact with non-security-typed components, such as resources managed by the operating system, some security policy mismatch can take place. This policy mismatch must be handled carefully to achieve system security.

**Observation 2** If all system resources are handled through a single, policy-driven mechanism, the labeling of input and output channels can be controlled through high-level policy.

**Solution 2** We introduce a new `Channel` abstraction for handling labeling decisions on I/O. `Channels` can be specialized for different environments and controlled through a high-level system policy infrastructure, SPolicy, that we provide. This runtime infrastructure
supplies portability governed by a separable policy. Chapter 7 is dedicated to deeper exploration of this.

3.2.3 Declassification

In our initial experiment, we specified that internet traffic should always be considered public (as described further in Section 3.3). Thus, socket labels must be \( \emptyset \). We implement this by requiring that our `socketFactory` only return public sockets. Then Jif can catch security violations such as the one above. A socket's `write` method requires that input parameters are no more secret than the label on the socket. Thus, trying to send `msgToBob`, whose label \( \{\text{bob:}\} \not\subseteq \emptyset \), causes an error.

This brings us to the most serious, practical problem with the code above: this email could never be sent to Bob. Because it is labeled as \( \{\text{bob:}\} \), Jif prevents it from being placed on a public channel and sent to the SMTP server. The only way around this would be if there were a channel directly to Bob that no one else could see, but this would preclude using existing mail servers and existing networks. Another obvious solution would be to use encryption. However, under the strict noninterference policy, even encryption would be disallowed, because putting a ciphertext on a public channel is a possibilistic leak, releasing a small amount of information about the plaintext.

We might decide that the information leaked through encryption is an acceptable leak, however. Then a Jif solution is to relax the policy slightly through declassification. For this purpose, Jif provides a primitive, the `declassify`-statement:

```java
outchannel.write(
    declassify(AES.encrypt(key, msgToBob), \{\});
```

This introduces a new problem. Although we have successfully published the email, we have now lost the meaning of the policy \( \{\text{bob:}\} \). Allowing any relaxations of the policy leaves the programmer wondering what the new policy actually is. The label \( \{\text{bob:}\} \) no longer means that only Bob and Charlotte can read the data. It now means that only Bob and Charlotte can read the data, modulo some information about the data that might be released by some declassification statements somewhere in the program. This is problematic, because the declassification statements have nothing to limit them and could actually release all the information to any security level, including public. At the same time, it is not a total loss, because we know, at least, that the information could only be leaked through declassification statements. A security analysis of Jif only needs to focus on the declassification statements to gauge whether the information leakage is dangerous or unacceptable. Such an analysis was done in the `jifpoker` case study [AS05] and the SIF project [CVM07].

The security analysis would be easier and safer, however, if it could be localized to a small, single policy file, separate from the application itself. Rather than treating every declassification as a potential wildcard, it is possible to place some limits on the allowed kinds of declassification for a particular principal and specify these in a small policy file.

\(^2\)The \( \not\subseteq \) operator indicates that \text{bob} does not delegate to public.
For example, the policy file might specify that Bob’s data can only be declassified if it is also encrypted. This restores local meaning to a label such as \{\texttt{bob}:\}. Consider an example policy file, focusing only on Bob’s policy statements:

\begin{verbatim}
bob -> charlotte % Bob trusts his wife with all data
bob allows smtp.DeclassMsgBody(family)
bob allows crypto.AES(public)
bob allows crypto.MD5(public)

% Bob’s children
family -> john
family -> sarah
\end{verbatim}

With this policy file, when the programmer sees that the email is labeled \{\texttt{bob}:\}, she knows that this email is limited in the ways it can flow: it can be sent on public channels, but only if it is encrypted or hashed first. Bob can send information via Email, but only to his family. This eliminates the need to scour the code for all declassifiers that could leak Bob’s data. The policy file states explicitly which declassifiers are allowed.

\textbf{Problem 3} Noninterference is a very strong security property that must often be relaxed somewhat for practical applications. Declassification is a mechanism for relaxing noninterference, but it introduces confusion about the security policy of a program, complicates security analysis and increases the likelihood of information leakage.

\textbf{Observation 3} Based on this, we make the observation that in order to understand the meaning of security-policy labels in a security-typed program with declassification, it is necessary to know three things: 1) the principals used in the program, 2) the delegations they make and 3) the declassifiers they trust. With this information in hand, the meaning of the policy \{\texttt{bob}:\} is restored. If we know, for example, that Bob delegates to no one and trusts only AES encryption, then we know that the only information which will be released about the body of this email will be the extremely small amount of information released by AES encryption.

\textbf{Solution 3} To address this challenge, we added a principal infrastructure to Jif that allows the programmer to define, up front, the principals, delegations and declassifiers that may be used in a program. We describe this infrastructure in Chapter 4, including a proof of the security it maintains. The infrastructure is designed to be expressive (capturing various declassification policies and principal relationships that can be altered for various settings) and separable from the application code.

3.2.4 Relabeling message bodies

Consider again the code for an email data structure, given above. If Alice wishes to send an email to Bob, she must first type in the email from her terminal. Thus, the email text enters the JPmail client from an input stream, stdin, labeled \{\texttt{alice}:\}. If she then wishes to send this string to Bob, it must be relabeled to \{\texttt{bob}:\}. Let us also
introduce a new concept, a *dynamic principal*, which allows the sending of an email to be parameterized based on two dynamic values: the user who is sending and the chosen recipient.

```java
Email{rcpt:} send( String{} to, String{} from, Principal user, Principal rcpt ) {
   String[user:] body = stdin.readLine();
   Email{rcpt:} msg = new Email( to, from, declassify( body, {rcpt:} ) );
   return msg;
}
```

Thus, if *alice* and *bob* are principals defined elsewhere, the email could be created and sent as follows:

```java
Email{bob:} msgToBob = Email.send( "bob@psu.edu", "alice@psu.edu", alice, bob );
outchannel.write( declassify( AES.encrypt( key, msgToBob ), {} ) );
```

Here again, we need declassifying filters. In this case, we need to leak more information than in the encryption declassifier described above—we need to leak the body of the text. Should such leakage be possible? *This is a policy decision that should not be buried in the code, but should be declared at a high level.*

The answer depends on the security model. In an MLS setting, this should not be possible unless Alice and Bob are both working at the appropriate relative security levels. In other words, this declassification should not be allowed at all and the method `Email.send(...)` should return `null` unless user delegates to rcpt, written `rcpt actsfor user` in Jif (e.g. in an MLS setting, `user` could be `secret` and `rcpt` could be `secret` or `top−secret`). In a corporate setting, it may be acceptable to declassify email text so that anyone in the company can read it. If it is going to an external principal, it may be necessary to perform an audit or add a disclaimer. We accommodate such security policies in the following way:

```java
Email{rcpt:} send( String{} to, String{} from, Principal user, Principal rcpt ) {
   String[user:] bodyIn = stdin.readLine();
   String[rcpt:] body = null;

   if ( rcpt actsfor user ) body = bodyIn;
   else if ( authorize( user, rcpt, DeclassifyMsgBody ) )
      body = DeclassifyMsgBody( user, rcpt, bodyIn );
   else if ( authorize( user, rcpt, DeclMsgBodyAudit ) )
      body = DeclMsgBodyAudit( user, rcpt, bodyIn );

   Email{rcpt:} msg = new Email( to, from, body );
   return msg;
}
```

3Note that this code does not correspond directly to the Jif implementation. We use Jif Closures for this which are such flexible constructions that they become syntactically cumbersome. We present a syntactically simplified but semantically equivalent form here to aid the reader.
The authorize method checks whether the principal in the first argument trusts the
declassifier (third argument) to declassify information to the principal in the second
argument. Thus, an MLS policy should not allow either declassifier to be used, while
a company policy may allow DeclassMsgBody if both principals are in the company and
DeclMsgBodyAudit if the first principal is in the company, but the recipient is external.
These details are specified in a policy file, which is compiled into Jif with our policy
compiler and established at the start of a Jif application. By teasing out the policy, we
have made it possible to change the policy model of an application merely by changing
the high-level policy file.

3.2.5 Label conflict resolution

In each of the previous examples, subtle information flows were highlighted and
analyzed. When the programmer fails to implement information flows properly and pro-
vides conflicting labels for data, the compiler flags an error and provides some information
about the cause of the error (the origin of the labeling conflict). Because information
flows are subtle, discovering the source of errors and reconciling conflicts can be quite
challenging. Consider the following code snippet that adds a read statement to the
previous code listing:

```java
static void main{user:}(){principal user, String[] args} {
    try {
        int inData = inS.read();
    } catch (IOException ioE) {}
    } catch (NullPointerException e) {} 
}
```

The Jif compiler flags this with an error that is quite hard to discern.

HelloWorld.jif:17: PC at call site more restrictive than begin label of native
public int read().
    int inData = inS.read();
          ^--------

This error message might lead one to believe that the cause of this error lies in
the label on inS, because read is called after the inS object is dereferenced. If read is
called successfully, it means that the NullPointerException was not thrown and thus that
some information about inS is leaked into the call to read. For this reason, the security
protection offered by read must be at least as strong as the information leaked about
inS. To try out this theory, we can change the label on inS to {user:}. Unfortunately,
this produces unsatisfactory results. Now, the compiler complains about an error in the
assignment to inS:
HelloWorld.jif:11: Label of right hand side not less restrictive than the label for local variable inS

```java
inS = new InputStreamReader[{user:}](
```

The cause of this error turns out to be the default label on main’s input parameter, user. All unlabeled method parameters are given the highest security label, {∗:}, by default. Figuring this out can be quite difficult, however. The reason the label on the principal user matters is that we are using it to make a new label as the input parameter to new InputStreamReader.

**Problem 4** Complex causes for information flow errors frustrate practical application development in STLs.

**Observation 4** Integrated development environment (IDE) tools for tracking down constraint conflicts and other subtle information flow errors can provide invaluable assistance to the STL programmer.

**Solution 4** We provide principles to guide the development of software engineering tools to aid STL programmers and provide a demonstrative set of tools for Jif. We consider this in more depth in Section 3.6 and following.

### 3.3 JPmail architecture using a PKI

In our first experiment implementing JPmail, the goal was to provide end-to-end confidentiality of email bodies from sender to recipient. We presumed no specialized operating system or infrastructure, nor specialized or security-aware SMTP and POP3 email servers. In this way, our results extend across many diverse systems.

The security policy we implemented depends on only the most basic operating system security assumptions, namely that the JPmail-local file systems are trusted to store information securely, based on the access control list on a given file (thus if a file is readable only by the user, it is considered safe from leakage). Internet communication is generally untrustworthy, and is deemed as public channels throughout. The SMTP and POP3 servers are not written in Jif, and do not enforce any security policy save that which is provided by their implementation and administration. For the purposes of this work, we assume nothing about the servers’ ability to prevent leakage of user data: i.e., any information sent to them is deemed public.

We also require a private-key infrastructure (PKI), with a trusted certificate authority (CA) and a local keystore (stored in file protected by the user’s access control list). There is no need for an extensive PKI, but each potential recipient must have a corresponding public key in our keystore, verifiable with the CA. Any recipients without a private key are handled in a separate category, since it is not possible to encrypt their emails to protect them from visibility on the internet.
JPmail is in no way bound to a single environment. To the contrary, in Chapter 5, we implement JPmail in a different runtime environment, a MAC operating system that provides labeled, encrypted sockets. The PKI and CAs are unnecessary in that context. This is a testament to the portability of our policy infrastructure. In the following, we give an architecture for JPmail in a PKI-based system.

![Diagram of JPmail architecture](image)

**Fig. 3.2.** Sending and retrieving a message using the JPmail client.

We now give a description of the process of sending and receiving an email in JPmail. In this description, we focus on the information flows that are necessary for sending an email from one principal to another. In both the sending and receiving processes, the data must pass through software filters (points of processing that may audit or modify data) that serve to relabel and/or modify it. In sending email, there are two filters involved; in retrieving it, there is only one (strictly speaking, this one may not be necessary because the information is being upgraded). The only requirements on these filters is that they are authorized by the owner of the data and that they produce the properly labeled output.

The following example refers to the numbered Figure 3.2 in which a principal Alice uses JPmail to securely send an email to another principal Bob, who in turn reads that mail.
Sending email  Alice initializes a MailSender with a policy and her principal name (alice in this case—the policy and principals are explained in more detail in Section 4.3) as well the necessary parameters for the outgoing mail server (address, user name, etc.).

1) Then Alice enters an email, including the header information and the text for the body of the email. This email is labeled as alice since it came from an input stream owned by Alice.

2) The email must then undergo two transformations. First, in order to send out an email, the headers must be readable by the mail server. This requires that they be declassified to public. Secondly, the body must be readable by the recipient, Bob, without being readable by the public. These two steps are performed by a reclassifier, as shown. At this point, the email headers are visible to the server while the body is visible only to the recipient.

3) The next step is to make the entire email visible to the server so that it can be sent out. At this time, we must not compromise the policy on the body, which requires that it should only be visible to bob. To do this, we use a random one-time symmetric key approach. The one-time key (k) is generated, used to encrypt the email body (b), and encrypted with bob’s public key (k^+_{bob}). Then the original body is replaced with the encrypted body along with the encrypted, one-time key, i.e. the message body contains E(k, b), E(k^+_{bob}, k). The encrypted values can be declassified to be visible by the server without compromising bob’s privacy.

4) Finally, the email is sent to the SMTP server, which in turn delivers it to the POP3 server.

Reading email  Bob retrieves his email from a POP3 server using the MailReader class.

5) After connecting to the server, the mail reader takes in each email and examines the label field in the header (Label in the figure). The header information can remain public, but the text of the body must be decrypted and reclassified according to the label field.

6) To do this, we require Bob’s private key. Since Bob has access to his own private key, it can be read in from the file system, labeled as bob. If another user were trying to impersonate Bob, the private key would not be available and the attempted decryption would fail.

7) Since Bob’s private key is labeled as bob, decrypting the body of the email automatically raises the plaintext’s security level to bob. Now that the body is safely in the confines of the Jif sandbox, it can be decrypted without fear that it will be leaked.

8) Finally, since the user who is running the mail reader is bob, this email can be printed to bob’s terminal.

3.4 Principal Infrastructure

Prior to the development of JPmail, there was no work to suggest how STLs could be incorporated into existing systems. JPmail highlights that principal infrastructure is necessary to achieve this goal. Firstly, there is a need to define principals. Concrete system entities must be integrated into the STL concept of a principal. This serves to localize one change that is necessary for portability of STL applications; different systems may require different concrete implementations of principals. Secondly, there is a need to capture and control the declassifications that are allowed in an application. Declassification increases the expressiveness of STLs, but can limit separability, by hard-wiring policy decisions in code. Finally, in order to separate security policy specification from
implementation and maintain *separable* (and easily re-configurable) policies, a principal policy infrastructure is needed to lift these policy decisions into a separate policy file.

For JPmail, we developed *cryptographic principals* as concrete implementations of the STL principal concept. Using JPmail, we also demonstrate the effectiveness of separating policy specification from implementation by showing how policy can be reconfigured, including the declassifiers and delegations each principal allows. In this way, the same application can be reused in diverse environments—a military MLS setting, a university laboratory setting or a corporate setting, for example. In this section, we describe the cryptographic principals and give a high-level view of how the JPolicy tools can be used to customize policy for the JPmail client. We describe the JPolicy tools in more detail in Section 4.3.

3.4.1 Example policy

As an example of a JPmail policy, a security research lab could design a policy in which all of the members are listed in the policy and their public keys are certified by a lab’s certificate authority. Emails can be sent freely throughout the lab. Emails destined for recipients outside the lab are handled by a separate filter that imposes the lab’s policy on external mail (whether it be adding a disclaimer, limiting the number of outgoing messages, auditing outgoing messages, etc.). In Figure 3.3, we illustrate this lab policy. Principals begin with lower-case letters and declassifiers with uppercase (public is repeated only for clarity of reading.) The solid arrows indicate delegations, the “T” arrows indicate allowances and the dashed arrows indicate the lowest level a filter may declassify to. Note that this lab policy was used for the development testing of JPmail.

![Delegation hierarchy and declassification allowances for a sample, research lab security policy.](image-url)
3.4.2 Cryptographic principals

Jif provides a Principal interface that allows for policy to be implemented directly in dynamic principal objects. In particular, Jif Principals maintain a list of principals they delegate to and they also allow the programmer to implement a method which is called to authorize a declassification. The Principal interface can also be implemented with additional member data, allowing us to push public keys (and if available, private keys) directly into dynamic principals. Our policy compiler automatically generates a Principal implementation for each principal given in the policy file.

Cryptography provides two central functions within JPmail\textsuperscript{4}: it is critical for ensuring data is not leaked as it passes outside of a Jif application and it plays an essential role in maintaining the consistency and integrity of principals from one Jif application to another. This former function is achieved via encryption of email bodies. In the latter function, principals are uniquely identified via an association with a public key (certificate). We leveraged existing facilities for creating and verifying X.509 certificates for this purpose. For certificate signing and verification, we created a JPmail-specific certificate authority (CA).

Our use of certificates required us to bind public keys to Jif’s principals. In Jif, principals are created by implementing a Principal interface. We created our own KeyPrincipal by implementing the standard Principal interface (which requires a name, closure authorization testing, equivalence testing and delegation testing) and also adding fields for a public key and, if it’s available to the current user, a private key. Before allowing a public key to be associated with a principal, the public key certificate is validated using the public key of the trusted CA. For sending email to users outside the system, the external principal can be used which can be declassified with the FilterBody declassifier (this could be used to audit emails or add a disclaimer, e.g.).

In order to add a new principal to the system, the principal’s public key certificate must be distributed and a delegation should be added by each principal to their policy to include the new user. Note that this must be done while the email client is offline. The client loads its policy on application start-up and the principals and keys must be available at that time. Handling dynamic updates of policy involves some subtleties still under investigation \cite{HTHZ05, SHTZ06}.

In order to integrate the delegations and authorizations defined in our policy with the local file system, we had to augment the Jif compiler’s runtime environment. By introducing a method to delegate from a NativePrincipal to a non-native principal, we make an association between the user running the program and one of our internal principals. In order to authorize this association, we require that the user can only delegate to a principal for whom the user can provide the private key. In other words, JPmail authenticates the claimed principal by checking that it has a signed certificate from the CA with the principal’s identity and the associated private key before it allows the user to assume that identity. It does this by loading the private key from the user’s

\textsuperscript{4}JPmail uses the following algorithms: DES, TripleDES, AES were used in CBC mode for all symmetric key operations, and RSA Electronic Codebook (ECB) mode with PKCS1 padding for asymmetric key operations. We also used MD5 hashes on passwords for authentication with the email servers according to the POP3 and SMTP protocols.
key store, for which it requires the user's password, and by checking it against the JPmail-specific CA.

### 3.4.3 Declassifiers

Declassifiers play a key role in providing security guarantees. Because many realistic security-typed applications need to have some way of declassifying information (since even functions like encryption and password checks leak small amounts of information), it is valuable to build up a collection of commonly used declassifiers. For example, encryption and auditing functions can declassify data—they expose some amount of data, but (under certain circumstances) it is not enough to be deemed a leak. Such functions are similar to seal classes [AS05, SS05], which provide a declassifying filter that limits when information is released. Libraries of common declassifiers can be carefully engineered and formally verified to prevent unacceptable or dangerous leakage (as opposed to the vanishingly small leak from encryption or the acceptable leak from an audit or password check). Applications can benefit from this vetting, and avoid declassifying through potentially dangerous or untested interfaces [SJS06].

We have built a library of declassifiers for use in Jif applications. The declassifiers we constructed for encryption have widespread value and could be re-used without modification in other applications. Some of the declassifiers we created are special-purpose, because they are made to handle only email objects. Even these, however, are useful as blueprints for other application developers. Because of its extensive use of declassification, JPmail required the exploration of new features available in Jif 2.0: Closures and the Principal interface. Our code serves as the first explorations into the utility and flexibility of these features. We describe our use of Closures as declassifiers in more detail in Section 4.3.

### 3.5 Security Evaluation for JPmail

In this section, we are primarily concerned with evaluating the security and usability of Jif as it applies to JPmail. Jif provides a strong basis for assessing the correctness of an security policy implementation. Jif is able to automatically govern information flow. Without declassification, it maintains noninterference. In the presence of trusted declassification, Jif is able to provide the weaker, but still strong security property of noninterference modulo trusted methods (see Theorem 4.2.8). Furthermore, this security property is automatically verified by Jif’s typechecker.

With this in mind, security analysis for JPmail becomes very strong, with relatively little human work. This is not to say that it is mindless, but rather that it greatly increases confidence and significantly reduces the amount of work that must be done by hand.

---

5. Our code is made available at [http://siis.cse.psu.edu/jpmail.html](http://siis.cse.psu.edu/jpmail.html).

6. Technically speaking, Jif does not have a proof of security for the entire type system. Such a proof has been completed for a substantial number Jif language features, and others are currently under analysis.
Recall our security policy given in Section 3.1.1: “The body of an email should be visible only to the authorized senders and receivers.” We can be sure that this policy has been correctly implemented by examining the label on emails and cross-checking with the policy file about what information flows are possible with that label. In particular, by examining the `smtp.MailSenderCrypto` class and the `readMessage` method, we find that an email is read in from an input stream that has the user’s label (the user who ran the mail client) and consequently is also labeled with the user’s label. Furthermore, the user provides the name of the principal `rcpt` to whom he wishes to send the email. That principal is looked up in the principal store and associated with a Jif principal (which was created from the policy file). The body of the email is then passed through a declassifier, `DeclassMsgBody` which will relabel the body to `{rcpt:}` if `rcpt` is one of the allowed recipient principals, given in the policy file. From this point, the `rcpt` policy governs all information flows. Namely, before this email can be placed on the public socket, it must pass through another declassifier based on `rcpt`’s policy. If `rcpt` allows for AES encryption, then the email is encrypted as described in Section 3.3, using a one-time randomly generated key (which cannot be leaked, because it is labeled `rcpt`) and the principal’s public key.

We can repeat this evaluation for other sensitive data such as keys and passwords. For these items the analysis is even simpler, because they are not dynamically labeled like emails (which depend on user input). The password is given a label when a `MailSenderCrypto` object is created. Checking the policy file, we can see that Strings cannot be declassified except through an MD5 filter. Creating an MD5 hash is necessary for authentication with the mail server. This was made clear when we tried to send the password as plaintext over the mail server socket when establishing a connection. This insecure practice was automatically disallowed by Jif.

At this point, we face a limitation in Jif’s security analysis. Namely, the SMTP and POP3 protocols’ password authentication ensures that a nonce is used to prevent replay attacks. This is not encoded in the Jif labels in any way. Merely knowing the declassifier that is used is not enough to ensure that replay attacks are avoided—only that the plaintext has not been leaked. We have to trust in the protocols for protection against more subtle attacks (a non-trivial assumption). We discuss the limitations of Jif automatic security analysis more extensively in 8. To mention a few briefly from JPmail, Jif is unable to analyze authentication mechanisms, such as the suitability of passwords, nor is it able to reason automatically about cryptographic guarantees (to understand why sending a hashed password is better than sending it plaintext, for example). It does require the programmer to reason about this by flagging leaks, however. Then explicit, intentional decisions can be made about cryptography and authentication.

There are other caveats to security that must not be overlooked. Firstly, the security properties of a program are dependent on the correctness of the Jif compiler (and our policy compiler). Secondly, the security properties may also be dependent on supporting infrastructures. This includes the correctness of encryption libraries and the strength of used cryptographic algorithms, the protection on keystores and correctness of public-key cryptographic libraries as well as the security enforced by the local file system. Moreover, for the system to be secure, the enforced policy must be consistent across all clients.
One advantage of Jif is that it forces the programmer to think in terms of information flows and to consider security concerns from the outset. Interestingly, there is a strong consensus in the software engineering community that performing these kinds of security analysis at design time is essential to the security of the resulting system [DS00]. On the other hand, security by design in advance can cause problems if it is too rigid and tied to particular operating system constructs (i.e. insufficiently portable). This was the long-term experience of MULTICS [KS02]. We developed our policy tools in the hopes of balancing the best of both worlds—security in advance by design along with expressive, separable policy and portability.

Finally, we observed that the policy tool effectively decoupled policy from the programs that they govern. This allowed us to modify policy easily in order to accommodate different security models. By instrumenting the code during development with different options for each filter, we could implement distinct security models without altering the code. Furthermore, by gathering the policy into a single file, it was easier to do a security analysis and gauge what information flows could take place for a given principal, in contrast to leaving declassify-statements in the code.

The difficulty of programming in Jif The shortcomings of Jif are frequently not specific to Jif, so much as they are issues that any security-typed compiler must face. Jif is the most advanced security-typed compiler available and the Jif team should be commended for their substantial efforts, but it is not yet ready for industrial development. Implementing the JPmail client took hundreds of man-hours (not including the time necessary to learn Jif) to generate around 6,000 lines of code. Furthermore, despite the substantial amount of work involved, our mail client is neither flashy nor full-featured. It uses text-based I/O and handles a minimal subset of the MIME standard just enough to allow communication between various principals.

This should be contrasted with the more modest efforts needed to retrofit composable security properties onto the full-featured, GUI-based email client, Pooka, by using the Polymer security policy system [BLW05]. Of course, the price paid with Jif is for provability and completeness. The inlined reference monitors (IRMs) used by Pooka do not provably have complete mediation over all security sensitive operations and almost certainly do not have complete mediation over all information flow sensitive operations, including implicit flows.

To improve on the usability of Jif, we explore principles and provide an implementation of an IDE in the following sections.

3.6 Language Development Tools

Security-typed languages offer a new, unique set of programming challenges, because they require programmers to reason in advance about the information flows that should be allowed in a program. Mature language development tools should aid the programmer in these challenges as much as possible. In this section, we propose some principles that will be helpful guidelines for designing development tools for security-typed languages. We group these principles into three main areas, 1) principal determination, 2) labeling data and 3) identifying and resolving information flow conflicts. In the rest
of the chapter, we will focus primarily on the third area, because we believe it finds its most natural solution in an IDE and this is what we have provided for Jif with our IDE, Jifclipse. We describe the other principles for the sake of completeness and as a target for future work. For each principle, we offer a key, motivating observation.

As a motivating example for these three principles (we will use this as a running example throughout the remainder of this chapter), consider an application that simulates the administration of exams to students. 1) The principals in this setting consist of students who can take the exam, along with the administrator who provides the exam. One policy is that none of the principals trust each other (to prevent all leakage between students). 2) The exam questions are labeled with the examiner’s principal. The students’ answers to the questions are labeled with both the student’s own principal and the examiner’s principal, since they must contain information from the student as well as information about the exam questions. 3) This application has some interesting information flow requirements. The simplest requirement is that students should not share information, so one student’s answers should not be visible to another student. Of course, the application should also not leak the correct exam answers to any of the students. This leads to an interesting information flow, however, because ultimately, each student’s answers must be compared against the correct answers and the result must be released to the student. This leaks information about the exam answers to the students and should only be allowed in a specific circumstance—after all students have turned in their exams.

### 3.6.1 Principal determination

As described through the JPmail experiment in preceding sections, a critical and challenging problem in building secure systems is determining who the principals are in the system and how their information flows can interact. Clearly these principals need to be connected with some identity outside of the application since the sources and sinks of information flows are going to be outside of the application (through various I/O channels). In JPmail, the principals are connected with a PKI and access controls lists (ACLs) provided by the operating system attached to files [MZZ+]. In Chapter 5, principals are connected with labels from a MAC system. In related work, principals were associated with login identities for two players in a game communicating through a serialized file channel [AS05].

One solution to meeting this need is a high level policy system that supports the declaration of principals, the specification of their information flow policies and the way they relate to principals outside the application. We describe such a policy system in more detail in Chapter 4. This policy system includes tools for implementing high-level policy constructs in a way that integrates smoothly with STLs (generating code in a security-typed language). Ideally, such a policy system should also allow for safe dynamic updates of policy [HTHZ05, SHTZ06] to support dynamic environments.

---

7This example is derived from a class exercise used by Andrei Sabelfeld at Chalmers: [http://www.cs.chalmers.se/Cs/Grundutb/Kurser/lbs/JifLab2006/](http://www.cs.chalmers.se/Cs/Grundutb/Kurser/lbs/JifLab2006/)
Observation: The determination of principals in an application and the establishment of information flow policies between those principals is essential for building applications with security-typed languages.

Principle 1 (Principal principle): Development tools for security-typed languages should provide a means for specifying potentially dynamic principals, information flow policies between those principals and the relationships between those principals and entities in the operating system. These tools should also generate or link with the secure infrastructure that implements such policies. This might be done in the spirit of UML specification [HH03], compile-able policy (see Chapter 4), or some other high-level design tools.

3.6.2 Labeling data

After determining what principals can interact in a given application, the data handled by the application must be tagged with the appropriate principal. How to tag data depends partially on the program’s specification. For example, a field that will contain an email server password should have restricted visibility by annotating it with the password owner’s principal, while that user’s name may be a public string. The exam questions should be secret (owned by the Examiner), while each student’s answers should be labeled with both the student’s principal and the Examiner’s principal. These labeling decisions depend on the semantics of the application and must be provided by the programmer.

On the other hand, there are many other variables in a program that could be inferred by the compiler or IDE based on these “seed” labels provided by the programmer. For example, the local variables in a method will have labels that often follow from the parameters on the method. Consider an iterator used to loop through a secret array or a stream used to display exam questions to a student. Likewise, some fields in an object will depend on other fields. The label on the total number of correct answers a student achieves on an exam flows directly from the “seed” labels on the student’s answer, the exam questions, and the exam answers. Inferring labels not only reduces the burden on the programmer with regard to providing labels, but is also more likely to reduce the number of information flow conflicts the programmer causes through mislabeling intermediate variables.

Observation: Some variables must be labeled by the programmer, because their labels are inherently part of the application’s specification, but many more variables can be labeled by automatic inference or default.

Principle 2 (Label inference principle): Development tools for security-typed languages should provide as much label inference as possible, so that the application developer only needs to specify those labels which are inherently part of the program’s specification (and thus impossible to infer with confidence).

3.6.3 Fixing information flows

As a programmer specifies the labels on data, conflicts inevitably arise due to illegal information flows. The number of these conflicts may be reduced when fewer
labels are specified and label inference is more robust, but it is unlikely they will be
eliminated entirely. In fact, it is in this that security-typed languages provide a great
benefit. They serve to expose interactions between data of different security levels even
when these interactions may be subtle and far-reaching in the code, identifying potential
information leaks.

Observation: A primary benefit of security-typed languages is that they expose
information leaks in programs.

Principle 3 (Information flow resolution principle): Security-typed language tools
should provide detailed information to programmers to help them identify and
repair information flow violations.

Information flow leaks may have various causes, and helping the programmer to
identify the cause and fix the broken information flow may be dependent on the language-
specific implementation used for resolving security-type annotations. For example, using
constraints to track information flows, allowing for declassification, and tracking im-

plicit control flows, all complicate the process of resolving information flow conflicts in
programs. For this reason, we break this principle down into a few language-specific
sub-principles.

3.6.3.1 Constraints, declassification, and implicit flows

It is common to use constraints, such as \( \{ \tau_1 \leq \tau_2 \wedge \tau_2 = \tau_3 \} \), to abstract type
system requirements, and existing security-typed languages have used this approach to
model information flows based on security-type annotations. As previous experience
(in security-typed languages as well as in other languages with complex type systems,
such as Haskell or Standard ML) has shown, however, isolating an error in a constraint
system can be challenging. It is not always obvious what the initial error which caused
the constraints to fail to be satisfiable is. Since this is critical to the contribution of
security-typed languages, some development tools are necessary to aid the programmer
in this regard.

To start, good heuristics should determine which constraint is to blame. This
helps the programmer focus in on the source of an illegal information flow. It would
also help to suggest how to repair the information leak. Finally, when all else fails, the
programmer should be able to explore all the constraints involved in the information
leak and determine the source for himself.

Observation: Constraints complicate error reporting and make it difficult to
determine what the source of the information flow may be.

Principle 3a (Constraint resolution principle): STL tools that use constraints to track
information flows should provide detailed information to programmers to help them
view all constraints in a program, especially focusing on constraints involved in a
bad flow.

A related problem regards some situations when declassifications may be needed.
Recalling the example of the exam room, a declassification is needed to expose each
student’s results, but only after the exam is complete. This data exposure violates the information flow policy on the exam’s questions and a security-typed language compiler will rightly flag this as an error. In the proper circumstances, however, it should be possible for a programmer to override such a violation—such as after all exams have been turned in. These declassification points must be chosen carefully, because they open up information leaks. An inference engine may be able to assist the programmer by suggesting places where introducing declassification can alleviate information flow conflicts. A more advanced inference engine may even be able to quantify the amount of information that could be leaked by adding such declassifications [Lau01, Lau03, CHM07, CM07].

Observation: Declassifications are necessary in realistic applications, but because they open information leaks, they should be placed carefully.

Principle 3b (Declassification principle): An STL that uses declassification should provide facilities for suggesting declassifications that can resolve information flow violations, while still allowing the programmer to make the ultimate decision based on the program’s security specification.

Security-typed languages can track implicit control flows in addition to explicit, data flows. This is commonly done by introducing a PC label which is set to the level of any implicit flows at a given point in the code. This label then induces a constraint on any explicit flows at that point in the code. For example, the PC label is raised by a high-secure guard on a conditional and any explicit flows in the body of the conditional are tainted by the PC label. Consider the following code from the Exam Room application, which has an implicit flow from \( qi \) to \( totalAlice \). When evaluating this code, the compiler raises the PC label by joining it with the label on \( qi \) and then joins the PC label to the right-hand side of the assignment, ultimately requiring that \( totalAlice \) be at least as secret as \( qi \).

```plaintext
if (qi != null && qi.isCorrect(x))
    totalAlice = totalAlice + 1;
```

Preventing implicit flows makes for stronger security, but causes more troubles for the programmer. The trouble comes in the fact that the label from the guard taints the whole body of the conditional; this effect is not easy to keep track of. The problem is exacerbated when exceptions come into play, because an implicit flow can be caused by premature termination and taints the code following the place where an exception can be thrown.

Observation: Keeping track of implicit flows by hand can be complex.

Principle 3c (Implicit flow principle): If a security-typed language supports implicit flows, development tools should provide a mechanism to reveal the PC label at a given point in the code.
3.6.3.2 Relaxing security checking

Allowing for variable degrees of security checking can assist in more rapid prototyping of applications (and may improve a programmer’s sanity when using security-typed languages). Furthermore, some dataflow analyses can help to determine when the compiler is being overly conservative and ruling out information flows that should be allowed. In this principle, we gather together three different areas that could accelerate a programmer’s development time.

Alleviating overly conservative checking Because exceptions can cause information flows, Jif requires all exceptions to either be caught or explicitly thrown. This means that every use of an object reference must be wrapped in a try/catch block, because it could potentially throw a NullPointerException. The same holds for array accesses, arithmetic divisions and all other runtime exceptions that could remain unhandled in Java. Handling all these potential control flows becomes arduous and clutters the code. Some dataflow analyses can be helpful for determining when such exceptions are impossible (and thus not requiring them to be handled), and some facilities to automatically insert handling code would be beneficial for the programmer.

Integrating with non-security-typed languages Integration with a non-security-typed language can be helpful in cases when low-level subroutines must be called. This is necessary for developing modular applications and arose numerous times in JPmail. Also, to facilitate the gradual migration to fully security-typed applications, the ability to reuse existing, but unannotated library code (for encryption, for example) is extremely important. Compiler facilities that enable the method headers of these modules to be annotated without requiring complete annotation and checking of the bodies is a great help. Jif already provides a mechanism for incorporating existing Java libraries with minimal annotation (the annotations are called Jif signatures), but an even deeper integration of Java and Jif development environments for this purpose would be helpful.

Weakening information flow checking Finally, some control over the amount of security enforced by the compiler can be a useful switch for more general usage of security-typed languages. Although weakening the information flow checking also weakens the guarantees produced by the compiler (and compiled applications should be signed with a manifest indicating the guarantees they provide), it can be effective for more rapid prototyping and may be sufficient in certain situations. To be specific, security-type annotations can be used to enforce confidentiality and integrity. Furthermore, as previously discussed, some additional constraints can be added to prevent implicit flows along with the usual explicit flows. Although not currently implemented in any STL, it is possible to imagine that other covert channels such as timing and termination flows could also be prevented. Finally, various models of declassification have been proposed in the literature [SS05]; the programmer may wish to use different ones for different applications.
Observation: All security guarantees are not needed for every application. Application developers may benefit from starting with weaker security checking and adding guarantees incrementally as they progress through the development process.

Principle 4 (Rapid prototyping principle): STL development tools should aid the programmer in rapid prototyping of applications by allowing the temporary weakening of security enforced by the compiler and also enabling the integration of existing, non-security-typed libraries into security-typed applications. Furthermore, it is beneficial to include dataflow analyses which can prevent the compiler from being overly conservative.

3.7 Resolving Information Flow Conflicts in Jif

For the remainder of the chapter, we focus our attention on the third set of principles, which pertain to fixing information flows. We first give some background on the implementation of the Jif compiler. We then look at a particular example and the challenges faced by the programmer in using the Jif compiler to find and resolve an information flow. This example will be revisited in Section 3.8 after we describe the advantages offered by Jifclipse, our main contribution.

3.7.1 The Jif compiler

The Jif compiler is an extension module for the Polyglot extensible compiler framework [NCM03]. Jif implements a superset of the JFlow language [Mye99a], but has recently been expanded with support for, among other things, integrity policies and meet labels [CM06, MZZ+]. The phases of the compiler which are important for our purposes are the type-checking pass and the label-checking pass.

The type-checking pass verifies that the program passes Java type-safety; e.g., there are no unsafe assignments or improperly-invoked methods. Jif takes a further step in forcing runtime exceptions (such as null pointer and class cast exceptions), to be either caught or explicitly thrown, as these may cause implicit flows.

The label-checking pass is where Jif verifies that a program satisfies certain security requirements. The compiler generates constraints that force these security requirements to hold throughout each section of the program. For example, for an assignment statement to be secure, the security level of the variable being assigned to must be more restrictive than the security level of the data being stored to it. The compiler generates the constraint \( L_1 \leq L_2 \), where \( L_1 \) is the label on the data being assigned and \( L_2 \) is the label on the variable being assigned to. After generating all of the constraints for a method\(^8\), the constraint solver verifies that these constraints are satisfied and thus that the desired security properties hold.

Jif also uses a special label variable \texttt{caller\_pc} to represent the level of the program counter at the time that a method was called. It is implicitly joined to local variables in the method. This can be a frequent source of errors for beginning Jif programmers [TW04].

\(^8\) A compiler option allows checking to be done per class instead.
3.7.2 Fixing information flows

In this section, we describe a series of errors as viewed by a programmer of a Jif class. We use the Exam Room application as described previously; it has the benefit of being a simple example of security-typed language programming while also not being tailor-made to show off the features of Jifclipse.

The primary objects that interact in this example are an Exam object and a Student object (in general, there could be many student objects, but we consider only one for this example). Furthermore, there are two principals, Alice and Examiner, which are used to restrict the information flows on the object fields. The challenge of the security-typed application programmer is to properly label the fields and method headers for the classes.

The method Exam.runExam, displayed in Listing 3.1, asks each of the students the questions in the exam, tallies the number of answers that they answered correctly, and then reports that value back to the student. The programmer has labeled the lower bound of side effects for runExam as being at level {Examiner:} (this lower bound is called the begin-label), thinking that only the examiner should observe an effect of this method. He has labeled the total number of questions answered correctly by Alice, totalAlice as having {Alice:}-level security. The error given when first compiling the code is:

```
src/Exam.jif:45: The actual argument is more restrictive than the formal argument.
    if (alice != null) alice.passResult(totalAlice);
        ^--------^
```

Checking the method Student.passResult, the programmer confirms that this method, when instantiated for Alice, is labeled such that it may take an integer at level Alice:

```
public void passResult{Alice:}(int{Alice:} x)
```

This error, however, is telling him that on line 24, he cannot pass totalAlice, which he has labeled at level {Alice:} (line 6), to a function that takes exactly that security level. Using the –explain flag to generate a more detailed error message is a little better, but can still be confusing. This message can be seen in Figure 3.4.

From this information, the programmer can determine that the left-hand side of the constraint (the label on the actual argument) contains an {Examiner:} policy which is not on the right-hand side (in the label on the formal argument). This may come as a surprise, because the programmer annotated the actual argument, totalAlice, with the label {Alice:} (line 6), but the error message asserts that the label of the first argument passed to the function is not {Alice:}. Instead, the label on totalAlice is some join of {Alice:} with the caller_pc along with some other innocuous confidentiality and integrity policies.

At this point, the programmer must figure out that the mismatch is caused because the method runExam could be called from a site with a program counter that is not less restrictive than the begin-label on passResult (i.e. the callee). Seeing this, the programmer realizes that runExam may have side effects visible to principals other than

```java
public void runExam(Examiner:){ where caller(Examiner){
    IStudent[Alice] alice = this.alice;
    Question[Examiner:] pool = questionPool();
    int[Examiner:] nq = pool != null ? pool.length : 0;
    int[Alice:] totalAlice = 0;
    for (int[Examiner:] i = 0; i < nq; i++) {
        Question[Examiner:] qi = null;
        try {
            qi = pool != null ? pool[i] : null;
        } catch (ArrayIndexOutOfBoundsException e) {} //
        String qtxt = qi != null ? qi.getText() : null;
        String[Examiner:] qvars = qi != null ? qi.getVariants() : null;
        int x = alice != null ? alice.getAnswer(qtxt, qvars) : -1;
        if (qi != null && qi.isCorrect(x))
            totalAlice++;
    }
    if (alice != null) alice.passResult(totalAlice);
}
```
src/Exam.jif:45: Unsatisfiable constraint:
actual_arg_1 <= formal_arg_1
{caller_pc; Alice: ; _!: _; _; _: _; *!: } <= {Alice: }
in environment
[{this} <= {caller_pc}]

Label Descriptions
------------------
- actual_arg_1 = the label of the 1st actual argument
- actual_arg_1 = {caller_pc; Alice: ; _!: _; _; _: _; *!: }
- formal_arg_1 = the upper bound of the formal argument x
- formal_arg_1 = {Alice: }
- caller_pc = The pc at the call site of this method
  (bounded above by {Examiner: })
- this = label of the special variable "this"

The label of the actual argument, actual_arg_1, is
more restrictive than the label of the formal argument,
formal_arg_1.

    if (alice != null) alice.passResult(totalAlice);
        ^--------^  Fig. 3.4. A detailed error message provided by the Jif compiler for information leak.

Examiner, and so adds a meet-label to the method header (line 1) indicating that its side
effects may be visible either to Examiner or to Alice.

However, a second error occurs after the program is recompiled. In this case, the
compiler highlights the point where totalAlice is incremented after Alice gets a question
correct, indicating that the program counter is too restrictive at that point. The relevant
code (lines 21–22) is

    // qi is a Question owned by the Examiner
    ...
    if (qi != null && qi.isCorrect(x))
        totalAlice++;

This violation occurs because totalAlice gets incremented for each question Alice
answers correctly. This implicitly leaks information about the questions (i.e. {Examiner:}-
level information, as labeled on line 9) into totalAlice. The actual policy on the variable
after this assignment is a join of {Alice:} and {Examiner:}. To reflect this, the
programmer needs to change the declared label on totalAlice (line 6). Once he modifies
totalAlice to being at level {Examiner::Alice:}, this error goes away, but one final error
occurs.

Now the formal parameter of passResult is not restrictive enough, because it ex-
pects only {Alice:}-level information. There are two ways to handle this. The program-
mer may decide that the implicit flow from the question into totalAlice is insignificant (it
only says how many she got right in total) and add a declassifier. Alternatively, stricter
confidentiality could be maintained by restricting the formal parameter on passResult.
to reflect that it carries \{\textbf{Alice:Examiner}\}-level information now. After either of these changes is made, the program successfully compiles.

It is obvious that writing security-typed language code without a precise understanding of the data and the interfaces involved can cause major difficulties. In the worst case, it is sometimes a challenge just to understand an error. Sometimes, changing the declared label on a variable or on a method header can fix a conflict. Other times, a declassifier is the right solution for a problem. Better tools can aid the programmer in finding and repairing errors in their code.

3.8 Jifclipse

In this section, we describe the implementation of features in Jifclipse which correspond to the problems and design principles that we have identified in previous sections. Jifclipse has several additional features which we do not highlight in this section; our primary contribution is in providing tools for programmers to understand and resolve security errors in their applications.

3.8.1 Fixing information flows

Quick fixes From our experience, many label errors in Jif result from the programmer incorrectly declaring the label on a variable. This is especially true for the declared labels on the upper bound of method arguments and the begin label. If a Jif program fails to compile, then we examine each of the variables involved with the broken constraint and attempt to determine, using our inference framework, what its label should be. Though this is a fairly simple approach, it gives good results in practice and can be augmented with more advanced techniques in the future.

![Fig. 3.5. Jifclipse suggests changing the declared label on a variable.](image)
We implement this using the “Quick Fix” framework in Eclipse. Figure 3.5 shows Jifclipse suggesting that the programmer change the explicit label on a variable declaration. In the code above, the programmer has incorrectly declared `totalAli`, a variable keeping track of how many answers the student Alice has gotten correct, as having `{Alice:}` level security, when in fact this value needs to be modified using information that the Examiner has. Jifclipse suggests raising the variable `totalAli` to its correct security level.

Jifclipse can also suggest modifications to a method’s begin label or labels on method arguments. To accomplish this, however, we needed to improve the algorithm for the constraint solver. Our solver allows us to infer optimal upper bounds for procedure arguments. We briefly describe the differences between the solver provided by the Jif compiler and the solver we implemented for Jifclipse in the following section.

**Label inference** The Jif compiler already contains a simpler label inference engine for determining the lower bounds of label variables [RM99]; programmers do not need to explicitly give the types of local variables when writing their programs (though giving them makes code more explicit and easy to understand). Label variables are also used as a shortcut to represent longer types in equations; for example, the local variable declaration `int {Alice : Bob :} i = 5` produces an equality constraint `{i} == {Alice; Bob;}`. Afterwards, when the label on the variable `i` is used in other constraints, the label variable `{i}` is used instead of the explicitly declared label on the originally declared variable.

The Jif solver initially assigns all variables to `⊤`, the highest security level, and then lowers these bounds down until the constraint equations are satisfied. If a solution exists, this method is guaranteed to produce one (in fact, the least restrictive solution).

However, in order to infer what the optimal upper bounds for procedure arguments are (where “optimal” means the highest possible upper bound, such that the method checks), we need to implement a more powerful solver that can fix both upper bounds and lower bounds for a label variable. For this, we use bounds consistency solving algorithm [MS98]. We briefly summarize the implementation of this algorithm here. All variables begin with upper bound `⊤` and lower bound `⊥`; we then adjust their bounds from the label constraints as generated by the Jif compiler. For example, the equation `{i} <= {Alice;}` fixes the upper bound of the label variable `{i}` to be the label `{Alice;}`. For the equation `{i} <= {j}`, we adjust the upper bound of `{i}` to be the meet of its current upper bound and the upper bound of `{j}` (since the label of `{i}` cannot be above the label of `{j}`).

In the event that applying the constraints to the label variables results in an undetermined domain (one where not every label variable has the same label as its upper and lower bound), then the solver restricts one of the variables to a specific label and applies the constraints again. A label variable is restricted to a label depending on whether it is a “lower” or “upper” variable label. For example, local variables, return labels, return value labels, and exception labels are all “lower” labels. In contrast, method arguments and PC bounds are “upper” labels. When a label variable is chosen for restriction, it is set to its current lower or upper bound depending on whether it is a lower or upper label, respectively.
The algorithm continues in this way until the solution is completely specified. If during this it discovers an unsatisfiable equation, we backtrack and choose a different restriction. While in the worst case, this is exponential in the number of variables, we only call the inference solver with very few variables at a time while generating a fix for a broken label constraint. We plan to use the label solver to greater effect in the future; for example, adding a feature to infer all of the labels at once for a method or class.

3.8.1.1 Constraint resolution principle

The most important service provided by security-typed languages is discovering information flows that violate the application’s information flow policy. This advantage is severely reduced, however, if it is not possible for the programmer to leverage the compiler to track down and fix these information flows. This principle can be in tension with our inference principle. While the inference principle seeks to hide details from the programmer, in some cases, it is necessary also to reveal more information to the programmer in order to fix errors. Jif has already laid the foundations for this, by generating and storing detailed information about the constraints needed for type checking. We have expanded the information contained in these constraints and exposed the information to the programmer through Eclipse Views. Since the Views need not be opened or examined, we have addressed the tension between hiding information from the programmer in general and revealing more information only when an information leak needs to be fixed.

Outline View  If a Jif programmer does not give explicit security labels for data in his code, a default value is used. (The exception is local variables; as mentioned above, these are already inferred by the compiler.) Method headers have three important labels associated with them; the return value label, the return label, and the begin label. If not explicitly annotated, the return value and return labels default to the ⊥ label, while the begin label defaults to the ⊤ label. Method arguments, if unlabeled, default to the most restrictive security level.

In order to reveal these implicit labelings, we have implemented an outline viewer for Jif. In Java, the outline viewer shows the imports in a file, the classes defined within a file, and the methods defined within a class. We implement this functionality as well as explicitly showing the programmer what the implicit labels are on their fields and methods. Figure 3.6 shows the outline view of a given class, with the labels for method headers made explicit.

Constraints View  By default, the Jif label checker checks sets of constraints one method at a time; the average method can have anywhere between five and a hundred constraints which need to be satisfied. How these constraints are generated and how they interact with one another is sometimes subtle: when a Jif program fails to compile, it only reports one unsatisfiable constraint, which may or may not be the original cause of the error.

The Constraints View exposes all of the constraints associated with a piece of code and all of the constraints that share variables with that constraint. In the Figure,
Fig. 3.6. The Outline View, similar to the outline view in Eclipse, quickly provides information about the methods in a file.
Fig. 3.7. The programmer can use the Constraints View to view all of the constraints generated by a line of code and navigate through related constraints.

The Constraints View can also reveal some additional information about constraint variables by double-clicking them, namely, the label inferred for a particular constraint variable and some additional explanation provided by the compiler for how the constraint got that label. Finally, by highlighting a constraint, Eclipse automatically highlights the expression in the program code that caused that constraint (in the Figure, the highlighted constraint was generated by the initialization of the variable x, which is highlighted).

3.8.1.2 Declassifier inference principle

Sometimes a security constraint is unsatisfiable because of a fundamental conflict between security requirements. In this case, Jif allows adding a declassifier to explicitly lower the sensitivity of a security level. To aid the programmer in this, we provide a feature to automatically declassify an expression and the current program counter, if necessary.

In Figure 3.8, the programmer has written code that he believes will tell Alice her result, stored in totalAlice and declared to have security label \{Alice; Examiner:;\}. The top half is the program code, the bottom half is a window containing the Constraint View. By using a hot key, the constraints for the cursor position in the program code are revealed in the view (the view in Figure 3.7 was generated by pressing the hot key while the cursor was on the error totalAlice++). The subtrees from a constraint are all the constraint variables that occur in the constraint. The subtrees on a constraint variable are all the constraints that involve that variable. (Clearly, these trees may involve cycles, but the on-demand generation of subtrees avoids infinite loops.) In practice, it is only valuable to traverse a few levels of these constraint trees to uncover relevant constraint relationships.

The Constraints View can also reveal some additional information about constraint variables by double-clicking them, namely, the label inferred for a particular constraint variable and some additional explanation provided by the compiler for how the constraint got that label. Finally, by highlighting a constraint, Eclipse automatically highlights the expression in the program code that caused that constraint (in the Figure, the highlighted constraint was generated by the initialization of the variable x, which is highlighted).
Fig. 3.8. Jifclipse suggests adding a declassifier to resolve a problem.

However, the Student interface (as presently written), when instantiated for Alice, expects that the result told to the student is owned wholly by the student, having label \{Alice\} in this case. This causes a security violation, as we cannot lower the security label on totalAlice when passing it to a function. Jifclipse suggests adding an explicit declassifier to lower totalAlice to the expected security level of \{Alice\}. Other fixes suggested by the IDE may explicitly declassify the program counter as well or instead of the data, as necessary.

Due to some limitations with using Polyglot as a backend for an Eclipse plugin, Jifclipse currently only supports adding declassifiers for a few classes of errors. A topic of future work is modifying the backend so that automatically adding declassifiers can be done more easily. Additionally, the programmer could benefit from some extra analysis to ensure that a minimal number of declassifiers can be added to fix an error.

### 3.8.1.3 Implicit flow principle

The program counter, which taints the labels of expressions with the label of information required to evaluate that expression in the code, serves to disallow implicit flows in a security-typed language. However, as mentioned previously, the value of the program counter changes at different points in the code in ways that may not be obvious to the programmer.

The PC Label View shows the different values that the program counter takes on at different points in the code. By walking down the PC Label View, the corresponding program code that caused the PC label to change is highlighted. If the programmer double-clicks on a label in the View, the system reports exactly why the program counter has been changed to this new value.

Figure 3.9 shows the PC Label View. In this example, the programmer has mislabeled totalAlice as having \{Alice\} level secrecy as in a previous example. The PC Label View for the if statement indicates that the program counter is \{Examiner:: _r:_ x; qi; nq: for; i\}. Prior to this it was \{nq: for; i\}. Double-clicking on the label, we can see that the change was based on the use of the Examiner’s qi object.
Because the PC Label View is so reflective of the underlying compiler framework, presenting the PC label at each change, we also provide a means to highlight all the code that produces implicit flows to a particular point in the program. This is shown in Figure 3.10. Hovering over one of the highlighted regions provides information about the constraint at that point and can explain why that implicit flow was added.

### 3.8.2 Rapid prototyping principle

We have augmented the compiler to limit the security checking it requires for successful compilation in various cases. In particular, the addition of integrity in Jif 3.0 broke existing applications which compiled under Jif 2.0. By adding a `-nointegrity` flag to the compiler, we limit the power of its security checking, but increase backward compatibility and enable programmers to build applications more rapidly. A similar switch was added for confidentiality to handle applications for which only integrity is important. We leave it to future work to provide hooks in the compiler for allowing the programmer to specify even more fine-grained security specifications.

### 3.8.3 Ease-of-Use modifications

To supplement Jif’s null pointer analysis for handling runtime exceptions, we also provide a quick fix to automatically insert `try/catch` blocks for catching exceptions. Handling every runtime exception is an exceedingly time-consuming part of converting Java code to Jif code. This quick fix enables the programmer to prototype an application more rapidly.

We have also added some simple, helpful configuration tools for setting up applications to leverage Java code and Jif signatures. These project setup and configuration tools enable programmers to move more rapidly from startup to development.
public void runExam(Examiner: meet Alice: meet Bob:()=> where caller(Examiner)
{
    ISStudent[Alice] alice = this.alice;
    ISStudent[Bob] bob = this.bob;
    DEBUG.println("Running the exam...");

    Question[Examiner:] pool = questionPool();

    int[Examiner:] pool != null ? pool.length : 0;
    int[Examiner:; Alice:] totalAlice = 0;
    int[Examiner:; Bob:] totalBob = 0;

    for (int[i = 0; i < na; i++]) {
        try {
            qi = pool != null ? pool[i] : null;
            String qtxt = qi != null ? qi.getText() : null;
            String[qvars = qi != null ? qi.getVariants() : null;

            int x = alice != null ? alice.getAnswer(qtxt, qvars) : -1;
            if (qi != null && qi.isCorrect(x)) {
                totalAlice++;
            }
        }
        int y = bob != null ? bob.getAnswer(qtxt, qvars) : -1;
        if (qi != null && qi.isCorrect(y)) {
            totalBob++;
        }
    }
    catch (ArrayIndexOutOfBoundsException e) {} 

    if (alice != null) alice.passResult(totalAlice);
    if (bob != null) bob.passResult(totalBob);
}

(*) All implicit flows affecting this point are highlighted.

Fig. 3.10. Jifclipse's implicit flow highlighting: the highlighted expressions produce implicit flows into the totalAlice++ expression marked with (*).
3.8.4 Exam room revisited

Here we review how our tools help in resolving the information flows presented in the example in Section 3.7.2. With the first error, the programmer discovers that there are in fact three problematic constraints (two are related to the begin-label and the third is regarding the error that we handle last). The first broken constraint is very clear; \{Examiner\} \leq \{Alice\} does not hold. Double-clicking on this constraint reveals the reason it was added:

The PC before evaluating the call must be less restrictive than the callee’s begin label.

Furthermore, it explains that the begin label of the method’s side-effects is \{Alice\} and the PC of the call site is \{Examiner\}. This leads to the conclusion that the begin-level on runExam should be lowered with a meet-label.

For the next error, the explain message gives the broken constraint:

\{Examiner; ;_; Alice; ;_;_; ;_; ;!; \} \leq \{Alice\}

The Constraint View helps the programmer by keeping track of what variables those labels originated in and allows the programmer to view the label on each variable as a subtree of this constraint.

\{Examiner; ;_; x; qi; nq; for; i \} \leq \{Alice\}

More importantly, however, Jifclipse suggests the correct solution through a quick fix: to relabel totalAlice as \{Examiner; Alice\}.

The final error is that totalAlice is too restrictively labeled to be an argument to passResult. A look at the Constraints View reveals all the variables involved in this constraint, including explicit flows from totalAlice and implicit flows from checking whether alice is null. Jifclipse proposes a declassification as a quick fix. The programmer may decide that he wants to maintain the stricter confidentiality and change the method labels on passResult. This is ultimately a decision that must be made based on the application specification. The quick fix suggestion at least gives the programmer an intuition about what will fix the constraint.
Trusted Declassification

Policy separability is one of the main qualities we identified for making STLs effective for bridging the semantic gap in security policy. The principle of policy separation calls for policy can be separated from application code so that it can be understood, re-configured and analyzed apart from the program in which it is enforced. By lifting all security policy decisions to a separate, high-level policy file, the application benefits from easier policy reconfiguration, more tractable security analysis and easier adaptation to different system environments. STLs have typically been defined as having a separate principal hierarchy to handle principal definitions and delegation relationships. However, no policy tools for managing this hierarchy were developed prior to this work. Furthermore, while delegation relationships were discussed, no proposal had been made for how to incorporate the declassification relationships between principals into the principal hierarchy or into a separate policy definition. In this chapter, we provide tools for managing principals and their relationships, including both delegation and declassification. We also provide an analysis of the security property we can enforce in the presence of trusted declassifiers.

In security-typed languages, each data item is labeled with its security policy. For example, Alice’s password can be labeled to indicate that only Alice may read it:

\[ \text{String}_{\text{Alice}} \text{alicePwd}; \]

Principals may delegate to other principals, so this label more precisely states that Alice and those principals who act for Alice may read \text{alicePwd}. The legal acts-for relationships are typically defined in a global policy kept separate from the program, as in Figure 4.3. Given this global policy and a particular program, standard type checking enforces the property of noninterference, which informally means that throughout the entire execution of the program, only those principals to which Alice (transitively) delegates may learn the contents of her data, whether directly or indirectly. This is quite convenient for the security analyst: to understand the security implications of a particular datum, the analyst needs only to examine the label on the datum and the global acts-for relationships; she does not need to examine the entire program.
Unfortunately, noninterference is too strong a property for real programs. Consider a password check in which a guess is compared with Alice’s password:

```java
def check(guess: String, pwd: String) -> bool:
    return guess == pwd
```

What should be the label of the boolean return value? The problem is that this function reveals one bit of information about Alice’s password, which is whether or not it is equal to the guessed string. Assuming that Alice does not delegate to the public, this program would not satisfy noninterference if `guess` were `public`. But then the function is useless as a password checker, because it would not be able to inform the public user whether the password was correct or not.

To remedy this problem, practical STLs support some form of declassification, in which high-security information is permitted to flow to a low-security observer. For example, we could rewrite the above function to support declassification selectively, based on a programmer annotation, as follows:

```java
def check(guess: String, pwd: String) -> bool:
    return declassify(guess == pwd, public)
```

Another useful example is when we want to encrypt some data to send it over a public channel:

```java
def encrypt(secret: String, key: String) -> String:
    return declassify(aesEncrypt(secret, key), public)
```

While the declassify operation is efficacious, the problem with such annotation-based declassification is that we have lost localized reasoning about data security. No longer can one simply examine a data label and the global acts-for relations; now one must also find and reason about each occurrence of declassification in the program; i.e., the global meaning of the policy Alice is lost. Another way of saying this is that we can no longer reason about a global security policy (i.e., the acts-for relations) in absence of a program that uses it.

To remedy this problem, we propose the following simple idea. Rather than permit declassification on the granularity of program statements, declassification may only occur within special functions called declassifiers. The `check` and `encrypt` functions above are declassifiers. Then, individual principals indicate whether or not they trust a given declassifier as part of the global policy. For example, Alice may allow her data to be encrypted via the `encrypt` declassifier, or may wish to release her personal, medical records for scientific investigation, but only so long as the personal information is stripped out of them first by an `anonymizeMR` declassifier. On the other hand, even the small amount of information released by `check` and `encrypt` might be too much for some sensitive data.
This chapter presents a global security policy system for managing principals in a security-typed language, which extends existing work by allowing each principal to indicate which declassifiers it trusts. We call our approach trusted declassification. We add functionality for a principal, $p$, to allow a function, $f$, to declassify any of its information to new label, $\text{lbl}$ (expressed as $p \text{ allows } f(\text{lbl})$). With one of our policies in hand, the label on Alice's password regains a global meaning without having to inspect the code of the whole program. For example, if, according to the policy, Alice trusts no declassifiers, then we can be certain that $\text{alicePwd}$ is only visible to principals who act for her. If, according to the policy, Alice trusts only encrypt and check, we can check the code and types for these two declassifiers, but not the entire program, to find that negligible information is leaked via the output from each encryption or password check. We have formalized our approach in a Java-like language called FJifP, and proven a noninterference property, called noninterference modulo trusted methods, and implemented it as an extension to Jif [MZZ$^+$], a full scale implementation of a security-typed language based on Java.

There has recently been a proliferation of work toward incorporating forms of declassification into security-typed languages [MS04, CM04, BS06, MB05, MSZ06, LZ05a] as detailed in a recent survey [SS05]. Placed next to much of this work, what we propose is comparatively simple. Nonetheless, the value of our approach is borne out of practical experience. In particular, we and others [AS05] have been trying to build applications in Jif. Jif supports selective declassification [Mye99b], similar in style to the examples we presented above. Based on existing experience, many uses of declassification—such as for encryption, anonymization, authentication, and filtering—fit nicely into the framework we have proposed. Indeed, when we used our framework to build an SMTP/POP3-compliant e-mail client, JPmail, we found that it made the process of reasoning about declassification and information flows far easier. This work takes a step toward making STLS more practical.

The structure of this chapter is as follows: in Section 4.1 we give an example of a program and policy which we will use throughout the paper to describe our approach. In Section 4.2, we describe a basic object-oriented, security-typed language, FJifP with declassification and an external policy. We also give the security theorems we have proven about FJifP, namely noninterference modulo trusted methods. In Section 4.3, we describe an external, global policy definition for our system and an implementation of our system in the security-typed language, Jif.

### 4.1 Example

Consider the code in Figure 4.1. Medical records are parameterized by a principal (indicated with $<>$’s) and a medical record could be instantiated for Alice by writing the following (presuming an implicit constructor which takes arguments of the appropriate security levels to assign each of the member variables).

```
MedicalRecord<Alice> rec = new MedicalRecord<Alice>(...)
```
class MedicalRecord\(<p>\) { 
    String\_\_\_public name;
    String\_\_p: history;
    Key\_\_p: aesKey;
    String\_\_p: password;

    String\_\_p: getHistory() \{ return history; \}

    void saveHistory(OutputStream\_\_public out) \{
        out.write(AES\_\_p: encrypt(history,aesKey));
    \}

    void updateName(String\_\_public guess, String\_\_public newName) \{
        boolean\_\_public} valid = Passwd\_\_p: check(guess,password);
        if (valid) name = newName;
    \}
}

---

Fig. 4.1. A simple example

---

Alice \(\rightarrow\) DrBob
Alice allows Passwd.check(public)
Alice allows AES.encrypt(public)
DrBob allows AES.encrypt(public)
DrBob \(\rightarrow\) DrJohn
Chuck \(\rightarrow\) DrBob

---

Fig. 4.2. A simple policy

---

Fig. 4.3. Example acts-for hierarchy and declassifier context.
A medical record can release its history with the method `getHistory`, but the label on the return value, $p$, ensures that it will remain protected after it is released. A medical record can also write its history to a public stream (a socket or a file, e.g.) via the `saveHistory` method, but because the stream is public, the history must be passed through a declassifier, in this case it is encrypted with AES. Finally, using the method `updateName`, the name on the medical record can be updated by someone other than $p$, but only if that principal knows the password. Here again, declassification is needed, because the result of comparing a public value, `guess`, and a secret value, `password`, is stored in a public boolean, `valid`. Thus, the declassifier `check` is used to do the comparison and declassify the result. Principals must authorize these declassifications explicitly in the global policy.

A simple global policy is shown in Figure 4.2. Global policies express both delegations, using $\rightarrow$, and trusted classifiers, using `allows`. Given this policy, we can determine all the possible ways in which Alice’s data can flow. Anything Alice can read can also flow to Dr. Bob, because Alice explicitly trusts him (indicated by Alice $\rightarrow$ DrBob). It can also flow transitively to his partner, Dr. John. More interestingly, this policy contains all of the declassifiers which Alice will allow to operate on her data. Thus, we see that Alice’s data can flow to a public output, but only if it is first encrypted with AES. This is asserted by the `Alice allows AES.encrypt(public)` policy statement. Alternatively, Alice’s data might be leaked (a bit at a time) via a password check.

In FJifP, security is enforced statically by the type-checker, by disallowing programs which violate their policy. Consider the two methods, `updateName` and `saveHistory`. These methods utilize classifiers, `Passwd.check` and `AES.encrypt`, respectively. In order to instantiate a MedicalRecord with a principal $p$, we require that $p$ allows the use of these classifiers. Thus, given the policy in Figure 4.2, the above instantiation of `rec` for Alice will succeed, because Alice allows both classifiers. On the contrary, attempting to instantiate a medical record for Chuck would cause a type error. Note that our implementation of this in Jif has a more dynamic behavior, using dynamic checks to ensure that a principal trusts a given classifier. We explain this further in Section 4.3.2.

In this example, we can see how policy can be lifted out of a program and stored in an external file. In this way, when examining any fragment of code, we can understand the security guarantees of policy labels by consulting a centralized policy file. It is worth noting that a precise characterization of how much information can be leaked would also require inspecting the code of the classifiers. For example, consulting the code for encryption and the code for password checks readily leads to the conclusion that very little information is leaked through these methods. Since the number of classifiers for an application should not be large, it is not hard to inspect them by hand. Furthermore, a standard collection of classifiers can be built up over time with careful analyses of the information leakage allowed by each.

The challenge of automatically quantifying information leakage is being studied elsewhere [Low02, CM07, CHM07]. We expect that as these results mature, they will integrate cleanly with our system.
4.2 Semantics and properties of FJifP

4.2.1 Introduction to FJifP

We first describe FJifP (short for Featherweight Jif with Policy), a security-typed, object-oriented language. FJifP is an extension of Featherweight Java [IPW99] that includes the essential security features of Jif as well as the option for certain methods to be used as declassifiers. We then give typing and evaluation rules for that system, show their soundness, and prove a theorem about the language’s security, noninterference modulo trusted methods.

Featherweight Java (FJ) is a minimal subset of the Java programming language that models essential features of an object-oriented language such as field access, dynamic dispatch, inheritance, casting, and mutually recursive classes. It does not include many features of the full language, including mutable state, concurrency, and introspection. Conditionals can be implemented through inheritance and loops can be implemented through recursive method calls.

In giving the definition of FJifP, we seek to add security types and runtime principals to FJ in order to provide a basic framework for the Jif language. We omit some of the more complex features of Jif such as authority and unrestricted declassification (we will replace these features with our own declassification mechanism about which we can prove some security properties) as well as exceptions\(^1\). We also omit some labels from Jif which are required for checking a pc-label in order to prevent illegal implicit flows (flows introduced by the control path). Because we do not have state, we are able to capture implicit flows without the use of a pc-label. To additionally simplify the presentation of our system, we omit two mechanisms of FJ: constructors\(^2\) and unrestricted casts. These features were originally included in FJ to ensure every FJ program was also a Java program. In FJifP, it is sufficient to consider upcasts: unrestricted casting can be easily added back to the language.

Figure 4.4 shows the Medical Record Example from Figure 4.1, modified to be a program in FJifP, extended with primitives for booleans and conditional expressions.

For the most part, the code in Figure 4.4 remains the same as the pseudo-code. We presume the standard encodings for if and the existence of OutputStream, String, Key, etc. The keyword Public is a special principal having the property that Public \(\preceq p\) for all principals \(p\) and the label public being the policy \{Public:}\. There are also a few things to note involving the lack of state, static methods. First, when the original code called for modification of a medical record through an assignment statement, the new code instead returns a new medical record. Static methods (such as the call to AES.encrypt) have been replaced by creating new instances of the class and then calling that member function on them.

Because there is an illegal, implicit flow between the public string guess and the \{\(\alpha :\}\}-level password in updateName, this class cannot be type-checked without some

\(^1\)Covering exceptions in a security-typed language has been covered elsewhere in the literature [PS02].

\(^2\)The basic constructor which simply assigns input parameters to member variables is, of course, provided.
Fig. 4.4. Figure 4.1, rewritten in FJifP

notion of declassification. In this example, to correctly type the `updateName` method, we need the `check` method in `Password` to allow data to flow from `Alice` to `Public`.

There is one other technical detail to note in `updateName`. In order to simplify the semantics of FJifP, we omit including a special security label that keeps track of the current security level of `this`. Therefore, the only legal instances of the `MedicalRecord` class are ones where the two branches of the if statement return an object of the same type, and so `this` must always have the type `MedicalRecord<α>public`. The inclusion of a security level for `this` would complicate the theory and the challenges this poses are orthogonal to studying trusted declassification. However, we do not wish to restrict what security levels class instances can take on beyond what is required by the code. Specifying the security level of all class instances would be another, though more restrictive, solution to these issues [BN05].

4.2.2 Definitions

A FJifP program consists of a series of defined classes C,D,... and terms t₁,t₂,... that are to be evaluated under a series of class definitions. Terms might invoke methods, access fields, create new instances of classes, and perform casts (to name a few possibilities). Classes contain fields f and methods m. Instantiated classes are parameterized by principals p and tagged by labels l for security. The language syntax for FJifP is given
in Figure 4.5. As in FJ, the notation \( \overline{x} \) represents a list: so \( \overline{x} \) is a list of variables, parameterized \( x_1, x_2, \ldots \). The notation \( t[v/x] \) represents a simultaneous substitution being performed: in this case the value \( v \) is substituted for the free variables \( x \) in the term \( t \).

Class Names
\( C, D \)

Field Names
\( f, g \)

Method Names
\( m \)

Variables
\( x, y \)

Principals
\( p, q, r \)

Policies
\( d ::= p_1 : \overline{q}_1 ; \ldots ; p_k : \overline{q}_k \)

Labels
\( l = \{ d \} \)

Param. Classes
\( N ::= C(p) \)

Security Types
\( S, T ::= N\{l\} \)

Class Definitions
\( CL ::= \text{class } C(\overline{m}) \land N\{ \overline{S}; \overline{M} \} \)

Methods
\( M ::= S\ m(\overline{S} \overline{x}) \{ \text{return}(t) ; \} \)

Terms
\( t ::= x \)
\( | t.f \)
\( | t.m(\overline{f}) \)
\( | \text{new } S(t) \)
\( | (S) \ t \)
\( | \text{actsfor}(p, q) \text{ in } t \)

Values
\( u, v ::= \text{new } S(\overline{v}) \)

Actsfor Hierarchy
\( (p, q) \in \Delta \)

Declass. Policy
\( (m, p, q) \in \Upsilon \)

Security Contexts
\( \Theta = (\Delta, \Upsilon) \)

**Fig. 4.5.** FJifP Language Syntax

FJifP classes and terms are typed under a global security context \( \Theta = (\Delta, \Upsilon) \). The trust relations between principals are given in the acts-for hierarchy \( \Delta \). For example, if Alice trusts Bob to act for her, then we have the pair \((\text{Alice, Bob}) \in \Delta \). The declassification policy \( \Upsilon \) allows for users to specify trust relationships with higher granularity. If the triple \((m, p, q) \in \Upsilon \), then the trust relation \((p, q)\) is added to the acts-for hierarchy \( \Delta \) when type-checking the method \( m \). \( m \) then acts as an information flow from \( p \)’s data to \( q \). We define the function \( \text{extract}(\Upsilon, m) \) as follows:

\[
\text{extract}(\Upsilon, m) = \{ (p, q) \mid (m, p, q) \in \Upsilon \}
\]

\( ^3 \)It would be simple, but technically more elaborate, to specify a more fine-grained policy that only added these new assumptions while typing certain methods \( m \) inside certain classes \( C \).
We overload the extract function on security contexts in the natural way: \( \text{extract}(\Theta, m) = \text{extract}(\Upsilon, m) \) if \( \Theta \equiv (\Delta, \Upsilon) \), while the notation \( \Theta \cup \Delta' \) represents, for \( \Theta \equiv (\Delta, \Upsilon) \), the security context \( (\Delta \cup \Delta', \Upsilon) \).

Our security labels follow the decentralized label model (DLM) [Mye99b], which permits multiple policies on values. A label \( l \) is made up of policies. Each policy consists of an owning principal \( p \) together with reader lists allowed by that principal (implicitly including \( p \)). The type system ensures that all of the policies in a label are enforced, requiring a reader to appear in all policies in order to read the data. For example, let \( l \) be the label \( \{ \text{Alice} : \text{Bob}, \text{Charlie}; \text{Charlie} : \text{Bob} \} \). Alice owns the first policy, and is implicitly a reader. Bob, and Charlie are also readers in this policy. The second policy is owned by Charlie and readable by both Bob and Charlie. If a value \( v \) has been instantiated and tagged with \( l \), then either Bob or Charlie can read \( v \); though Alice owns a policy on \( v \), she is not a reader in Charlie’s policy.

The metavariable \( d \) represents a list of policies \( p : q \). These rules are given in Figure 4.6.

### Actsfor Checking

\[
\begin{align*}
\frac{}{\Theta \vdash p \leq p} \quad \text{(PLT-REFL)} \\
\frac{(p, q) \in \Theta(\Delta)}{\Theta \vdash p \leq q} \quad \text{(PLT-ACTSFOR)} \\
\frac{\Theta \vdash p \leq r}{\frac{\Theta \vdash r \leq q}{\Theta \vdash p \leq q}} \quad \text{(PLT-TRANS)}
\end{align*}
\]

### Label Comparison

\[
\begin{align*}
\forall p : q : d_1 . \exists p' : q' : d_2 . & \quad \Theta \vdash p : q \sqsubseteq p' : q' \quad \text{(SEC-LAB)} \\
\Theta \vdash \{d_1\} \sqsubseteq \{d_2\} \\
\forall q_i : q' \in q . & \quad \exists q_j : q \in q . \quad \Theta \vdash q_j \leq q_i' \quad \text{(SEC-LIST)} \\
\Theta \vdash p : q \sqsubseteq p' : q'
\end{align*}
\]

Fig. 4.6. Security Context Judgements

In FJifP, classes can be templated by principals, which introduces a principal variable \( \alpha \) that can be used within the class. When we create a new instance of a class, the templated principals are then substituted in for the principal variables of a class. Templated classes, \( \langle \mathcal{C}(\bar{\mathcal{P}}) \rangle \), are represented by the meta-variable \( \mathcal{N} \). Security types, \( \langle \mathcal{C}(\mathcal{P}) \rangle \{l\} \), are templated classes with labels attached, and they are ranged over by \( S, T \).

The function \( \text{lab} \) returns the label associated with a security type, while the expression \( S \sqcup l \) represents the security type \( S \) raised to the security level \( \text{lab}(S) \sqcup l \). The definitions for these are as follows:

\[
\text{lab}(\langle \mathcal{C}(\bar{\mathcal{P}}) \rangle \{l\}) = l \quad \langle \mathcal{C}(\bar{\mathcal{P}}) \rangle \{l\} \sqcup l' = \langle \mathcal{C}(\bar{\mathcal{P}}) \rangle \{l \sqcup l'\}
\]
As in FJ, there is a special class, \texttt{Object}, which has no principal variables, no fields, and no methods. Every other class inherits from this one.

FJifP contains a class table CT which looks up a class’s definition. We examine a class’s definition:

\[
\text{CT}(C) = \textbf{class } C(\overline{\alpha}) \triangleleft D(\overline{q}) \{ \overline{S}; \overline{T}; \overline{M} \}
\]

C is then a class with principal parameters \(\alpha\) (the bar indicates a list), which inherits from the class \(D(\overline{q})\) (some of the \(q_i\) might be in \(\overline{\alpha}\)). C has whatever fields are declared in its parent along with the fields \(\overline{S}; \overline{T}\). C also has the methods declared in \(D(\overline{q})\), along with those in \(\overline{M}\); these might override the implementation of its parent’s methods.

We define a few simple functions for future reference, to avoid continual reference to the class table in our inference rules.

- \(\text{parent}(C) = D(\overline{q})\): the parent of a class.
- \(\text{pvars}(C) = \overline{\alpha}\): the principal variables of a class.
- \(\text{localfields}(C) = \overline{S}; \overline{T}\): fields declared locally. Each field has a security type associated with it.
- \(\text{localmethods}(C) = \overline{M}\): methods declared locally. Each method specifies the security type of its arguments and the security type of the returned value.

Member methods \(m\) are declared as follows: \(S_0 \ m(\overline{S}; \overline{x})\). Then the method \(m\) takes arguments \(\overline{x}\) of security type \(\overline{S}\) and returns a value of the security type \(S_0\). We now give important auxiliary definitions for field lookup, method lookup, method type lookup, method overriding, and others. We first give these definitions for parameterized classes, then later overload their definition for security types in our inference rules; they are defined in Figure 4.7 and closely follow the analogous functions from FJ.

### 4.2.3 Subtyping

In FJ, a class \(C\) is a subtype of another class \(D\) if \(D\) is \(C\), \(C\) inherits from \(D\), or there is a \(C'\) such that \(C\) is a subtype of \(C'\) and \(C'\) is a subtype of \(D\). For FJifP, we need to define exactly what it means for a security type \(C(\overline{\alpha})\{l\}\) to be a subtype of \(D(\overline{q})\{l\}\). The combination of two observations forms our subtyping rules, given in Figure 4.8. If we have \(\text{CT}(C) = \textbf{class } C(\overline{\alpha}) \triangleleft D(\overline{q}) \{ \cdots \}\), then \(C(\text{Alice})\{l\}\) is a subtype of \(D(\text{Alice})\{l\}\) for all \(l\). Following Jif, even when \(\Theta \vdash \text{Alice} \preceq \text{Bob}\), we do not have \(C(\text{Alice})\{l\}\) as a subtype of \(C(\text{Bob})\{l\}\).

As we can always safely raise the security level of a class, \(C(\overline{p})\{l_1\}\) is a subtype of \(C(\overline{p})\{l_2\}\) if \(l_2\) is at least as restrictive as \(l_1\). Subtyping for security classes then needs to be done under a security context \(\Theta\).

### 4.2.4 Typing Rules

We are prepared to present our typing rules for terms. Let \(\Gamma\) be an environment mapping variables to security types. There are three important judgements here. The
Field Lookup

\[ \text{fields(Object\{l\})} = \bullet \]

\[ \text{localfields}(C) = S \not \exists \]
\[ \text{parent}(C) = D[q] \quad \text{pvars}(C) = \alpha \]
\[ \text{fields}(D[q[p/\alpha]]) = T g \]
\[ \text{fields}(C[p]) = (T g, S[p/\alpha]) \]

Method Typing

\[ S \text{ m}(S x) \{ \text{return}(t); \} \in \text{localmethods}(C) \]
\[ \text{pvars}(C) = \alpha \]
\[ \text{mtype}(m, C[p]) = (S \rightarrow S)[p/\alpha] \]
\[ m \text{ not defined in localmethods}(C) \]
\[ \text{parent}(C) = D[q] \quad \text{pvars}(C) = \alpha \]
\[ \text{mtype}(m, D[q[p/\alpha]]) = S \rightarrow S_0 \]
\[ \text{mtype}(m, C[p]) = S \rightarrow S_0 \]

Method Body Lookup

\[ S \text{ m}(S x) \{ \text{return}(t); \} \in \text{localmethods}(C) \]
\[ \text{pvars}(C) = \alpha \]
\[ \text{mbody}(m, C[p\{l\}]) = (x, t[p/\alpha]) \]
\[ m \text{ not defined in localmethods}(C) \]
\[ \text{parent}(C) = D[q] \quad \text{pvars}(C) = \alpha \]
\[ \text{mbody}(m, D[q[p/\alpha]]) = (x, t) \]
\[ \text{mbody}(m, C[p]) = (x, t[p/\alpha]) \]

Declared Methods

\[ \text{mbody}(m, S) = (x, t) \]
\[ m \in \text{methods}(S) \]

Method Overriding

\[ \text{mtype}(m, D[q]) = \not \rightarrow T_0 \text{ implies} \]
\[ S = T \text{ and } S_0 = T_0 \]
\[ \text{override}(m, D[q], S \rightarrow S_0) \]

Overloaded Functions for Security Types

\[ \text{fields}(C[p]\{l\}) = \text{fields}(C[p]) \]
\[ \text{mbody}(m, C[p]\{l\}) = \text{mbody}(m, C[p]) \]
\[ \text{mtype}(m, C[p]\{l\}) = \text{mtype}(m, C[p]) \]
\[ \text{override}(m, C[p]\{l\}, S \rightarrow S_0) \]
\[ \text{override}(m, C[p]\{l\}, S \rightarrow S_0) \]

Fig. 4.7. Auxiliary Definitions
Subtyping Rules

\[
\begin{align*}
\Theta & \vdash S <: S \quad \text{(S-REFL)} \\
\Theta & \vdash S <: S' \quad \Theta & \vdash S' <: T \\
\Theta & \vdash S <: T \\
\Theta & \vdash l_1 \sqsubseteq l_2 \\
\Theta & \vdash \text{parent}(C) = D(q) \\
pvars(C) & = \overline{\alpha} \\
\Theta & \vdash C[p\{l_1\}] <: D[q[p/\overline{\alpha}]\{l_2\}] \quad \text{(S-CLASS)}
\end{align*}
\]

Fig. 4.8. Subtyping Rules

first is term typing, written \(\Theta; \Gamma \vdash t : S\); under security context \(\Theta\) and environment \(\Gamma\), the term \(t\) has type \(S\). The second and third involve checking that classes and methods are well-formed. The judgement \(\Theta \vdash S \text{ OK}\) specifies that a security-tagged and parameterized class \(C\) is well-formed under security context \(\Theta\); we can view \(\Theta\) as the constraints that need to be satisfied in order to use \(C\). The judgement \(\Theta \vdash m \text{ OK IN } S\) says that the method \(m\) is well-formed within a security-tagged and parameterized class \(S\) under a security context \(\Theta\). Inference rules for term typing, class checking, and method checking are given in Figure 4.9.

Unfortunately, we must individually check that a class is well-formed at each instantiation of a security type. For example, suppose \(C\) has an integer in field \(f\) and the class \(D\) has a method \(m\) that takes an integer at \{Alice\} security level. If a method in \(C\) calls \(D.m\{\text{this.f}\}\), then this call is alternatively legal or illegal depending on the current security level that \(C\) has been instantiated to. This difficulty could be circumvented by adding a special “this” security level, bound locally within each class. We do not include such a feature for reasons mentioned above and thus we are willing to accept this checking behavior.

4.2.5 Evaluation

Evaluation in FJifP is done in a way similar to FJ, with one exception. To evaluate some terms, we need security information. The evaluation judgement is thus \(\Theta \vdash t \mapsto t'\); under security context \(\Theta\), \(t\) makes a single step to \(t'\). When we talk of a complete evaluation from a term to a value, we write \(\Theta \vdash t \mapsto^* v\), representing multiple evaluation steps. The evaluation rules for FJifP are given in Figure 4.10.

Note that there are two method invocation rules, (EV-INVKNEW) and (EV-INVKNEW-DEC). If \(\Theta \vdash t \mapsto^* v\) without using the (EV-INVKNEW-DEC) rule, then noninterference still holds and an observer cannot gain any additional information from the term’s evaluation. Otherwise, it is possible that some data has been leaked, but only through trusted declassifiers.

4.2.6 Type System Properties

With the following lemmas, we prove that FJifP is sound.

\textbf{Lemma 4.2.1 (Weakening).} Suppose \(\Theta; \Gamma \vdash t : S, \Gamma' \supseteq \Gamma, \text{ and } \Theta' \supseteq \Theta\). Then \(\Theta'; \Gamma' \vdash t : S\).
Typing

\[ \Gamma(x) = S \]  \quad \text{(TP-VAR)} \]
\[ \Theta; \Gamma \vdash x : S \]

\[ \Theta; \Gamma \vdash t_0 : S \quad S, f_i \in \text{fields}(S) \]  \quad \text{(TP-FIELD)}
\[ \Theta; \Gamma \vdash t_0.f_i : S \cup \text{lab}(S) \]

\[ \Theta; \Gamma \vdash t_0 : S_0 \quad \text{mtype}(m, S_0) = \mathcal{S} \rightarrow S \]
\[ \Theta; \Gamma \vdash \overline{t} : S' \quad \Theta \vdash S' <: \mathcal{F} \]  \quad \text{(TP-INVK)}
\[ \Theta; \Gamma \vdash t_0.m(\overline{t}) : S \cup \text{lab}(S_0) \]

\[ \text{fields}(S_0) = \mathcal{F} \quad \overline{t} \in \mathcal{F} \]  \quad \text{(TP-NEW)}
\[ \Theta; \Gamma \vdash S_0 \text{ OK} \]
\[ \Theta; \Gamma \vdash t_0 : S_0 \quad \Theta \vdash S_0 <: S \]  \quad \text{(TP-UPCAST)}
\[ \Theta; \Gamma \vdash (S) t_0 : S \]
\[ \Theta; \Gamma \vdash t : S \quad \Theta \vdash p \sqsubseteq q \]  \quad \text{(TP-ACTSFOR)}
\[ \Theta; \Gamma \vdash \text{actsfor}(p, q) \text{ in } t : S \]

Class Checking

\[ \text{for all } m \in \text{methods}(C(\overline{p})\{l\}), \Theta \vdash m \text{ OK IN } C(\overline{p})\{l\} \]  \quad \Theta \vdash C(\overline{p})\{l\} \text{ OK} \]

Method Checking

\[ \text{mbody}(m, C(\overline{p})\{l\}) = (\overline{x}, t_0) \]
\[ \text{mtype}(m, C(\overline{p})\{l\}) = \mathcal{S} \rightarrow S_0 \]
\[ \Theta \cup \text{extract}(m, \Theta) ; \overline{x} : \mathcal{S} ; \text{this} : C(\overline{p})\{l\} \vdash t_0 : T_0 \]
\[ \Theta \vdash T_0 <: S_0 \]
\[ \text{parent}(C) = D(\overline{q}) \quad \text{override}(m, D(\overline{q})\{l\}, \mathcal{F} \rightarrow S_0) \]  \quad \text{(TP-ACTSFOR)}
\[ \Theta \vdash m \text{ OK IN } C(\overline{p})\{l\} \]

Fig. 4.9. Typing Rules
fields(S) = $S \overline{f}$
\[\vdash \text{new } S(\mathcal{V}).f_i \mapsto v_i\] (EV-PROJNEW)

$m\text{body}(m,S) = (x,t_0)$ \[\vdash \text{extract}(T,m) = \emptyset\] (EV-INVKNEW)
\[\vdash \text{new } S(\mathcal{V}).m(\overline{u}) \mapsto t_0[\overline{u}/x, \text{new } S(\mathcal{V})/\text{this}]\] (EV-INVKNEW-DEC)

\[\vdash S <: T\] (EV-CASTNEW)
\[\vdash (T) \text{new } S(\mathcal{V}) \mapsto \text{new } S(\mathcal{V})\] (EV-CASTNEW)
\[\vdash p \leq q\] (EV-ACTSFOR)
\[\vdash \text{actsfor}(p,q) \text{ in } t \mapsto t\] (EV-ACTSFOR)
\[\vdash t_0 \mapsto t'_0\] (EV-FIELD)
\[\vdash t_0.f \mapsto t'_0.f\] (EV-FIELD)
\[\vdash t_0 \mapsto t'_0\] (EV-INVK-recv)
\[\vdash t_0.m(\overline{u}) \mapsto t'_0.m(\overline{u})\] (EV-INVK-recv)
\[\vdash v_0.m(\mathcal{V}, t_i, \overline{v}) \mapsto v_0.m(\mathcal{V}, t'_i, \overline{v})\] (EV-INVK-arg)
\[\vdash t_i \mapsto t'_i\] (EV-INVK-arg)
\[\vdash \text{new } S(\mathcal{V}, t_i, \overline{v}) \mapsto \text{new } S(\mathcal{V}, t'_i, \overline{v})\] (EV-NEW-ARG)
\[\vdash t_0 \mapsto t'_0\] (EV-CAST)
\[\vdash (T) t_0 \mapsto (T) t'_0\] (EV-CAST)

Fig. 4.10. Evaluation Rules
Proof. Proof proceeds by induction on the typing derivation.

Suppose \( \Theta; \Gamma \vdash x : S \); then by inversion we have \( \Gamma(x) = S \). Since \( \Gamma' \supseteq \Gamma \), \( G'(x) = S \), and so \( \Theta; \Gamma' \vdash x : S \) as required.

All other cases follow by straightforward induction on the typing assumptions. \( \square \)

Lemma 4.2.2. Suppose \( \Theta \vdash S <: T \) and let \( \text{mtype}(m, T) = S \rightarrow S_0 \). Then \( \text{mtype}(m, S) = S \rightarrow S_0 \).

Proof. The proof proceeds by induction on the derivation of \( \Theta \vdash S <: T \).

Suppose the last subtyping rule used was (s-refl); then \( T = S \) and the result follows by assumption.

Suppose the last subtyping rule used was (s-trans), so \( \Theta \vdash S <: S' \) and \( \Theta \vdash S' <: T \). By induction, \( \text{mtype}(m, S) = \text{mtype}(m, S') \) and \( \text{mtype}(m, S') = \text{mtype}(m, T) \), so \( \text{mtype}(m, S) = \text{mtype}(m, T) \) as required.

Suppose the last subtyping rule used was (s-class), so \( \Theta \vdash l_1 \subseteq l_2 \), \( \text{parent}(C) = D(\bar{q}) \), and \( \text{pvars}(C) = \bar{a} \). There are now two cases, depending on whether \( m \) is defined in \( \text{localmethods}(C) \) or not.

If not, then \( m \) is defined in \( D \) or one of its parent classes and by the second definition rule for \( \text{mtype} \), \( \text{mtype}(m, D(\bar{q}/\bar{a})) = \text{mtype}(m, C(\bar{p})) \) as required.

Otherwise we have \( \Theta \vdash m \text{ OK IN } C(\bar{p}) \{l_1\} \). We thus have \( \text{override}(m, D(\bar{q})\{l_2\}) \rightarrow S_0 \) and so \( \text{mtype}(m, C(\bar{p})\{l_1\}) = \text{mtype}(m, D(\bar{q})\{l_2\}) = S \rightarrow S_0 \) as required. \( \square \)

Lemma 4.2.3 (Value Substitution). Suppose \( \Theta; \Gamma, x : S_0 \vdash t : S \) and \( \Theta \vdash v : S'_0 \), where \( \Theta \vdash S'_0 <: S_0 \). Then \( \Theta; \Gamma \vdash t[v/x] : S' \) for some \( S' \) such that \( \Theta \vdash S' <: S \).

Proof. Proof proceeds by induction on the derivation of \( \Theta; \Gamma, x : S_0 \vdash t : S \).

Suppose the last rule used was (tp-var) and \( t \equiv y \) for some variable \( y \). If \( y \neq x \), then there is nothing to prove. Otherwise \( x[v/x] = v \) and the result follows by the assumption \( \Theta \vdash v : S'_0 \).

All other cases follow by straightforward induction on the typing derivation. \( \square \)

Lemma 4.2.4. Suppose \( \Theta \vdash S_0 \text{ OK} \), \( \text{mtype}(m, S_0) = \bar{T} \rightarrow T \), and \( \text{mbody}(m, S_0) = (\bar{x}, t) \). Then for some \( T_0 \) such that \( \Theta \vdash S_0 <: T_0 \), there exists \( S \) with \( \Theta \vdash S <: T \) and \( \Theta \cup \text{extract}(m, \Theta); \bar{x} : T, \text{this} : T_0 \vdash t : S \).

Proof. Proof proceeds by induction on the judgement \( \text{mbody}(m, S_0) \). This splits the proof into two parts: where \( m \) is defined in \( \text{localmethods}(C) \) and not.

First suppose \( S \vdash m(\bar{S} \bar{x}) \{ \text{return}(t); \} \in \text{localmethods}(C) \). By the judgement \( \Theta \vdash m \text{ OK IN } S \), we have

\[
\Theta \cup \text{extract}(m, \Theta); \bar{x} : S, \text{this} : S \vdash t : T
\]

By taking \( T_0 = S_0 \) and \( S = T \). This is the required result.

Otherwise, let \( S_0 \equiv C(\bar{p})\{l\} \). We have \( \text{parent}(C) = D(\bar{q}) \), \( \text{pvars}(C) = \bar{a} \), with \( \text{mbody}(m, D(\bar{q}/\bar{a})) = (\bar{x}, t') \), and \( t = t'[\bar{p}/\bar{a}] \). By Lemma 4.2.2, the types of the methods are the same, and since \( m \) is not defined in \( \text{localmethods}(C) \), \( \text{mbody}(m, S_0) = \).
mbody(m, T₀). By induction there exists a T₀ and S such that Θ ⊢ S₀ <: T₀, Θ ⊢ S <: T and

θ ∪ extract(m, θ); x : T, this : T₀ ⊢ mbody(m, D(q₁)) : S

T₀ and S then satisfy the requirements of the theorem.

\( \Box \)

**Theorem 4.2.5 (FJifP Type Preservation).** If Θ; Γ ⊢ t : S and Θ ⊢ t ← t', then there exists Θ' such that Θ'; Γ ⊢ t' : S' for some Θ ⊢ S' <: S.

*Proof.* Proof proceeds by induction on Θ ⊢ t ← t'.

Suppose the last rule used was \( (ev-projnew) \); then \( t ≡ new S₀(v) \) and \( S ≡ S_i ∪ lab(S₀) \).

By assumption on the typing derivation, we have Θ ⊢ vᵢ : Sᵢ' for some Θ ⊢ Sᵢ' <: Sᵢ; let Θ' = Θ. This is the required result.

Suppose the last evaluation rule used was either \( (ev-invknew) \) or \( (ev-invknew-dec) \) (the proofs are similar). We then have \( t ≡ new S₀(v).m(u) \) and \( S ≡ T₀ ∪ lab(S₀) \).

By assumption on the typing derivation, we have \( S ⊢ u : T' \) for some \( Θ ⊢ T' <: T \) such that \( mtype(m, S₀) = T → T \).

Let Θ' = Θ ∪ extract(m, θ). By Lemma 4.2.4 we have T₀ such that there exists a S₁ with Θ' ≡ x : T₀, this : Θ ⊢ t : S₁. The result then follows by value substitution; we substitute each of the arguments in for their variables and then finally the value \( new S₀(v) \) for this. This gives us a final Θ' and type S* such that we have the typing

\[ Θ' ⊢ mbody(m, S₀)[u/x, new S₀(v)/this] : T^* \]

We have Θ' ⊢ T* <: T, as required.

Suppose the last case used was \( (ev-castnew) \). Then we have \( t ≡ (T) new T₀(v) \) and \( Θ ⊢ (T) new T₀(v) : T \).

By inversion, we have

\[ Θ ⊢ new T₀(v) : T₀ \quad Θ ⊢ T₀ <: T \]

Let Θ' = Θ. As we have Θ ⊢ (T) new T₀(v) ← new T₀(v), the above gives us a typing of new T₀(v) under Θ at type T₀, a subtype of T. This is the required result.

If the last case used was \( (ev-actsfor) \). The proof is similar to the above: the typing of the actsfor expression gives a typing of t by inversion at the same type S.

\( \Box \)

**4.2.7 Noninterference**

In order to show that the desired security policies hold, we define a bisimulation relation \( ≈ \) on FJifP terms. The judgement is written Θ ⊢ t₁ ≈ t₂ : S: “under security context Θ, the terms t₁ and t₂ are observationally equivalent at security type S to an observer at security label \( ζ \)”. Noninterference is only true for programs that have been
verified to be secure by our type system. By doing this, we ensure that all information leakage is done through predetermined classifiers. This ensures non-occlusion [SS05].

Essentially, two values are equivalent at security type $S$ to the security label $\zeta$ if any equivalent operation that is performed on those values looks the same. If $S$ is at a security level above the observer, then, assuming both values are typable to subtypes of $S$, any two values “look” the same. Otherwise, any action that the values can take, notably field access and method invocation, must also “look” the same. Two terms are equivalent if they both eventually evaluate to equivalent values. For our definition of noninterference, we do not address termination leaks: one term might finish evaluation while the other diverges.

The above reasoning is formalized in Definition 4.2.6.

**Definition 4.2.6** (Observational Equivalence). Under security context $\Theta$, two terms $t_1, t_2$ are observationally equivalent at security type $S$ at security label $\zeta$, written $\Theta \vdash t_1 \approx \zeta t_2 : S$, if:

- $\Theta \vdash t_1 : S_1$ and $\Theta \vdash t_2 : S_2$ and both $\Theta \vdash S_1 <: S$ and $\Theta \vdash S_2 <: S$.

- Suppose $t_1 \equiv \text{new } S_1(\overline{v})$, $t_2 \equiv \text{new } S_2(\overline{u})$. Then:
  1. If $\Theta \vdash \text{lab}(S) \sqsubseteq \zeta$, then for all $T_i$, $f_i \in \text{fields}(S)$, $\Theta \vdash v_i \approx \zeta w_i : T_i$.
  2. If $\Theta \vdash \text{lab}(S) \sqsubseteq \zeta$, then for all $m \in \text{methods}(S)$ with $\text{mtype}(m, S) = T \rightarrow T$ and for all $\overline{u}, \overline{u'}$ such that $\Theta \vdash \overline{u} \approx \zeta \overline{u'} : T$, then $\Theta \vdash \text{new } S_1(\overline{v}).m(\overline{u}) \approx \zeta \text{new } S_2(\overline{u}).m(\overline{u'}) : T$.

- Otherwise, for all $v_1$ and $v_2$ such that without using the evaluation rule (ev-invknew-dec), both $\Theta \vdash t_1 \mapsto v_1$ and $\Theta \vdash t_2 \mapsto v_2$ then $\Theta \vdash v_1 \approx \zeta v_2 : S$.

Value equivalence only makes sense without classifier methods. Classes that use classifiers usually cannot be shown to be observationally equivalent to one another, as it would require typing the bodies of their methods under a reduced security context. This is intuitively what we want: if a class has a method that can be used as a classifier, it may not be noninterfering. On the other hand, by constructing the system in this way, we can be certain that the only points of noninterference are the points allowed explicitly in the security context. Thus, all information leakage is governed by the classification policy.

We now state the main security theorem. Suppose we have a program that is well-typed that relies on a free variable $x$. If we substitute in two observationally equivalent values to the term, then the evaluations of the program are also observationally equivalent. This captures the essence of noninterference: if we make a change in the program the observer cannot determine, then he cannot distinguish between the results of the two different evaluations.

**Lemma 4.2.7** (Security Subtyping). Suppose $\Theta \vdash t_1 \approx \zeta t_2 : S'$ and $\Theta \vdash S' <: S$. Then $\Theta \vdash t_1 \approx \zeta t_2 : S$. 
Proof. Because of invariance of typing under subtyping, we must check the second two cases in the definition of ≈.

Proof is by cases as to whether or not \( t_1 \) and \( t_2 \) are both values.

First suppose \( t_1 \equiv \text{new } S_1(v) \) and \( t_2 \equiv \text{new } S_2(w) \). Note that fields\( S' \subseteq \text{fields}(S) \), so if \( \Theta \vdash \text{lab}(S) \subseteq \zeta \), then for all \( T_i, f_i \in \text{fields}(S) \), \( \Theta \vdash v_i \approx \zeta w_i : T_i \) as required.

For all \( m \in \text{methods}(S) \), we have

\[
\text{mtype}(m, S) = \text{mtype}(m, S) = T \rightarrow T
\]

Thus for all \( \overline{u}, \overline{u}' \) with \( \Theta \vdash \overline{u} \approx \zeta \overline{u}' : T \),

\[
\Theta \vdash \text{new } S_1(v) \approx \zeta \text{new } S_2(w) : T
\]

This is the required result.

Otherwise \( t_1, t_2 \) or both are not values; so \( \Theta \vdash t_1 \mapsto^* v_1 \) and \( \Theta \vdash t_2 \mapsto^* v_2 \) with \( \Theta \vdash v_1 \approx \zeta v_2 : S' \). By the above, \( \Theta \vdash v_1 \approx \zeta v_2 : S \) and so \( \Theta \vdash t_1 \approx \zeta t_2 : S \) as required.

\[\square\]

Theorem 4.2.8 (Security). Suppose \( \Theta; x : S_0 \vdash t : S \) and \( \Theta \vdash v_1 \approx \zeta v_2 : S_0 \). Then \( \Theta \vdash t[v_1/x] \approx \zeta t[v_2/x] : S \).

Proof. Proof proceeds by induction on the derivation of \( \Theta; x : S_0 \vdash t : S \).

Suppose the last rule used in the typing was \( \text{TP-FIELD} \), so we have the following:

\[
\frac{\Theta; x : S_0 \vdash t_0 : S \quad T_i f_i \in \text{fields}(S)}{\Theta; x : S_0 \vdash t_0[f_i] : T_i \cup \text{lab}(S)}
\]

By the induction hypothesis,

\[
\Theta \vdash t_0[v_1/x] \approx \zeta t_0[v_2/x] : S
\]

Let \( \text{new } S_1(\overline{v}), \text{new } S_2(\overline{w}) \) be such that

\[
\begin{align*}
\Theta &\vdash t_0[v_1/x] \mapsto^* \text{new } S_1(\overline{v}) \\
\Theta &\vdash t_0[v_2/x] \mapsto^* \text{new } S_2(\overline{w}) \\
\Theta &\vdash \text{new } S_1(\overline{v}) \approx \zeta \text{new } S_2(\overline{w}) : S
\end{align*}
\]

We thus have

\[
\Theta \vdash t_0[v_1/x], f_i \mapsto^* v_i \quad \Theta \vdash t_0[v_2/x], f_i \mapsto^* w_i
\]

By the observational equivalence, if

\[
\Theta \vdash \text{lab}(T_i \cup \text{lab}(S)) \subseteq \zeta
\]

We must have

\[
\Theta \vdash \text{lab}(S) \subseteq \zeta
\]
Thus if the security level is observable by \( \zeta \), we have

\[
\Theta \vdash v_i \approx_{\zeta} w_i : T
\]

and so

\[
\Theta \vdash v_i \approx_{\zeta} w_i : T \sqcup \text{lab}(S)
\]

This is the required result.

Suppose the last typing rule used was (TP-INVK). By inversion we then have the typing

\[
\Theta; x : S_0 \vdash t_0 : T \quad \text{mtype}(m, T) = \mathcal{S} \rightarrow \mathcal{S}
\]

\[
\Theta; x : S_0 \vdash \overline{t} : \mathcal{S}' \quad \Theta \vdash \overline{t}' : \mathcal{S}
\]

\[
\Theta; x : S_0 \vdash t_0.m(\overline{t}) : S \sqcup \text{lab}(T)
\]

By the induction hypothesis,

\[
\Theta \vdash t_0[v_1/x] \approx_{\zeta} t_0[v_2/x] : T_0
\]

Thus if \( \Theta \vdash t_0[v_1/x] \rightarrow v'_1 \) and \( \Theta \vdash t_0[v_2/x] \rightarrow v'_2 \) with \( v'_1 \equiv \text{new } S_1(\overline{v}) \) and \( v'_2 \equiv \text{new } S_2(\overline{w}) \), then by definition

\[
\Theta \vdash v'_1 \approx_{\zeta} v'_2 : T_0
\]

By induction hypothesis on the arguments to the method,

\[
\Theta \vdash \overline{t}[v_1/x] \approx_{\zeta} \overline{t}[v_2/x] : \mathcal{S}'
\]

and so as \( \Theta \vdash \overline{t}' : \mathcal{S} \) and by Lemma 4.2.7

\[
\Theta \vdash \overline{t}[v_1/x] \approx_{\zeta} \overline{t}[v_2/x] : \mathcal{S}
\]

We then have

\[
\Theta \vdash v'_1.m(\overline{t}[v_1/x]) \approx_{\zeta} v'_2.m(\overline{t}[v_2/x]) : S
\]

The result follows.

Suppose the last typing rule was (TP-NEW). Then we have the following:

\[
\text{fields}(S_1) = \mathcal{S} \quad \Theta; x : S_0 \vdash \overline{t} : \mathcal{S}' \quad \Theta \vdash \overline{t} : \mathcal{S}
\]

\[
\Theta \vdash S_1 \text{ OK}
\]

\[
\Theta; x : S_0 \vdash \text{new } S_1(\overline{t}) : S_1
\]

By the induction hypothesis,

\[
\Theta \vdash \overline{t}[v_1/x] \approx_{\zeta} \overline{t}[v_2/x] : \mathcal{S}'
\]

So if

\[
\Theta \vdash \overline{t}[v_1/x] \mapsto^{*} \overline{v} \quad \Theta \vdash \overline{t}[v_2/x] \mapsto^{*} \overline{w}
\]
Then

\[ \Theta \vdash v \approx_\zeta \overline{w} : S' \]

By Lemma 4.2.7,

\[ \Theta \vdash v \approx_\zeta \overline{w} : S \]

The result then follows.

Suppose the last rule used was \((\text{tp-upcast})\). Then we have

\[
\begin{array}{c}
\Theta; x: S_0 \vdash t_0 : S_0 \\
\Theta \vdash S_0 <: T \\
\Theta; x: S_0 \vdash (T) t_0 : T
\end{array}
\]

Note that if \( \Theta \vdash t_0 \leftrightarrow^* v \) and \( \Theta \vdash S_0 <: T \), then \( \Theta \vdash (T) t_0 \leftrightarrow^* v \). By the induction hypothesis,

\[ \Theta \vdash t_0[v_1/x] \approx_\zeta t_0[v_2/x] : S \]

The result then follows.

Finally, suppose the last typing rule used was \((\text{tp-actsfor})\). The logic here is identical to the case for \((\text{tp-upcast})\).

Note that if the term \( t \) uses any declassifiers, then \( \Theta \vdash t[v_1/x] \approx_\zeta t[v_2/x] : S \) holds vacuously, since \( t[v_1/x] \) and \( t[v_2/x] \) cannot finish evaluation without using the evaluation rule \((\text{ev-invknew-dec})\). Suppose a term \( t \) finishes evaluation under a security context \( \Theta \); then any informational leakage that occurred must have been done through declassification methods. The sections of the program which do not involve declassification are subject to the above theorem and so they remain observationally equivalent as they evaluate to values. Those values are then used in the larger program by methods that involve declassification; after information has been safely released, observational equivalence no longer holds. In short, all information leakage can be justified by the declassification policy, \( \Upsilon \).

### 4.2.8 Related Work

Various languages have been extended with security types for statically validating noninterference, but only one other system exists using an object calculus [BN05]. FJifP differs from the object calculus of Banerjee and Naumann in several ways. FJifP does not include notions of state or permissions. On the other hand, it is closer to Jif, because security types are built directly into the language as opposed to being an inferred annotation. The differences illustrate the difference in our motives for designing the language. Namely, we are primarily concerned with showing noninterference in a simple object-oriented language with declassification. Myers describes rules for the decentralized label model as implemented in Jif [Mye99b], but does not prove security properties about the system.
4.3 Policy framework implementation

We implemented our trusted declassifiers in Jif [MZZ+]. In this section, we first describe how we compile an external policy into Jif code and access it from a Jif program. Then we comment on our approach, relating it to FJifP.

4.3.1 Compiling policy into Jif

We have developed a simple policy language for introducing principals and describing the delegations and declassifiers allowed by each principal. We built a small translator to compile policies into Jif code. The translator automatically generates principal class definitions as well as a Policy class. The Policy class instantiates these principals and establishes the delegations described in the policy. In order to use our system, a programmer must provide a policy file (such as in Figure 4.2, an application and the declassifiers mentioned in the policy file. This policy is applied to the application by adding a single line to the starting point of the application. Finally, the automatically generated files must be compiled (other than the one line inserted into the main application file, all other files in the application do not need to be changed and thus do not need to be re-compiled). This is illustrated in Figure 4.11.

![Fig. 4.11. Integrating an external policy into Jif.](image)

Our policy language currently consists of only two kinds of statements, \( \rightarrow \)-rules corresponding to delegations and allow-rules, establishing trust in declassifiers. The syntax is shown in Figure 4.12. There is a special allow rule, allow None. Since a principal must be used in a rule in order to be added to the system, a principal, \( p \), which trusts no declassifiers and has no delegations should be added with the special policy, \( p \text{ allows None} \). The policy compiler takes policies and produces Jif code. To understand the Jif code, a brief explanation of Jif Principals and Closures is necessary.
The Jif Principal class has methods for adding delegations called addDelegate and for checking authorizations called isAuthorized. Our policy compiler leverages this interface by automatically generating Principal subclasses which override the authorization method in order to authorize only the declassifiers mentioned in allow statements in the given policy file. To establish the delegations given by ->-rules, code is automatically generated for the Policy.setupPolicy method. This method instantiates each principal and uses the principal’s addDelegate method to perform the delegations given in the policy file. This gives the desirable result that, after writing the policy in a simple syntax, the programmer merely has to invoke the Policy.setupPolicy method at the beginning of an application in order to put the policy into effect.

We implement declassifiers using Jif’s Closure class. The Jif Closure class provides a way of packaging up a function with some arguments and then treating it as a first-class value. More importantly, it is parameterized by a principal, whose authorization it needs in order to execute. This authorization is sought from the principal’s isAuthorized method when it is invoked. By building Closure subclasses for each declassifier, we can be sure that all declassifications will consult the policy before executing.

Consider the example policy in Figure 4.2. Compiling this policy generates classes for AlicePrincipal, DrBobPrincipal, DrJohnPrincipal and ChuckPrincipal, as well the Policy class with a setupPolicy method that instantiates each class and performs the indicated delegations. The principals are automatically generated such that the isAuthorized method give authorization to the Closures named in the allow statements in the policy file. The AlicePrincipal class, for example, allows for the Passwd.check(public) and AES.encrypt(public) closures to operate on data labeled with a policy owned by Alice. The declassifier in the allow statements is parameterized by a principal which indicates the lowest possible security level to which the method may declassify.

Adding a declassifier One of the selling points of our system is that adding a declassifier is simple. Consider a declassifier for triple DES encryption. In our system, this would require the programmer to provide a closure to call the encryption function and do the declassification, as shown in Figure 4.13. In order to use this closure to encrypt and declassify some plaintext, the principal who owns the plaintext must authorize AESClosure. This authorization must be established through the policy file with a command such as:

Alice allows crypto.TripleDESClosure(public)
This command is automatically translated into a line of Jif code in the automatically generated AlicePrincipal class. Once this has been done, the programmer simply needs to use the declassifier by first instantiating the closure class with the particular arguments that are to be used. Then Jif’s built-in authorize method must be called with the principal and the declassifier closure as arguments:

```
principalUtilauthorize(...)  
```

This built-in method calls the principal’s isAuthorized method and if it authorizes the closure, allows the closure to be executed.

```
public class TripleDESClosure[principal P,label L]  
  implements Closure[P,{}] {  
    byte{P}[]{P} plaintext;  
    Key{P} key;  
    ...  
    public Object{this} invoke{P:}() where caller{P} {  
      return declassify( AES[{}].encrypt( key, plaintext ), {this} );  
    }  
  }
```

Fig. 4.13. A closure for declassifying the cipher text generated by triple DES encryption. The standard constructor is defined, but not displayed.

### 4.3.2 Relating the implementation to FJifP

In FJifP, typing and evaluation take place in the presence of a security context Θ, which contains an acts-for hierarchy, ∆ and a declassification policy, Υ. The implementation of the acts-for hierarchy is straightforward; all delegation statements indicated by −→-rules in the policy file are automatically generated in the Policy.setupPolicy method. We implement Υ by first defining all the principals which may be used in the program. Recall that Υ contains triples (m,p,q). These correspond to allow statements in the policy written p allows m(q). Such allow statements correspond to lines of Jif code in the particular Principal class definitions, such that exactly the methods in Υ relating to a particular principal are explicitly allowed by that principal’s isAuthorized method. For example, if p = Alice then for all triples (m,Alice,q), the isAuthorized method for the AlicePrincipal class explicitly allows closures m with return type, q. In our example, this would be AES.encrypt(public) and Passwd.check(public).

In order to faithfully implement FJifP, and achieve noninterference modulo trusted methods, we must place some restrictions on Jif’s principals and declassification mechanism:

1. We require that no declassification may take place other than through Closures. This is because all declassifications should first consult the declassification context,
which is distributed throughout the Principal classes in our implementation. Since Closures require an authorization before they may be executed, they will always consult the principal whose data they are trying to declassify, to make sure that the newly introduced flow is allowed by policy.

2. We require that no new principals are introduced other than the ones introduced in the Policy.setupPolicy method which is automatically generated from the policy file.

3. We require that no delegations are established or revoked, other than the ones introduced in the Policy.setupPolicy method which is automatically generated from the policy file.

We believe these restrictions present only minor limitations to the language. The declassification restriction does not limit the expressive power of Jif at all, since it would be possible to wrap every declassify statement in a Closure and add the appropriate allows statements to the policy. The restrictions on the principal hierarchy could be somewhat more serious. In particular, by requiring that all principals and delegations are established at the outset of the program, this would disallow dynamic updates to the security policy. Currently, however, the mechanism for dynamic updating in Jif is arguably unsafe [SHTZ06], and needs revision. Additionally, the static, global nature of the acts-for hierarchy is less critical for our approach and it is easy to imagine this restriction could be adapted to work with safe and secure dynamic updates.

One difference between FJifP and our Jif implementation is in the enforcement of the security policy. Jif is currently configured to do all delegations and policy authorizations using a runtime mechanism. Although we use this runtime mechanism, the Jif compiler could be modified to check the policy at compile-time. Our restrictions force delegations and declassifications to be static, global entities. Thus, the policy must be established at the outset of the program and the policy checks could be integrated into the type-checker, which would give static enforcement, as we have in FJifP.

4.3.3 Practical applications

Since a key motivation for our approach is the hope of gaining practical experience with security-typed programming, we have used trusted declassification to implement several significant applications, including an e-mail client (described throughout this paper), chat client, chat server, a logrotate UNIX tool and other applications are in progress. JPmail uses several declassifiers, including a variety of symmetric and asymmetric encryption declassifiers for sending sensitive data to an insecure mail server, as well as other filter declassifiers which filter e-mails for certain recipients. Principals can choose which encryption and filter declassifiers they trust, merely by changing a few lines in the policy file. Likewise, groups may be changed by merely changing a few lines in the policy file.

A pervasive theme in this work is the philosophy that declassification is dangerous and should be limited as much as possible. It is difficult to reason about, too tied to the code and complicates the security model. Perhaps analogous to goto statements [Dij68], it should be used with care, only when necessary, and, if possible, not at all. In the
cases when it is allowed, this policy decision should be marked and made explicit at a high level so as to understand what ways an application compromises the strong security provided by noninterference. In this way, we maintain the expressiveness of Jif’s policy system, but make it more flexible by separating policy decisions out from the code into a high-level policy file.

We offer an anecdote here from our experience to support the effectiveness of our model. When adding the policy to JPmail, we forgot that we had introduced a temporary work-around. When encrypting the body of an e-mail, we use skip-style encryption. We encrypt the body of the email with a symmetric encryption method and then include the symmetric key in the body after encrypting it with the public key of the principal. Prior to integrating asymmetric cryptography, so that we could encrypt the key with the principal’s public key, we had introduced a hack to simply declassify shared keys before sending them (without first encrypting the key). We placed this declassification in a closure, as required by our model. When later developing our policy, we did not think to allow it in the policy file, because we clearly did not want to permit a declassifier which declassified shared keys. Consequently, our program, obedient to the policy, refused to send any keys over e-mail. This led us to track down the deprecated closure which was correctly maintaining the security we had established in our policy file. Lifting our policy to a global viewpoint was beneficial to understanding the security enforced in our application.

Implementing policy in our model was significantly easier than managing all the complex structures provided by Jif for principals, delegations and declassifications. The ability to implement a policy by merely giving a series of delegation and allow-statements made the policy easier to construct and easier to manage. Furthermore, we found that it is quite beneficial to be able to understand all possible flows, including the fact that no symmetric keys were allowed to be declassified for any principal, by merely examining the policy file.

4.4 A Separable, Expressive Policy

The JPolicy framework presents an important step forward in reaching the goal of closing the semantic gap between high-level policy semantics and the operational semantics of an application. By compiling policy into code and utilizing Jif to verify the information-flow properties of the combination of code and applications, we are able to provably implement the policy in a program. Furthermore, the expressiveness of this policy, improved by adding trusted declassifiers, is able to capture a larger number of high-level policies that could not be captured without declassification.

The biggest advance with trusted declassifiers, however, is not so much expressiveness, but separability. Introducing a policy specification for a Jif program that introduces principals, establishes delegations and also integrates declassification is a significant advance in improving the separation of policy from application code in Jif. Such policies can be understood, modified and analyzed apart from the programs in which they are used. This facilitates high-level reasoning about program policy not previously possible, opening the door for policy reconciliation analyses such as the one described in Chapter 6, which we use in Chapter 5.
Previously, determining the security of a Jif application would require combing over all the code to find all the \texttt{declassify} statements, as done by Askarov et al. in their analysis of mental poker [AS05] and by Chong et al. for SIF [CVM07]. This is already a great improvement over other ad hoc security certification techniques, because it narrows down the escape hatches to a small number of \texttt{declassify} statements. Our approach takes this a step further, however. For our system, the security analysis only requires inspection of the policy and the code of the declassifiers.

The principal infrastructure described in this chapter represents a significant advance in separating policy specification from implementation for practical implementation of applications with provable enforcement of security policy using STLs. In the next chapter, we show how this infrastructure can be leveraged for implementing applications in a different operating system environment.
Integrating Security-typed Languages with Mandatory Access Controls

Having improved the expressiveness of policy in STLs with the JPolicy system, we turn our attention to the portability of STL applications. In Chapter 3, we showed that it is possible to implement an email client in a system with minimal support for information flow security by using encryption and PKIs. We now explore how well these languages can adapt to alternative settings, integrating with alternative operating systems security mechanisms. A particularly promising direction is in the integration with MAC operating systems that implement information flow policies, because the policies and security mechanisms they use resonate with STLs. To motivate the proper role of STLs in a MAC OS, we reflect on the security mechanism, security properties, and boundaries of MAC OSs.

5.1 Mandatory Access Controls

MAC systems are grounded in the reference monitor concept put forth by Anderson [And72]. A reference monitor must satisfy three requirements: (1) it must have complete mediation over all security sensitive operations; (2) it must be tamperproof, unable to be compromised by an attacker; (3) it must be small enough for its correctness to be verified.

In a MAC system, all processes and all data containers are labeled and the security sensitive interactions between processes and data containers are checked by a reference monitor against a global policy. The policy checks may be of the sort, “Can an application labeled X perform a WRITE to a file labeled Y?” They can also be more specific, “Can an application labeled X get the attributes from a file labeled Y?” The reference monitor for a system oversees all the applications in the system and sustains the same global security policy from end to end, even as data may be handled by one application after another. This provides for a powerful security model, because data can be protected throughout its lifetime in the system.

The essential components of MAC systems are the reference monitor, the policy, labels, processes (subjects) and data (objects). These components define important
qualities of the system as a whole, namely the security properties and the boundaries of the system.

The security properties are defined by the labels on subjects and objects and by the label semantics defined in the policy. In this work, we focus on lattice-based information flow policies, ala Denning [DD77]. Recall that for these policies, the universe of labels form a lattice. The reference monitor will allow data to flow up the lattice, but not down. In other words, if label A dominates label B, then a process labeled A can read data labeled B, but cannot write out data labeled B. Such a write would be an information leak. These are not the only policies that can be implemented by MAC OSs, but we focus on them because of the natural integration with STLs. Integrating STLs with policies such as Type Enforcement Access Control [BK85, BSS+95] used in the MAC OS, SELinux, is future work.

As we have already seen, information flow policies are both powerful and simple. They make it easy to enforce confidentiality properties such as, “Data labeled Secret will never be written to a Public socket.” Integrity properties can be enforced by enforcing the dual of the lattice, ensuring that data never flows up from lower to higher integrity. Furthermore, these powerful properties can be ascertained simply by knowing that the reference monitor is enforcing a lattice in which Secret dominates Public. Without knowing anything about the applications running in a particular system or the inputs the system may receive in the future, one can be certain this property holds on data from end to end, throughout the system. The reference monitor will ensure that every application maintains this property.

The system boundaries are defined by the extent of the reference monitor’s control. Because the security semantics of the labels are defined by the policy and the policy is only enforced over the extent of the reference monitor, the labels lose meaning as they cross outside the boundaries of the system. Thus, outgoing flows must be controlled by the reference monitor according to the global policy. Likewise, data entering the system from the outside will not (necessarily) be labeled in any meaningful way; it must be labeled as it enters into the system. The data may only enter and leave the control of the reference monitor through channels.

The edges of a MAC system are obvious boundaries, but the reference monitor also has blindspots in MAC systems that are less intuitive. Specifically, the reference monitor has no control over what happens within applications at the granularity of variable assignments (involving hardware register and memory copies rather than system calls). In some cases this requires MAC operating systems to be overly conservative in hardening applications, restricting them to handle only one security level of data. Applications, such as network-facing servers, user applications, and system utilities, may need to handle data with multiple levels of confidentiality or integrity. Using an STL application to fill these holes is a perfect fit.

The fit is perfect, because STLs utilize the same concepts as MAC OSs. An STL type checker serves as a reference monitor whose boundaries are the application boundaries and whose security properties depend on the label model and the particular policy for the application. STLs do not have concepts of subject and object, but can track where data is flowing from and where data is flowing to. The security sensitive operations tracked by the type checker include all explicit or implicit information flows
in the program. Such fine granularity is possible while maintaining efficiency, because much of the checking is done at compile-time. The compiler also ensures that all checks that must be deferred to runtime are provided.

The policy mismatch comes primarily in the label model. For MAC OSs that implement information flow properties, this is less problematic, because both the OS and the application seek to control information flows. The application can support this process by implementing the system’s information flow policy.

5.1.1 Integrating STL applications into MAC systems

A goal for this thesis is to build modular applications in STLS, meaning that the applications are to be inserted into existing systems and able to interact with non-STL applications. This should not be limited to a single setting, like the one in Chapter 3, but also portable to various environments. To investigate the portability of STLS, we explore their integration into MAC systems. Three main discoveries come from this: (1) only the STL runtime system must be changed to adapt the STLS to a new environment (this helps to motivates a more generalized, flexible runtime system infrastructure presented in Chapter 7); (2) operating system infrastructure is required to analyze and validate STL applications before entrusting them with special privileges; (3) the interaction of two applications with two reference monitors and two independent information flow policies necessitates policy reconciliation—here we use policy compliance; other reconciliation mechanisms are also possible.

To this end, we have designed and built an infrastructure that 1) allows an operating system with mandatory access controls to pass labeled data into an application and 2) to be certain that the data will not flow through the application contrary to the operating systems policy. For our investigation, we have focused on the most mature security-typed language, Jif/Pol (Jif enhanced with our policy system as discussed in Chapter 4) and the widely-studied, open source, SELinux operating system [SEL]. There is no infrastructure available in the Jif compiler distribution for interacting with secure operating systems. To remedy this, we provide 1) an API for Jif by which labeled data such as sockets, files and user I/O can be received from and passed out of the OS—this API ensures consistency between operating system and application labels. 2) We provide a compliance analysis that ensures that the labeled data will be handled securely within the application, in compliance with the OS’s mandatory policy. We integrate these changes into an operating system service we call SIESTA, that can be used to securely execute multi-level applications written in Jif, by first verifying that they will not violate the operating system’s security policy.

To demonstrate the effectiveness of our approach, we used Jif/Pol to build some prototype applications: a security-typed version of the `logrotate` utility and an email client that can handle multiple email accounts of varying security levels. For `logrotate`, we were able to determine that it is possible to have total separation between log files of different programs and between log files and configuration files, so long as the configuration files have a lesser or equal confidentiality than the log files.

The rest of the chapter proceeds as follows. In Section 5.2, we describe the problems involved with integrating security-typed languages into MAC OS’s. We give
our architecture in Section 5.3 and describe the implementation of this architecture and some demonstrative applications in Section 5.4. We evaluate these applications, as well as our approach, for its usability, efficiency and security in Section 5.5.

5.2 Problem

Far from being a contrived example in order to explore STL portability, STLs promise to be a great benefit to MAC operating systems enforcing information flow policies. Because applications act as holes for OS-level reference monitors, reference monitors must enforce especially strong restrictions on application to ensure they do not launder secret information. MAC policies normally restrict applications from handling multiple levels of data for fear the data may be copied internally, outside the control of the reference monitor, in a way that is contrary to the system policy. Unfortunately, this approach is ultimately limited, because many applications must handle multiple levels of data. To have a functional operating system, the solution is merely to trust some applications to behave with additional privileges for handling multiple levels of data. The list of applications that need such privileges is startling, though, as depicted in Table 5.1.

For this work, we explore this practical application of STLs in commodity, MAC operating systems by implementing one of these applications. We also explore the portability of STLs by revisiting the JPmail client and adapting it for this new operating environment.

<table>
<thead>
<tr>
<th>Type of utility</th>
<th>Trusted applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Policy management tools</td>
<td>secadm, load_policy, settrans, setfiles, semanage, restorecon, newrole</td>
</tr>
<tr>
<td>Startup utilities</td>
<td>bootloader, initrc, init, local_login</td>
</tr>
<tr>
<td>File tools</td>
<td>dpkg_script, dpkg, rpm, mount, fsadm</td>
</tr>
<tr>
<td>Network utilities</td>
<td>iptables, sshd, remote_login, NetworkManager</td>
</tr>
<tr>
<td>Auditing, logging services</td>
<td>logrotate, klogd, auditd, auditctl</td>
</tr>
<tr>
<td>Hardware, device management</td>
<td>hald, dmidecode, udev, kudzu</td>
</tr>
<tr>
<td>Miscellaneous services</td>
<td>passwd, tmpreaper, insmod, getty, consoletype, pam_console</td>
</tr>
</tbody>
</table>

Table 5.1. A list of trusted applications in the SELinux release for Fedora Core 5 using mls-strict policy version 20.

5.2.1 Enforcing MAC policies within applications

In an OS with MAC security, the OS can monitor its resources (such as files, sockets, etc.) and when an application tries to read them, write them, delete them, etc. it can prevent the application from performing one of these security-sensitive operations. To accomplish this, OS entities are divided into subjects and objects. Every operating
system resource (socket, file, program, etc.) is an object, labeled with a type and an MLS level\(^1\), such as `system_u:object_r:user_t:s0` for a public object owned by the reader. We will abbreviate this as `user_t:s0` since the user and role labels are always the same for system resources. The running process is considered a subject and also has a label, such as `system_u:system_r:logrotate_t:s0−s1`, where the colon-separated label consists of user, role, type and MLS level (reading the label from left to right). Notice that the MLS level may consist of a range, indicating that a particular process can handle a range of levels. In this case, the subject would have access to objects in the range `s0` to `s1`.

![Diagram of data flows through OS and application](image)

**Fig. 5.1.** As data passes from a disk through the OS into an application and again when it is written back out, adherence to system security goals requires consistency in labels and permitted flows at the OS and application levels. This, in turn, requires proper labeling and compliance of the application policy with the operating system policy.

If a running process for `logrotate`, for example, has this label and attempts to read from a user’s file labeled with `user_t:s4`, a security check will be triggered by this security-sensitive operation and (under a typical policy) it will be stopped by the Linux Security Module (LSM). However, if `logrotate` has permission to read from a log file labeled `var_log_t:s1` and to write to a configuration file labeled `logrotate_var_lib_t:s0` (which it normally does), the OS cannot stop it from reading the log data and leaking

---

\(^1\) In SELinux, “MLS” has a broad meaning with the names and semantics being drawn from a general policy. We explore this formally in Chapter 6. The meanings of all type and MLS level names are also defined in the policy, but typically `s0` is most public and `s15` is most secret.
it down to the lower security configuration file. This could leak secrets stored in the log
data to the publicly readable configuration file. Currently, the logrotate utility and the
other utilities in Table 5.1 are merely trusted not to leak data and it is not easy to verify,
by manual inspection, that the C code for these utilities does not contain such a leak.

What is needed is 1) a way to pass the security labels into the application along
with the resources, and 2) an automated way to ensure that the application honors the
flow requirements on the labels. Furthermore, both of these conditions should be checked
prior to executing the application. This situation is illustrated in Figure 5.1. Because
the second requirement is precisely what STLs do well, we consider how they might be
used to meet this need.

Consider a program that has been executed by Alice (who can enter information
through stdin and read from stdout), but that also has access to files, some of which can
not be accessed by Alice (like persistent state such as statistics from others’ executions)
and some of which are publicly readable (like config files). In Listing 5.1, data is read
from the keyboard on line 1 and properly stored in a variable labeled with Alice’s policy.
In line 2, the label on leak can be inferred as \{alice\}. Then a file is opened to write out
configuration information (which is publicly readable). A leak occurs, however, when
the program attempts to write Alice’s data out to the configuration file. This code also
contains a security violation in line 8, because statistics, which Alice should not be able
to access, are printed to the screen. The typechecker would flag these errors and prevent
this program from compiling.

Listing 5.1. Jif code with inputs from stdin and a stats file and outputs to stdout and a
configuration file.

```java
1 String\{alice\} secret = stdin.read();
2 String leak = secret;
3 FileOutputStream\[config\] conf = Runtime.openFileWrite("tool.conf",\{config\});
4 conf.write(leak); // this leak is flagged by the compiler as an error
5 FileInputStream\[state\] statsFile = Runtime.openFileRead("stats.dat",\{state\});
6 String stats = statsFile.readLine();
7 if (stats.split(0).equals("bob"))
8    stdout.write(stats); // this leak is flagged by the compiler as an error
```

Returning to Figure 5.1, we can presume that objects stored in the system are
already labeled (with an OS label, \(L_{os}\), for example), but we still need an OS API to
get the labels and provide them to the application. Additionally, this must connect into
a language-provided API to translate these labels into labels that the application can
enforce (\(L_{ap}\)). This must be a carefully controlled interface so that the labels cannot
be manipulated or spoofed. Finally, to enforce OS security goals, the information flows
that the application will enforce must comply with the information flows enforced in
the operating system. In the example, if the operating system policy were to state that
\(L_{os}' \leq L_{os}\) (\(L'\) is less confidential than \(L\)), but the application policy still has \(L_{ap} \leq L_{ap}'\)
(L is less confidential than \(L'\)), then the application would violate the operating system’s
policy.
Fig. 5.2. The lattice of principals describing all possible flows for \texttt{logrotate}. The two basic levels are logs and configuration files. The data in configuration files affect the logs but the reverse should not be true. Also, the logs should not be able to flow into each other.

Compliance testing is complicated by the mismatches between the lattices used by the operating system and those used by the application for enforcing information flow. Firstly, there may be principals in each lattice that are not found in the other. These cannot merely be removed, because they might connect shared principals and be involved in information flows. Secondly, there may be a mismatch in the kinds and granularity of permissions that the OS handles (set attribute, open, link, delete, read, etc.) compared to the application. Finally, the application policy could be more restrictive than the OS policy, but the reverse should not be true.

For example, consider the lattice we constructed for \texttt{logrotate} in Figure 5.2. \texttt{logrotate} only needs to handle two kinds of files—the log files and the configuration and state files. Furthermore, the log files should be disjoint from each other and more sensitive than the configuration and state files. In this lattice, \texttt{configP} and \texttt{logP} do not have corresponding principals in the operating system. Also, we can see that this policy is more restrictive than the OS policy (notice that \texttt{var_log_t:s1} is normally able to flow into \texttt{var_log_t:s2} according to OS policy, but not in this lattice). Also, this policy does not capture all possible OS principals. Finally, this policy only describes basic information flows—read and write.

We identify the following tasks. (1) We need a mechanism by which an application can prove that its information flow enforcement does not violate the system information flow policy. Both Jif/Pol (JP) applications and SELinux express the information flow policies that they are enforcing, so we need an approach to compliance testing the application policy against the system policy. This must include some control of application-level declassification preventing both unacceptable declassification filters and also the overuse of applications with declassifying filters. (2) We need mechanisms for the application to determine the label of its input channels necessary to enforce information flow. If a
JP application cannot distinguish between secret and public inputs, it must label them all secret to enforce information flow requirements, thus impacting usability. (3) The system must be able to determine the label of all JP application outputs. Again, the lack of an accurate label would either result in overly conservative enforcement (i.e., the application may only send secrets) or possible vulnerabilities (i.e., the application sends a secret to a public entity).

To summarize, these considerations motivate the following concrete problems:

1. How can we pass operating system resources along with their labels into an application?
2. How can we pass application data along with their labels out into the operating system?
3. How can we be sure that the application will faithfully enforce the operating system's policy on these labels?

With the solution of these problems, we have a guarantee of system information flow enforcement, based on reconciliation of information flow enforcement and accurate communication of information flow labels between application and system layers. In the next section, we provide an architecture that solves these problems. In Section 5.4, we give the details of our implementation of this architecture for Jif and SELinux.

5.3 Architecture

In this section, we provide a general architecture for solving the problems described in Section 5.2. Namely, we describe the necessary steps for ensuring that a security-typed application can handle data with a range of security levels and still enforce the information flow goals of the OS. Note that this architecture is independent of any particular language or OS. We describe, in general, the features that are required for our approach. In Section 5.4, we will describe our implementation of this architecture for Jif and SELinux.

5.3.1 Process steps

We begin with a description of the overall process and then focus on the details of various steps in the subsequent subsections. Our five process steps are illustrated in Figure 5.3.

0) Initial state The OS must have a MAC policy implementing some information flow security goals. We focus on SELinux in this paper, but this could include other high assurance operating systems such as TrustedSolaris or TrustedBSD. The key is that there must be an explicit MAC policy that is accessible to a system daemon for analysis of confidentiality policies. (Here we focus on ensuring confidentiality, but other information flow goals, such as integrity, could also be examined.)
The process for executing an application with a range of MLS privileges consists of 5 steps. The steps are performed and components are provided by different entities as shown by the different colors.

1) **Program secure application** An application developer provides the bytecode for a security-typed application along with a policy template that can be specialized by the user for a particular operating system configuration. We have used Jif/Pol in this paper, but the concepts extend to any security-typed language. A key point is that the language must provide a policy system such that each application will have an explicit policy that can be analyzed by a system daemon to understand the security lattice and declassifiers the application uses. We discuss this further in Section 5.3.2.

2) **Specialize application policy** Although a program will be developed with some basic security goals in mind, the application policy may be customized for different users running on different systems. This is especially important because the application policy must make connections to operating system label names which may not be the same from system to system. Of course, a reference policy should always be provided by the developer which should run on a default system configuration. The reference policy also serves as a template for customization to a customized OS. We discuss this further in Section 5.3.3.

3) **Invoke service** In an MLS environment, a user may have the authority to run at various security levels, but typically only logs in at one level at a time. In our approach, when he desires to run an application with a range of levels, he must first invoke an operating system service to check the application for compliance with operating system security goals. There must be no way to subvert this, i.e. to run the application without allowing the system first to perform the necessary checks. This should be enforced in the system policy.

   The operating system service performs checks based on four inputs: the system policy, the object code for the application, the application policy and the desired range of levels. We discuss this further in Sections 5.3.4 & 5.3.5.
4) **Run application** If all the checks succeed, the application has proven itself trustworthy to handle data within a range of security levels. Thus, the operating system service grants the requested security level range to the application and then launches it.

### 5.3.2 Programming infrastructure

To address the last two problems listed in Section 5.2, namely that operating system resources—both inputs and outputs—must be labeled properly in the application, operating system and language APIs are necessary. First, the operating system API must offer procedures for an application to get labels on files, sockets and other OS resources. It must also be possible for the application to set labels on resources when they are created by the application. Secondly, the security-typed language API must supply procedures for getting and creating operating system resources. The primary concern is that these API abstractions provide the only way to access operating system resources. One solution, the one we use, is to provide a single way of creating new, or opening existing resources. With this approach, when an application’s data structure is mapped onto a system resource, the internal label assigned to the data structure can be checked to correspond with the external label on the system resource. Thereafter, throughout all possible program executions, the normal security-type analysis provided by security-typed languages can ensure that that label is never violated.

### 5.3.3 Specializing application policy

Our approach assumes that a developer will construct applications that enforce some security goals. For example, a program variable into which a secret password will be read should be labeled differently from a variable that will contain public information. This must be part of the program code. The meanings of these labels are established in a high-level application policy external to the program code, however, and configured according to user preferences and system policy. For example, the public information could be treated just as secretly as the password if the user desires. Furthermore, the application developer will not know the names of security labels on the user’s operating system; these must be configured by the user in light of the operating system policy. Another consideration is that a user may prefer not to use certain declassifiers in a particular application; this should be customizable in the application policy. Once the application policy has been specialized, the policy and application can be passed to an operating system service for compliance checking and execution.

### 5.3.4 Verifying run requests

The operating system service must be on the critical path for running any security-typed application, because it will ensure that all three requirements listed in Section 5.2 are met. Namely, it will ensure that 1) the labels passed into and 2) out of the application are consistent with the operating system labels and it will ensure that 3) the application will enforce the operating system’s policy throughout its execution. To do this, the service must make four checks: (1) the application’s information flow policy must be provably compliant with the operating system policy, (2) the code should be
verified as having been compiled by a proper security-typed language compiler, (3) the
declassification filters required by the application, if any, must be acceptable for the
operating system and (4) from a global view, there is no suspicious behavior in running
this trusted process that would appear to be covert channels (such as forking dozens of
processes which might each leak a small amount of information by their existence and
through declassifiers).

The key requirement here is the first, compliance testing, which is discussed in
more detail below. The other three requirements are more general or more ad hoc—there
are no general solutions so lots of ad hoc ones are possible. They remain ongoing areas
of research. We discuss some preliminary approaches in Section 5.4.2.

Note that because the service itself is trusted to handle multiple security levels
of data, it should either be written in a security-typed language and bootstrapped into
place, or it should be small enough to be verified by hand.

5.3.5 Compliance analysis for an STL and OS MLS policy

A trusted application is given some flexibility to handle information in ways that
a normal application could not. Before granting such privileges, however, the operating
system should check to be sure that the application will not abuse them. In other words,
the application policy should comply with the operating system policy.

An application is said to be compliant if it introduces no information flows that
violate the policy of the operating system in which it is running.

Other approaches to policy reconciliation are possible here, such as modifying
both the application and system policy until they mutually compliant. We consider the
base case here and leave more advanced reconciliation to future work.

Recall that for a security-typed application all possible information flows can be
determined based solely on the high-level delegation policy, modulo some declassifica-
tions. The operating system need only check the compliance of flows that are relevant
to its own principals.

The compliance analysis between a security-typed application with high-level pol-
icy and a MAC-based operating system with static policy consists of three steps. (1)
Convert the application policy and operating system policy into a form in which they
can be compared. (2) Determine which security levels are shared between the operating
system and application. For each of the security levels, collect all allowed flows for the
application. (3) Compare these to the flows allowed by the operating system. If there
are strictly more flows allowed by the operating system with respect to shared security
levels, then the application can be declared compliant and can be safely executed.

This problem contains several challenges. One is that the OS may contain security
levels not used in the application and the application may contain security levels not used
in the OS. Another is that the OS and application may have a mismatch in the granularity
of permissions. Also, either policy could be quite large and unwieldy, making analysis
slow or even intractable. These were all problems we had to solve when implementing
this analysis for Jif and SELinux. We describe our implementation in Section 5.4.3. We
discuss compliance more formally and more generally in Chapter 6.
5.4 Implementation

For our implementation, we use the Security Enhanced Linux (SELinux) operating system [SEL] provided as part of the mainline Linux kernel. This kernel contains recent upgrades for labeled, secure sockets, integrated with IPsec [JKB+06].

Our implementation consists of three major endeavors. First, we extended the Runtime infrastructure of the Jif compiler with an interface to SELinux kernel 2.6.16 for getting and setting SELinux security contexts on network sockets and files. In order to make this configurable we added some primitives to the Jif/Pol language and implemented the changes in the Jif/Pol policy compiler. Second, we constructed the Service for Inspecting and Executing Security-Typed Applications (SIESTA). This includes a system daemon along with an interface that can be run by the user; both were written in C. It also includes a policy compliance checker which was written in XSB Prolog. Thirdly, we have utilized this infrastructure to build and test two demonstrative applications: logrotate and JPmail/SE.

5.4.1 Extensions to the Jif Runtime

The basic paradigm in Jif for labeling operating system resources is to parameterize the resource stream with a label and pass that label into the proper method of the Runtime class when opening or creating the resource. The Runtime method then checks to ensure that the label passed in by the application is acceptable (not too high, not too low) for the resource being requested. For example, the following code gets the standard output stream and attempts to leak a secret.

```java
// user is a principal passed in through main(...) Runtime[final label] rt = Runtime[getRuntime();
final label{}} lb = new label{user:};
PrintStream[stderr] stdout = rt.stdout(lb);
int{high:} secret = ...;
stdout.println(secret); // compiler flags this as an error!
```

The default Jif runtime system ensures (1) that the Runtime class, which is instantiated by getRuntime(), is parameterized only by the user who executed this program (for SELinux this would be the security context of the program). Jif also ensures that when creating a stream for stdout, that the stream is parameterized by a label which is (2) equivalent to the label passed as a parameter to rt.stdout() and (3) no more secure than the label on the Runtime class. This corresponds to the notion that we should be able to print public data or user data to the user terminal, but nothing more secret than this. Note that these are policy decisions may be altered in other systems. We provide a means for modifying this using Channels and SPolicy, described in detail in Chapter 7.

Following this paradigm, we extended the Jif Runtime to handle labeled IPsec sockets (both client and server sockets) and labeled files. The basic concept is straightforward: we provided openSocket, openServerSocket, openFileWrite and openFileRead methods which ensure that the streams attached to these operating system resources are properly labeled within the Jif application. The details of this implementation required
some additional work, however, in order to properly interface with the SELinux API for
getting and setting security contexts on sockets and files.

The code for implementing the socket interface was particularly challenging, be-
cause of the way labeled IPsec handles SELinux security contexts and IPsec security
associations (SAs). Namely, in order to provide the proper cryptographic protocols for
a particular socket, it is necessary that the application first creates the socket, then
assigns the proper security context and then attempts to make a connection (it must
occur strictly in that order). At that point, the IPsec subsystem attempts to establish
an SA for the given security context, local host and port number and remote host and
port number.

The problem is that the standard Java Socket API (on which we must build for
Jif) does not distinguish between creating a socket file descriptor and attempting to make
a connection. Consequently, we had to extend the Socket class to implement our own
SelinuxSocket and SelinuxServerSocket. The constructors for our new classes can take
a security context. Then, when the socket attempts to establish a connection, a shim is
inserted between socket creation and socket connection that calls into the SELinux API
to change the socket’s security context. After connection, the Runtime class ensures that
the SA retrieved for the socket corresponds to the security context that it was set to.

The rest of the code in Runtime.openSocket(...) follows the model of stdout(...),
checking to ensure that the label which is attached to the Jif Socket object corresponds
to the security context attached to the operating system resource. The difference is that
because sockets are always two-way in Java, the label must be equivalent (neither higher
security nor lower security) to the SELinux security context.

Extending the Jif policy system Because Jif and SELinux use different kinds of
labels and principals, we needed to make some connection between them. We handled
this by extending the Jif/Pol policy language with an operator, $\cdot$, to signify an operating
system label and also wild cards to match a series of labels. This new operator takes
regular expression patterns and matches these against labels provided by the system.
Because multiple matches might occur, our semantics specify that the first match is
always chosen.

Consider the lattice for logrotate in Figure 5.2. The SELinux principal logrotate_var_lib_t:s0\(^2\) is at a lower secrecy level than the Jif principal configP. With our new policy syntax, we can express this relationship as
$\cdot:*\cdot:logrotate_var_lib_t:s0 \rightarrow configP$. The Jif Runtime does not create principals
in advance to correspond with every operating system principal; it only creates them as
needed (e.g. when a file labeled with that principal is opened). Furthermore, when they
are created they are assumed to be unrelated (incomparable) to any other principals in
the lattice. Thus, the effect of the policy statement given above is that a hook is inserted
into the Runtime to watch for any principals matching this wildcard and when one is
created, it will be properly placed in the lattice.

Recall that logrotate_var_lib_t:s0 is an abbreviation for
system_u:object_r:logrotate_var_lib_t:s0.
When the relationship is reversed and a Jif principal must be lower in the security lattice than an SELinux principal, the relationship is stored in the Jif principal at the time of its creation (the beginning of the program). For example, if we have the policy statement \( \logP \rightarrow [\ast:\ast:\ast:\ast] \), placing the Jif principal \( \logP \) lower in the lattice than any SELinux principal. Note that this policy statement does not presume anything about the relationship between different SELinux principals that match the same wildcard—only that each of them is higher than \( \logP \).

To summarize, the lattice policy we used for \( \logrotate \) is small and concise (it is the lattice that was illustrated in Figure 5.2):

\[
[\ast:\ast:\ast:\ast: \logrotate\_var\_lib\_t:s0] \rightarrow \text{configP}; \\
\text{configP} \rightarrow \logP; \\
\logP \rightarrow [\ast:\ast:\ast:\ast];
\]

**5.4.2 SIESTA**

![SIESTA Diagram]

Fig. 5.4. SIESTA: a service which securely validates and executes trustworthy security-typed applications.

The Service for Inspecting and Executing Security-Typed Applications (SIESTA) consists of two parts: a service interface and a system daemon, as shown in Figure 5.4. The service interface takes two inputs from the user—the security-typed application to be executed and the desired MLS range at which it should be executed. The service interface calls a long-running system daemon to carry out the checks described in Section 5.3.4. If everything is acceptable, SIESTA proceeds to execute the Jif application with the special MLS privileges. In the following, we describe SIESTA’s operation in more detail.

The SIESTA service interface starts running with the same MLS level as the process that called it. Running the interface also causes an SELinux transition into the `siesta_t` domain which limits the process’s functionality to communicating with the daemon and forking a new trusted Jif application. The communication between
the interface and the daemon is supported by IPC mechanisms plus security functions, creating what we called a SIESTA channel. Furthermore, supported by OS policy we make the siesta_t domain the sole gateway for executing Jif applications in a domain with special MLS privileges. This guarantees that the user cannot directly run a Jif application with special privileges unless it has first been checked by SIESTA. The logic in the service interface is quite simple, deferring the complex considerations to the system daemon.

First, the SIESTA system daemon is responsible for ensuring that the Jif application it has been given is trustworthy. It must first ensure the “Jif-ness” of the application bytecode archive (jar). Doing this in a general way is really an orthogonal issue and a research topic in itself, so here we take a straight-forward approach and just validate a jar signature against a potential list of third-party, trusted Jif compiler signatures.

Once the Jif-ness of the application has been established, the policy jar that has been passed to the daemon must be checked for compliance against the system policy. The jar contains the Jif-compiled policy files (principals and policy store) that will govern the application while it executes. It also contains a manifest of the policy from which it was built. The policy compliance check is described in more detail in Section 5.4.3.

Thirdly, the SIESTA daemon ensures that the declassifiers used by the application are acceptable to the operating system. Although there are no general solutions to this problem, we provide a preliminary approach. Before running an application, a compliance algorithm should check the declassifiers used by the application against a white-list or black-list of declassifiers. For example, standard military procedure prohibits the use of DES for protecting secret data. This can be easily checked in the application policy and applications that violate such requirements can be prevented from executing. As more security-typed applications are used, a list of trusted declassifiers can be established and become a more natural part of the operating system policy. Also, some applications, like logrotate, don’t need any declassifiers at all. Other approaches could also be feasible here, taking advantage of ongoing research in quantifying the information leaked through declassification [Lau01, Lau03, CHM07, CM07]. We discuss this further in Section 5.5.1.

Lastly, although we do not attempt to implement any particular policy for eliminating the covert channels which could be created through the execution of hundreds of these security-typed applications, we provide hooks that could be used as such policies are developed.

The SIESTA daemon must be executed by a system administrator prior to executing any security-typed applications. It must run with a full range of MLS privileges so that it can handle security-typed applications of all sensitivities. At the same time, it can be limited to a fairly constrained functionality, because it only needs to read from files and communicate with the SIESTA service interface via IPC.

5.4.3 Compliance analysis for Jif and SELinux MLS policy

A more general and more formal presentation of policy compliance is provided in Chapter 6. Here we describe how policy compliance is specialized for comparing Jif policies to SELinux policies.
There are a few key challenges in attempting to determine compliance between Jif policy and SELinux policy. The foremost challenge is in the semantic difference between Jif’s information flow lattice and SELinux’s MLS constraints. Although SELinux claims to have an MLS policy (which normally means a lattice-based policy enforcing the ⋆-property and simple security property), the so-called “MLS” extension is really a richer policy language and can be used to implement MLS, but can also implement more general policy goals. The second challenge lies in the difference in granularity between Jif policy and SELinux policy. While SELinux policy distinguishes various operations and resource types (the policy for setting the attribute on a file could be different from writing to a socket, for example), Jif policy gives a more comprehensive view of all information flows in an application. The third challenge lies in the size of the SELinux policy for a whole operating system. The standard, complete operating system policy consists of well over 20,000 policy statements.

For the first challenge, analyzing policy compliance would have been a straightforward lattice comparison if not for the generality of the SELinux MLS policy. Thus, some policy analysis tools are needed to determine what information flow goals are implemented by the operating system and whether they are compatible with the information flows in the application we are seeking to execute. Although several SELinux policy analyzers exist, none were suitable for our purposes, because none handles the new SELinux MLS extensions which were our primary concern. Consequently, we developed our own policy analysis tools for SELinux MLS policy, inspired by the policy analysis engine, PAL [SSS04]. Our tool, called PALMS, determines what information flows are allowed between MLS levels. We describe this analysis in more detail in Chapter 6, including a formal consideration of correctness. For this work, we extended and utilized PALMS to compare the flows allowed in a Jif application to the flows allowed in the host operating system.

The second challenge is that in order to compare the operating system policy and the application policy, they must be in a comparable form. Since SELinux policy is more expressive than Jif policy, we translate our Jif policy into an SELinux policy. This also allows us to reuse our policy analyzer for both policies.

For example, consider the Jif policy in Figure 5.5 in the box labeled app-policy.jifpol. This policy says that the Jif program has access to operating system files and network sockets. Also, it allows data to flow from pub to siis to sec. Furthermore, the policy states that pub is equivalent to the security level s0 and sec is equivalent to the security level s1, while siis has no corresponding identity in the operating system.

Given this policy, the contents of an operating system file at level s0 could be read into the application at level sec (through a read-down) and then written out to a file at level s1. This constitutes a flow from s0 to s1. This flow must then be checked against the operating system’s policy to determine if it is an allowed flow. Note that although siis has no corresponding principal in the operating system, it cannot simply be ignored, because it could be involved (as in this case) in a flow between two operating system principals. At the same time, the flow from pub to siis need not be checked against the operating system policy, because the siis principal does not correspond to
an OS principal. Only when both endpoints of a flow have corresponding OS principals does the flow need to be checked against the OS policy.

Next, if the Jif application asks for file_read_access and file_write_access, we then add, respectively, read-down rules and write-up rules for file access, giving an SELinux-style policy as shown in the box, app-policy.conf. We add similar rules for user I/O if stdio_read_access and/or stdio_write_access are set and for sockets if net_read_access and/or net_write_access are set in the Jif policy.\(^3\)

The third challenge we faced was the magnitude of the operating system policy, which threatened to make the analysis intractable. We are able to manage this in several ways. Firstly, once the policy has been compiled into Prolog, it need not be compiled again. Furthermore, XSB Prolog has some efficient methods of storing the policy, using tabling, which improve performance for analysis. Most importantly, however, we are able to radically reduce the analysis of the operating system policy by first analyzing the application policy. Because we are only interested in verifying that the flows allowed by the application are also acceptable to the operating system, we don’t need to check all operating system flows—just the ones that intersect with the application.

5.4.4 Sample applications

We have implemented two sample applications in order to demonstrate the range of functionality provided by our architecture. The first is logrotate which demonstrates proper labeling of files and tracking of information flows from multi-leveled files handled within the same application. This is an example of a secure implementation of an operating system tool and demonstrates features that would be common to many of

\(^3\) For brevity and clarity of presentation, we have given a truncated version of the policy, but to be complete, our implementation includes all the write-like and read-like operations necessary. For the same reason, although our implementation also handles category sets, we forego a discussion of that here.
the utilities listed in Table 5.1. The second application is larger and more complex—a modification of the JPmail email client presented in Chapter 3. For this work, we migrate this client from using a PKI for achieving end-to-end confidentiality to using the SIESTA infrastructure with labeled IPsec.

5.4.4.1 logrotate

The logrotate utility is regularly executed by cron to gradually phase out old log files. The utility rotates each set of log files based on some configuration. For example, the messages log is renamed to messages.1, messages.1 to messages.2, etc. The configuration specifies which logs to rotate and each log has a rotate attribute indicating how many back logs to save. The full version of this utility can also run scripts, compress logs and send emails. We did not implement these additional features, but chose to focus on the essential functionality of log rotation.

The logrotate program handles a variety of sensitive information flows (an example lattice is shown in Figure 5.2 and the lattice policy is given in Section 5.4.1). It handles two files which are (typically) publicly readable: a configuration file and a state file. It handles various other log files at various security levels, creating and modifying the files as needed in order to rotate and delete logs according to their particular configurations. The data in the log files is more or less secret depending on the nature of the log. For example, the logs for X Windows and wtmp are usually publicly readable. Other logs such as secure or maillog are more secret due to their contents. On the other hand, the attributes of the log files (e.g. seeing that they exist, getting their names, reading their last date of modification, etc.) are public.

In order to rotate logs, the program needs to read configuration information and state information and based on that, the logs themselves are renamed. This effectively passes information from the configuration files into the log files (it is clear from looking at the directory, for example, what the rotate attribute is for each log—usually it is the highest filename extension, like the 4 in messages.4). Thus, in order for our application to function properly, the level of the configuration data must be lower or equal in the lattice, i.e. less secret, compared to the level of the log data. On the other hand, we do not want to leak log data into the configuration file (since it is often publicly readable) or into other log files. In fact, our Jif application verifies that this policy can be upheld: not even small bits of information released by control flows are leaked from the log files into the configuration files and not even a single declassifier is needed to implement this system utility.

5.4.4.2 Email client, JPmail/SE

The JPmail application, described in Chapter 3 is an email client built in Jif, using our Jif/Pol policy framework, described in Chapter 4, that enforces information flow control on emails according to a given Jif policy. JPmail utilized encrypting declassifiers to send out email on public networks. By utilizing labeled IPsec sockets and trusting

\footnote{\texttt{wtmp} is queried by the UNIX command \texttt{last}.}
the operating system to handle distributed security (i.e., the MAC OS security ensures that emails are not leaked from intermediate or destination servers), we were able to remove the cryptographic infrastructure from JPmail and significantly simplify the code. Furthermore, we were able to extend our mail client to handle communication with mail servers at multiple security levels within a single process.

While this application serves to demonstrate the usage of client sockets, the real significance of this application is mainly in its size and complexity. It is the largest existing Jif application and so it gives us some insight on the difficulty of augmenting a realistic application to work with SIESTA. In this vein, we were gratified to discover how much cleaner and simpler the code became when it could trust the operating system to handle security concerns over its resources.

Also significant about this application is its use of declassifiers. This is due to the fact that it gets user input for all levels of email accounts from the same terminal window. The application logic handles the proper downgrading of input when responding to a public as opposed to a secret email. Another, minimal source of leakage is through an implicit flow caused by handling both public and secret email accounts in the same user interface loop. This small flow that normally occurs when a single user interface is used for multi-level inputs is handled with a declassifier. The declassifier and its use in the code must be determined to be safe for the email client and then it is included among the white-list of declassifiers in the operating system policy.

5.5 Discussion and evaluation

5.5.1 Declassification

One area of integration between STLs and MAC OSs that requires further study regards declassification. The Bell-LaPadula model quickly faced the practical limits of noninterference and needed to introduce the concept of Trusted Subjects. Likewise, MAC OSs such as SELinux provide similar mechanisms. The MLS policies in SELinux give special privileges to special classes of applications so that they may serve as filtering, trusted subjects for secure data that must be downgraded. Depending on the downgrading, requiring a separate application call for every declassification may be too heavy-weight, but it is imaginable that our trusted declassifiers could be separated from applications and implemented this way.

Another alternative would be that the compliance test could check for what declassifications in an application (all must be listed in the policy, so this is easy to do) against a white-list or black-list of declassifiers. For example, standard military procedure prohibits the use of DES for protecting secret data. This can be easily checked in the application policy and applications which violate such requirements can be prevented from executing. As more security-typed applications are used, a list of trusted declassifiers can be established and become a more natural part of the operating system policy. Also, some applications, like logrotate, do not need any declassifiers at all.
Table 5.2. Time (ms) to perform one rotation of 50MB of log files in both Jif and C, and time (ms) to start up the Jif process using SIESTA (includes Jif-ness validation and compliance checking).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Configuration</th>
<th>Median</th>
<th>Mean</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>logrotate</td>
<td>C</td>
<td>7.923501</td>
<td>7.943820</td>
<td>0.133496</td>
</tr>
<tr>
<td>logrotate</td>
<td>Jif</td>
<td>13.949643</td>
<td>13.925600</td>
<td>0.122477</td>
</tr>
<tr>
<td>SIESTA</td>
<td>compliance</td>
<td>241.060289</td>
<td>252.830086</td>
<td>25.025038</td>
</tr>
<tr>
<td>SIESTA</td>
<td>cached</td>
<td>31.639957</td>
<td>32.368633</td>
<td>3.353408</td>
</tr>
</tbody>
</table>

5.5.2 Performance

In this section we consider performance costs associated with the approach outlined in the preceding sections. We stress the preliminary nature of the implementation, experiments, and test-bed. Because of the unique and cross-cutting nature of these experiments, it is highly difficult to isolate performance cost (simultaneously at the application, OS and network layers). Experimental error is caused by interactions between the OS (process scheduling, interrupts), network delays, Java (garbage collection and dynamic class loading), and other system services and applications (process interference). Hence, we focused our initial experimentation toward obtaining a broad performance characterization of the design, leaving more precise evaluation and the invention of apparatus to achieve it to future work.

We study the overheads associated with information flow controls at the application (Jif) level. We compile the Jif programs using Ahead-of-Time (AOT) compilation with the gcj compiler v. 4.1.1 with classpath 0.92 [Sal06]. We examine the operation, logrotate, which renames up to four log files per set and as many sets as requested; we compare our Jif application with the latest C version 2.7.1, using only minimal features of the applications. The tests were run between two identical 3GHz Intel hosts running SELinux Kernel version 2.6.16 with 1GB memory. All experiments were repeated 100 times. Table 5.2 provides macrobenchmarks for rotating forty log files totalling 50MB.

5.5.2.1 Sample applications

In the case of logrotate, the Jif application consistently runs 2x slower than the C version. We tested the two programs with various log files of different total sizes. Since there is no inspection of the contents of the files, the size is the determining factor. The displayed result is for a standard complement of log files totalling 50MB. For this utility, the decrease in speed is inconsequential. The logrotate application is generally a cron job that only runs daily or even weekly. Additionally, our Jif code is entirely unoptimized and could be improved.

5.5.2.2 Compliance testing

For SIESTA, the overheads are constant and small. The policy compliance check requires a call into the XSB prolog interpreter but executes relatively efficiently, requiring only 15.512256 ms on average. 5.577328 ms of this time is spent in loading the policy (both for the application and the OS) while 8.902952 ms is spent doing the flow checks.
XSB prolog is highly optimized and the prolog source files can be compiled for greater efficiency. The majority of time is spent in signature validation for the jar file that is being loaded.

Fortunately, this validation process (checking Jif-ness and checking compliance) is a one-time cost when first verifying the compliance of the application and its policy. This process only needs to be repeated if the application changes (for a new version), if the application policy changes or if the OS system policy changes, all of which should be rare events. Otherwise, the service may compile the jar file together with its policy into a binary executable and the hash of the binary can be cached for future executions. Checking the hash is almost an order of magnitude faster than validating the jar signature and checking compliance (32.368633 ms vs. 252.830086 ms averaged over 100 runs).

5.5.3 Practicality/usability

What we have implemented is a prototype using Jif as a basis for trust when constructing trusted applications for a secure operating system. The guarantees that we are capitalizing on are not specific to Jif, but are part of the static type analysis that Jif implements for security types. Jif has some problems. For example, Jif is built on Java and requires that the JVM be loaded. This may not be desirable for some applications. Furthermore, the JVM introduces a large amount of code into the trusted computing base. Fortunately, this is not an inherent limitation to our approach because the security-type analysis we depend on is orthogonal to the JVM (and any security goals it may implement).
Chapter 6

Analyzing Policy

Applications do not occur as stand-alone closed systems, but generally interact with other applications and users through devices and services controlled by the operating system. When inserting an application into the context of an operating system, as is the case with logrotate in Chapter 5, or inserting a new application component into a distributed system, as in the case of JPmail in Chapter 3, the application will need to interact with the operating system. In this case, if the application wishes to trust the operating system to provide any security on its inputs and outputs, and/or the operating system wishes to trust the application to handle its security policy in some way, policy analysis is required to understand how the application’s policy interfaces with the operating system policy.

This provides additional motivation for creating STL applications with separable policy. Not only can the policy be re-configured easily, but it can also be studied more easily and understood apart from the application itself. Likewise, operating systems that implement MAC security allow for automated reasoning about the security of a whole system, because the whole system policy is available in a centralized policy file.

It is easy to imagine that various policy analyses may be helpful for reconciling operating systems policy and application policy. For example, an application may wish to ensure that the operating system policy is at least as strong as the application policy. Very important calculations performed within the application may necessitate extra protection on outputs. If these protections cannot be provided by the operating system, the application may wish to employ other mechanisms to protect the information policy on its data, such as encryption. In some other cases, the operating system policy and the application policy may be highly divergent and an analysis could be used to recast the operating system policy as an information flow model before attempting to compare it with the application policy (this would be necessary if we were trying to analyze an SELinux policy other than the MLS policy, for example).

In this chapter, we solve the base case problem, comparing information flows in the SELinux MLS model against information flows allowed in a JPolicy for a Jif application. The requirement we seek to enforce here, following the criteria established in the previous chapter in Section 5.3.5, is to ensure that the application’s information flow policy is at least as restrictive as the operating systems information flow policy. We
solve this problem specifically for SELinux and Jif. Before we can describe the analysis, however, it is first essential to understand the semantics of the SELinux MLS policy and then formulate the compliance algorithm.

6.1 Introduction to SELinux MLS Policy

SELinux seeks to fully specify the principle of least privilege on modern operating systems using a mandatory access control (MAC) security policy. To accomplish this goal, the SELinux policy system has combined three different policy models: Role-based Access Control (RBAC), Type Enforcement (TE) and multi-level security (MLS).

While the TE policy can be used to control the integrity of information flows [JEZ03] (i.e. where information flows from), an MLS policy is designed to control the confidentiality of information flows (i.e. where information flows to). In particular, an MLS policy is meant to prevent the leakage of information from more secret sources to less secret channels. Protecting against such a leakage of information is especially important to nearly all government and military sectors, who widely use the MLS model. With the widespread occurrence of electronic data theft, costing individuals and institutions billions of dollars in damages and lawsuits, MLS policies may find increasing use in other sectors as well.

Perhaps anticipating such a broad usage, the MLS policy language in SELinux is general enough to express a wide variety of confidentiality policies. The problem is that the MLS policy language is so broad that it is not easy to determine exactly what information flow goals are enforced by a given policy. For example, we may want assess a policy’s coverage, to know whether all possible information flows are constrained by the policy. Also, it may be important to know that the policy faithfully implements standard high-level goals, such as the simple security condition (no read-up) or the \(\ast\)-property (no write-down) as defined by Bell and LaPadula [BL75]. Finally, there are cases when it is valuable to know that one MLS policy is compliant with another. For example, in a distributed system when a new machine joins a trusted group it is important to determine that the new machine will faithfully enforce the policy goals of the group [JMS+06]. Thus a policy compliance test is warranted.

Performing such an analysis is not easy, however. The standard reference policy contains hundreds of lines of policy statements, constraining access of some 40 kernel objects that may be accessed in almost 50 modes. A manual analysis of this policy is impractical. This is further complicated by the lack of a logical presentation of the semantics of the policy. While the RBAC and TE models have existed in SELinux for many years and have been studied at length [GHR05, JEZ02, JEZ03, Smo02, SSS04, ZM04], the MLS model in SELinux is quite new [Han06]. Since the MLS model is largely orthogonal to the TE model, existing analyses for TE cannot be applied to it. What is still needed are a formal policy semantics by which we can reason about MLS policy and an analysis tool to aid in this process.

Consequently, in this chapter, we present the first logical specification for modeling SELinux MLS policy. We use this specification to develop analyses for determining 1) all information flows allowed in a given policy and 2) whether one policy is compliant with another. Finally, we implement the specification and analyses in Prolog in an analysis
tool called PALMS (for Policy Analysis using Logic for MLS in SELinux). PALMS takes two policies in SELinux MLS policy syntax and automatically determines all the information flows allowed in the policies as well as whether one policy is compliant with the other.

We have found PALMS to be valuable for various tasks. First, we were able to determine that the reference MLS policy covers all possible classes of objects and modes of access. Second, we have used this analyzer for determining compliance of the SELinux reference policy with a standard military policy implementing the \(*\)-property and simple security condition. Thirdly, we have also used this analysis tool for determining the compliance of an application’s MLS policy with the MLS policy of its host operating system. The implementation was discussed in Section 5.4.3.

In the next section, we give some background on SELinux policy along with a motivating example. In Section 6.3, we give a logical specification for an SELinux MLS policy model. We use this model in Section 6.4 to describe some algorithms which determine information flows for a given policy and also check compliance between two policies. In Section 6.5, we describe our implementation of the model and analyses in the tool, PALMS, and give some test results, namely that the reference MLS policy for SELinux is, in fact, compliant with the standard \(*\)-property and simple security condition.

6.2 SELinux Background

SELinux implements three security models: the Type Enforcement (TE) model, Role-Based Access Control (RBAC) Model and Multi Level Security (MLS) model [Sma02]. First, every element in the system is associated with a class (file, tcp_socket, ipc, process, etc.) and security sensitive operations are divided into modes of access (read, write, open, connect, getattribute, etc.). Both the TE and MLS models use these classes and modes to determine what accesses are granted or denied.

The RBAC model has been used minimally in SELinux security, while the TE model has been the predominant focus. The TE model further associates a security type with every element in the system and manages an access control matrix based on the type of the subject that makes a request and the type of the target object that is being accessed (as well as the class and mode of the target object).

The current MLS model was recently developed by Trusted Computer Systems (TCS) [Han06]. It is largely orthogonal to the TE model meaning there is practically no interaction between the two. It associates an MLS level with every element in the system. On every security-sensitive operation, a set of MLS constraints is checked based on the MLS level of the subject and the object as well as the object class and mode of access. There is a standard reference MLS policy provided in the SELinux distribution which seeks to implement a confidentiality policy in accordance with the definitions by Bell and LaPadula [BL75]. A logical specification of the syntax and semantics is given in Section 6.3.
6.2.1 Example

The number and complexity of MLS constraints for a standard SELinux policy make manual analysis impractical. Here we present a motivating example of the difficulty, based in our own experience.

A brief study of the hundreds of lines of policy statements in the reference SELinux MLS policy gave the appearance that it might be possible to violate the standard MLS information flow goal preventing write-downs. One complication is that SELinux uses an expanded form of the standard MLS model, allowing a range of levels to be associated with a subject (this will be introduced more formally in Section 6.3.1), as in the DG/UX System [DG/96]. At first glance, the policy prevents a process from reading data out of a file at a high level and writing to a lower level. At the same time, it seemed that it might be possible simply to relabel a file and downgrade it using a process with a particular MLS range. Thus, unlimited downgrading would be possible for an unprivileged user.

Even a more thorough study of all the constraints applied to the file class did not reveal a counter-example. In this case, in order to disprove our hypothesis, we had to construct an experiment on an actual SELinux system and read the audit logs to determine our mistake. In our study, we had overlooked a different permission, the mlsvalidatetrans permission that is only minimally documented in the literature. Even discovering that was difficult, because the audit logs were vague about which constraint was violated.

6.3 SELinux MLS Model

In this section we develop a model that defines the meaning of a set of SELinux MLS statements. This is the first logical specification of SELinux MLS policy. It was developed based on informal descriptions of the SELinux MLS constraints given in books [MMC06], articles [Han06, DWS06] and online discussions with the developers as well as our own experimental exploration of the MLS system.

6.3.1 Extended Security Context

An SELinux security context in a system that enables the MLS extension implemented by TCS [Han06] adds a fourth field to the three fields, user, role and type (which are all used for the RBAC and TE models described in Section 6.2): an MLS range defined by a low and a high MLS level. Each level is composed of a sensitivity level and an optional set of category compartments. Sensitivity represents an MLS clearance (on subjects) or classification (on objects), while categories represent a set of non-hierarchical compartments to which the subject may have access.

The following example is an SELinux security context in a system with the MLS extension disabled, it includes user, role, and type:

```
staff_u:staff_r:staff_t
```

An MLS-enabled SELinux system contains one additional field:

```
staff_u:staff_r:staff_t:s0-s2:c0.c15
```

Most of the objects in the system have the same value for their low and high levels (they are single-level); there are some exceptions like multi-level directories.
the other hand, it is not unusual for subjects to have different low and high levels. The low level means the current security clearance and the high level represents the upper bound security clearance for the same subject. In the following example s0 is the low level and s15 is the high level, in addition s15 has access to compartments c0 through c15:

\[ s0\rightarrow s15 : c0 . c15 \]

### 6.3.2 MLS Model

Although an SELinux policy includes thousands of statements that define the Mandatory Access Control rules for a particular system (implementing the RBAC and TE models), the focus of this work is the behavior of an SELinux MLS policy. Therefore the input of our model is the set of MLS-specific statements: sensitivity, category, level, dominance, mlsconstrain and mlsvalidatetrans. All definitions given in this section and Sections 6.4 and 6.5 use the notation presented below.

- **s**: Security context for a given subject
- **o**: Security context for a given object
- **c**: Single class
- **p**: Single mode in which an object may be accessed
- **C**: Set of classes
- **P**: Set of modes in which an object may be accessed
- **u**: user
- **r**: role
- **t**: type
- **sl**: Sensitivity level
- **ca**: Category
- **exp**: Boolean expression
- **Policy**: Set of SELinux statements and rules (TE and MLS) that define a policy
- **stmt**: statement in an given policy

#### 6.3.2.1 Syntax

In this section we present a brief description of our set of MLS statements: sensitivity, category, level, dominance, mlsconstrain and mlsvalidatetrans. At the end of every paragraph we give the actual syntax used in SELinux.

**sensitivity** Sensitivities in an MLS model represent security clearance for subjects or security classification for objects. In SELinux the statement that declares a sensitivity level follows the syntax presented below. The set of sensitivity statements define the set of valid sensitivities in a particular SELinux system.

\[ sensitivity\ id\ [\ alias\ id\_set\ ];\]

**category** Categories expand an MLS model by making it possible to represent different families of data associated with each sensitivity. For example, categories allow us to make a distinction between Top Secret (sensitivity) Nuclear (category) data and
Top Secret Policital (another category) data.

category id [ alias id_set ];

level The MLS level commands define legal combinations of sensitivities and category sets.

level sl : [ ca_set ];

dominance MLS sensitivities are organized into a hierarchy; higher sensitivities represent higher security clearances or higher security classification. The first sensitivity in the dominance statement is assigned the lowest position in the hierarchy, the last element is assigned the highest position.

dominance { sl_1 sl_2 ... sl_n }

mlsconstrain This statement restricts access rights assigned in an SELinux policy, according to relationships between the security context of the subject that requests access and the security context of the target object, the class of the target object and the mode in which the subject wants to access the object. Objects are classified into classes (filesystem, file, dir,...); for each class a set of access modes is defined (read, write, create,...).

mlsconstrain C P exp;

mlsvalidatetrans This statement restricts the ability of a subject to change the security context of a target object, according to relationships among the new context, the old context and the security context of the subject that requests the change, and the class of the target object.

mlsvalidatetrans C exp;

The following set of statements defines a system with sensitivities s0 to s3, the lattice over those elements, the set of allowed categories and levels. Examples for mlsconstrain and mlsvalidatetrans are presented in the next section.

sensitivity s0;
sensitivity s1;
sensitivity s2;
sensitivity s3;
dominance { s0 s1 s2 s3 }
category c0; category c1; category c2;
level s0:c0.c2;
level s1:c0.c2;
level s2:c0.c2;
level s3:c0.c2;
6.3.2.2 Grammar

This section presents the Backus Naur Form of part of the SELinux policy language; the part that enables the definition of MLS statements. The grammar is given in Table 6.1.

6.3.2.3 Semantics

In this section we present the analytical model we developed to understand the meaning of a set of MLS statements. The following paragraphs present the components of the model. The semantics defined here foreshadows the Prolog implementation presented in Section 6.5. This is intentional for validating the correspondance between the Prolog and the formal definition.

This part of the section presents four operators to handle MLS statements: name, classes, modes and expr. The operator, name, returns the name of a given statement, classes returns the set of classes a statement applies to, modes returns the set of modes a statement applies to, and expr returns the boolean expression a statement is based on. Not all the operators are defined for all the MLS statements; classes and expr are defined only for mlsconstrain and mlsvalidatetrans, and modes is defined only for mlsconstrain. Below are some examples of the described operators.

\[
\begin{align*}
\text{name (sensitivity s1)} &= \text{sensitivity} \\
\text{name (category c0)} &= \text{category} \\
\text{name (level s1:c0,c1,c2)} &= \text{level} \\
\text{name (dominance \{ s1,s2,s3,s4 \})} &= \text{dominance} \\
\text{classes (mlsconstrain file \{ create relabelto \}}} \\
\quad (12 \text{ eq h2}) &= \{\text{file}\} \\
\text{modes (mlsconstrain file \{ create relabelto \}}} \\
\quad (12 \text{ eq h2}) &= \{\text{create relabelto}\} \\
\text{expr (mlsconstrain file \{ create relabelto \}}} \\
\quad (12 \text{ eq h2}) &= (12 \text{ eq h2})
\end{align*}
\]

The operators classes and modes also apply to a Policy. In that case they respectively return all the classes declared and all the modes in which objects may be accessed with respect to a given Policy.

The model also includes operators to get the components of a given security context: getu, getr, gett, getl and geth. They take a security context \((u,r,t,(l,h))\) where the
<table>
<thead>
<tr>
<th>mlspolicy</th>
<th>mlspolicy_decl</th>
</tr>
</thead>
<tbody>
<tr>
<td>mlspolicy_decl</td>
<td>mlsconstraint_def</td>
</tr>
<tr>
<td>mlsconstraint_def</td>
<td>mlsvalidatetrans_def</td>
</tr>
<tr>
<td>mlsvalidatetrans_def</td>
<td>MSLVALIDATETRANS names cexpr</td>
</tr>
<tr>
<td>sensitivity_def</td>
<td>SENSITIVITY identifier alias_def ';'</td>
</tr>
<tr>
<td>alias_def</td>
<td>ALIAS names</td>
</tr>
<tr>
<td>dominance</td>
<td>DOMINANCE identifier</td>
</tr>
<tr>
<td></td>
<td>DOMINANCE '{'identifier_list'}'</td>
</tr>
<tr>
<td>opt_categories</td>
<td>categories</td>
</tr>
<tr>
<td>categories</td>
<td>category_def</td>
</tr>
<tr>
<td></td>
<td>categories category_def</td>
</tr>
<tr>
<td>category_def</td>
<td>CATEGORY identifier alias_def ';'</td>
</tr>
<tr>
<td></td>
<td>CATEGORY identifier ';'</td>
</tr>
<tr>
<td>levels</td>
<td>level_def</td>
</tr>
<tr>
<td></td>
<td>levels level_def</td>
</tr>
<tr>
<td>level_def</td>
<td>LEVEL identifier ';'</td>
</tr>
<tr>
<td></td>
<td>id_comma_list ';'</td>
</tr>
<tr>
<td></td>
<td>LEVEL identifier ';'</td>
</tr>
<tr>
<td>mls</td>
<td>sensitivities dominance</td>
</tr>
<tr>
<td>sensitivities</td>
<td>sensitivity_def</td>
</tr>
<tr>
<td>sensitivities</td>
<td>sensitivities sensitivity_def</td>
</tr>
</tbody>
</table>

**Table 6.1.** BNF grammar for SELinux MLS constraint language.
pair, \((l,h)\) represent an MLS range and return the elements \(u, r, t, l,\) and \(h,\) respectively:

\[
\begin{align*}
\text{getu}((u, r, t, (l, h))) &= u \\
\text{getr}((u, r, t, (l, h))) &= r \\
\text{gett}((u, r, t, (l, h))) &= t \\
\text{getl}((u, r, t, (l, h))) &= l \\
\text{geth}((u, r, t, (l, h))) &= h
\end{align*}
\]

SELinux has a dominance rule that defines a partial order over the MLS sensitivities.

\[
\text{dominance}(s_{l_1}, s_{l_2}, ..., s_{l_n}) \equiv \text{induces a partial order, } \sqsubseteq \\
\text{over the elements } s_{l_1}, s_{l_2}, ..., s_{l_n}, \text{s.t.} \\
s_{l_1} \sqsubseteq s_{l_2} \sqsubseteq ... \sqsubseteq s_{l_n}.
\]

The model includes operators to get the components of a given MLS level: \text{getsens} and \text{getcat}. For example, they take an MLS level \(s_1:c_0,c_1\) and return the elements \(s_1\) and the set \(\{c_0,c_1\},\) respectively.

The model also includes operators to compare two MLS levels: \text{dom}, \text{domby} and \text{incomp} based on the partial order defined by the dominance statement for sensitivities and the set defined by the categories associated with each level.

\[
\begin{align*}
\text{opl}(==, l_1, l_2) &= (l_1 = l_2) \\
\text{opl}(!=, l_1, l_2) &= (l_1 \neq l_2) \\
\text{opl}(\text{dom}, l_1, l_2) &= (\text{getsens}(l_2) \sqsubseteq \text{getsens}(l_1)) \land \\
& \quad (\text{getcat}(l_2) \sqsubseteq \text{getcat}(l_1)) \\
\text{opl}(\text{domby}, l_1, l_2) &= (\text{getsens}(l_1) \sqsubseteq \text{getsens}(l_2)) \land \\
& \quad (\text{getcat}(l_1) \sqsubseteq \text{getcat}(l_2)) \\
\text{opl}(\text{incomp}, l_1, l_2) &= \neg(\text{opl}(\text{dom}, l_1, l_2)) \land \neg(\text{opl}(\text{domby}, l_1, l_2))
\end{align*}
\]

Dominance over roles is defined in a way that is analogous to the dominance over levels, thus the operators \text{dom}, \text{domby} and \text{incomp} also apply. Roles are not directly related to MLS policies but they are used to express some MLS statements. Because of that, the meaning of roles is included in our framework. Details are presented next.

\[
\text{dominance}(r_1, r_2, ..., r_n) \equiv \text{induces a partial order, } \sqsubseteq \\
\text{over the elements } r_1, r_2, ..., r_n, \text{s.t.} \\
r_1 \sqsubseteq r_2 \sqsubseteq ... \sqsubseteq r_n.
\]
We define an operator to generate the set of all valid ranges in a given Policy. Some subjects and multi-level objects require access to multiple MLS levels; SELinux makes this possible through MLS ranges, but not every range is allowed.

\[
\text{ranges}(\text{Policy}) = \{ (l_1, l_2) \mid (\text{getsens}(l_1) \subseteq \text{getsens}(l_2)) \land \\
(\text{getcat}(l_1) \subseteq \text{getcat}(l_2)) \}
\]

The definition of the previous operators is straight-forward. They serve primarily to support the main definition, which consists of the operators \( \gamma_{\text{MLS}} \) and \( \gamma_{\text{MLSvt}} \). These operators determine the result of applying all relevant constraints to a particular subject, object, object class and access mode. If the result of applying all relevant constraints (a possibly empty set) is true then \( \gamma_{\text{MLS}} \) is true, otherwise it is false.

\[
\gamma_{\text{MLS}}(s, o, c, p) = \{ \{ \text{stmt} \mid \text{stmt} \in \text{Policy}, \\
\text{name}(\text{stmt}) = \text{mlsconstrain}, \\
c \in \text{classes}(\text{stmt}), p \in \text{modes}(\text{stmt}), \\
\| \text{expr}(\text{stmt}) \|_{s,o} =\text{FALSE} \} = \emptyset \}
\]

Next we present an inductive definition for the semantics of

\[
\| \text{expr}(\text{stmt}) \|_{s,o}
\]
s represents the subject that is requesting the operation that initiates the check of the constraint; o is the object that s attempts to access.

\[
\begin{align*}
\| \text{not}(exp) \|_{s,o} &= \neg (\| exp \|_{s,o}) \\
\| exp_a \text{ and } exp_b \|_{s,o} &= \| exp_a \|_{s,o} \land \| exp_b \|_{s,o} \\
\| exp_a \text{ or } exp_b \|_{s,o} &= \| exp_a \|_{s,o} \lor \| exp_b \|_{s,o} \\
\| u1 == u2 \|_{s,o} &= (\text{getu}(s) = \text{getu}(o)) \\
\| u1 != u2 \|_{s,o} &= (\text{getu}(s) \neq \text{getu}(o)) \\
\| r1 \text{ operator r2} \|_{s,o} &= \text{opr}(\text{operator}, \text{getr}(s), \text{getr}(o)) \\
\| t1 == t2 \|_{s,o} &= (\text{gett}(s) = \text{gett}(o)) \\
\| t1 != t2 \|_{s,o} &= (\text{gett}(s) \neq \text{gett}(o)) \\
\| l1 \text{ operator l2} \|_{s,o} &= \text{opl}(\text{operator}, \text{getl}(s), \text{getl}(o)) \\
\| h1 \text{ operator h2} \|_{s,o} &= \text{opl}(\text{operator}, \text{geth}(s), \text{geth}(o)) \\
\| l1 \text{ operator h1} \|_{s,o} &= \text{opl}(\text{operator}, \text{getl}(s), \text{geth}(s)) \\
\| l2 \text{ operator h2} \|_{s,o} &= \text{opl}(\text{operator}, \text{getl}(o), \text{geth}(o))
\end{align*}
\]

In addition, the values of the fields user, role and type from a subject’s security context or object’s security context may be tested against predefined values:

\[
\begin{align*}
\| u1 == userset \|_{s,o} &= (\text{getu}(s) \in \text{userset}) \\
\| u1 != userset \|_{s,o} &= (\text{getu}(s) \notin \text{userset})
\end{align*}
\]

The the operators getr and gett perform analogous operations over u2 (object’s user), r1 and t1 (subject’s role and type) and r2 and t2 (object’s role and type).
The following example shows the behavior of $\gamma_{MLS}(s, o, c, p)$ in a given case. A user with MLS range $s_1-s_2$ has a file with MLS level $s_1$, the user tries to upgrade his file to $s_2$.

Two of the permissions that must be checked in the default MLS policy are relabelto and relabelfrom, therefore the following mlsconstrain rules are checked:

**First** The following MLS constraint is checked:

\[
\text{mlsconstrain } \{ \text{file lnk\_file fifo\_file} \} \\
\{ \text{create relabelto} \} \\
( l2 \text{ eq h2 } ) ; \\
\text{mlsconstrain} \{ \text{dir file lnk\_file chr\_file blk\_file} \} \\
\text{relabelto} \\
( h1 \text{ dom h2 } ) ; \\
\]

The evaluation of these constraints gives:

\[
\gamma_{MLS}( \text{staff\_u:staff\_r:staff\_t:s1-s2:c0,c2,} \\
\text{staff\_u:object\_r:user\_home\_dir\_t:s2,} \\
\text{file, relabelto} ) = TRUE \\
\]

**Second** The following MLS is checked:

\[
\text{mlsconstrain } \{ \text{file lnk\_file fifo\_file} \} \\
\{ \text{write create setattr relabelfrom rename} \} \\
( ( l1 \text{ eq l2 } ) \text{ or } ( ( t1 = \text{mlsfilewritetoclr} ) \text{ and ( h1 dom l2 )} \\
\text{and ( l1 domby l2 )} ) \text{ or } \\
( t1 = \text{mlsfilewrite} ) \text{ or } \\
( t2 = \text{mlstrustedobject} ) ) ; \\
\]

\[
\| u2 == set \|_{s, o} = (\text{getu}(o) \in \text{userset}) \\
\| u2 \neq set \|_{s, o} = (\text{getu}(o) \notin \text{userset}) \\
\| r1 == roleset \|_{s, o} = (\text{getr}(s) \in \text{roleset}) \\
\| r2 == roleset \|_{s, o} = (\text{getr}(o) \in \text{roleset}) \\
\| r1 \neq roleset \|_{s, o} = (\text{getr}(s) \notin \text{roleset}) \\
\| r2 \neq roleset \|_{s, o} = (\text{getr}(o) \notin \text{roleset}) \\
\| t1 == typeset \|_{s, o} = (\text{gett}(s) \in \text{typeset}) \\
\| t2 == typeset \|_{s, o} = (\text{gett}(o) \in \text{typeset}) \\
\| t1 \neq typeset \|_{s, o} = (\text{gett}(s) \notin \text{typeset}) \\
\| t2 \neq typeset \|_{s, o} = (\text{gett}(o) \notin \text{typeset}) \\
\]
The evaluation of this constraint gives:

\[ \gamma_{MLS}(staff_u:staff_r:staff_t:s1-s2:c0.c2, \\
    staff_u:object_r:user_home_dir_t:s1, \\
    file,relabelfrom) = FALSE \]

The semantics of the last MLS operator we define is for handling dynamic transitions of objects from one MLS level to another. \( \gamma_{\text{MLS}v_t} \) detects the result of the constraints that apply in this particular transition case.

\[ \gamma_{\text{MLS}v_t}(o1,o2,s,c) = (\{stmt | stmt \in Policy, \\
    name(stmt) = \text{mlsvalidatetrans}, \\
    c \in \text{classes}(stmt), \\
    \parallel expr(stmt) \parallel_{o1,o2,s} = FALSE\} = \emptyset) \]

Next we present an inductive definition for the semantics of

\[ \parallel expr(stmt) \parallel_{o1,o2,s} \]

for \text{mlsvalidatetrans}. These definitions look similar to the ones presented for \text{mlsconstrain} but notice that now we have three elements to evaluate instead of two: \( o1 \): old security context, \( o2 \): new security context and \( s \): security context of the process that requests the transition. In the boolean expression, elements indexed with 1 \((u1,r1,t1)\) make reference to \( o1 \), elements indexed with 2 \((u2,r2,t2)\) make reference to \( o2 \) and elements indexed with 3 \((u3,r3,t3)\) make reference to \( s \).

\[ \parallel \text{not}(exp) \parallel_{o1,o2,s} = \neg (\parallel exp \parallel_{o1,o2,s}) \]
\[ \parallel exp_a \text{ and } exp_b \parallel_{o1,o2,s} = (\parallel exp_a \parallel_{o1,o2,s} \land \parallel exp_b \parallel_{o1,o2,s}) \]
\[ \parallel exp_a \text{ or } exp_b \parallel_{o1,o2,s} = (\parallel exp_a \parallel_{o1,o2,s} \lor \parallel exp_b \parallel_{o1,o2,s}) \]
Next we define the meaning of boolean expressions for $\text{mlsvalidatetrans}$. 

\[
\begin{align*}
||u_1 == u_2||_{o_1,o_2,s} &= (\text{getu}(o_1) = \text{getu}(o_2)) \\
||u_1 == u_2||_{o_1,o_2,s} &= (\text{getu}(o_1) = \text{getu}(o_2)) \\
||u_1 != u_2||_{o_1,o_2,s} &= (\text{getu}(o_1) \neq \text{getu}(o_2)) \\
||r_1 \text{ operator } r_2||_{o_1,o_2,s} &= \text{opr}(\text{getr}(o_1), \text{getr}(o_2)) \\
||t_1 == t_2||_{o_1,o_2,s} &= (\text{gett}(o_1) = \text{gett}(o_2)) \\
||t_1 != t_2||_{o_1,o_2,s} &= (\text{gett}(o_1) \neq \text{gett}(o_2)) \\
||l_1 \text{ operator } l_2||_{o_1,o_2,s} &= \text{opl}(\text{getl}(o_1), \text{getl}(o_2)) \\
||l_1 \text{ operator } h_2||_{o_1,o_2,s} &= \text{opl}(\text{getl}(o_1), \text{geth}(o_2)) \\
||h_1 \text{ operator } l_2||_{o_1,o_2,s} &= \text{opl}(\text{geth}(o_1), \text{getl}(o_2)) \\
||h_1 \text{ operator } h_2||_{o_1,o_2,s} &= \text{opl}(\text{geth}(o_1), \text{geth}(o_2)) \\
||l_1 \text{ operator } h_1||_{o_1,o_2,s} &= \text{opl}(\text{getl}(o_1), \text{getl}(o_1)) \\
||l_2 \text{ operator } h_2||_{o_1,o_2,s} &= \text{opl}(\text{getl}(o_2), \text{geth}(o_2))
\end{align*}
\]

Notice that, as previously indicated, elements indexed with 1 are linked to $o_1$ and elements indexed with 2 are linked to $o_2$.

In the same way as $\text{mlsconstrain}$, the values of the fields user, role and type from the involved security contexts may be tested against predefined values:

\[
\begin{align*}
||u_1 == \text{userset}||_{o_1,o_2,s} &= (\text{getu}(o_1) \in \text{userset}) \\
||u_1 == \text{userset}||_{o_1,o_2,s} &= (\text{getu}(o_1) \in \text{userset}) \\
||u_2 == \text{userset}||_{o_1,o_2,s} &= (\text{getu}(o_2) \in \text{userset}) \\
||u_1 != \text{userset}||_{o_1,o_2,s} &= (\text{getu}(o_1) \notin \text{userset}) \\
||u_2 != \text{userset}||_{o_1,o_2,s} &= (\text{getu}(o_2) \notin \text{userset}) \\
||r_1 == \text{roleset}||_{o_1,o_2,s} &= (\text{getr}(o_1) \in \text{roleset}) \\
||r_2 == \text{roleset}||_{o_1,o_2,s} &= (\text{getr}(o_2) \in \text{roleset}) \\
||r_1 != \text{roleset}||_{o_1,o_2,s} &= (\text{getr}(o_1) \notin \text{roleset}) \\
||r_2 != \text{roleset}||_{o_1,o_2,s} &= (\text{getr}(o_2) \notin \text{roleset}) \\
||t_1 == \text{typeset}||_{o_1,o_2,s} &= (\text{gett}(o_1) \in \text{typeset}) \\
||t_2 == \text{typeset}||_{o_1,o_2,s} &= (\text{gett}(o_2) \in \text{typeset}) \\
||t_1 != \text{typeset}||_{o_1,o_2,s} &= (\text{gett}(o_1) \notin \text{typeset}) \\
||t_2 != \text{typeset}||_{o_1,o_2,s} &= (\text{gett}(o_2) \notin \text{typeset})
\end{align*}
\]

Since $\text{mlsvalidatetrans}$ involves a third security context, there are additional operators to handle it. The following paragraph presents the ways in which this security context
may be tested:

\[ u_3 = \text{userset} \parallel o_{1,2,s} = (getu(s) \in \text{userset}) \]
\[ r_3 = \text{roleset} \parallel o_{1,2,s} = (getr(s) \in \text{roleset}) \]
\[ t_3 = \text{typeset} \parallel o_{1,2,s} = (gett(s) \in \text{typeset}) \]
\[ u_3 \neq \text{userset} \parallel o_{1,2,s} = (getu(s) \notin \text{userset}) \]
\[ r_3 \neq \text{roleset} \parallel o_{1,2,s} = (getr(s) \notin \text{roleset}) \]
\[ t_3 \neq \text{typeset} \parallel o_{1,2,s} = (gett(s) \notin \text{typeset}) \]

Taking the same example as before: a user with MLS range s1-s2 has a file with MLS level s1, the user tries to upgrade his file to s2, we show the result of \( \gamma_{MLSvt}(o_1, o_2, s, c) \). In the current policy there is only one mlsvalidatetrans statement.

# the file upgrade downgrade rule
mlsvalidatetrans
dir file lnk_file chr_file blk_file sock_file fifo_file
((( l1 eq l2 ) or
(( t3 == mlsfileupgrade ) and ( l1 domby l2 )) or
(( t3 == mlsfile downgrade ) and ( l1 dom l2 )) or
(( t3 == mlsfile downgrade ) and ( l1 incomp l2 ))) and
(( h1 eq h2 ) or
(( t3 == mlsfileupgrade ) and ( h1 domby h2 )) or
(( t3 == mlsfile downgrade ) and ( h1 dom h2 )) or
(( t3 == mlsfile downgrade ) and ( h1 incomp h2 ))));

This is the result:

\[ \gamma_{MLSvt}( \text{staff\_u:object\_r:user\_home\_dir\_t:s1,}
staff\_u:object\_r:user\_home\_dir\_t:s2,
staff\_u:staff\_r:staff\_t:s1-s2:c0.c2,
file) = FALSE \]

The analytical model described in this section offers a logical framework for analyzing MLS policies. This is a contribution in itself, because no logical specification of the semantics is given elsewhere. Users of SELinux MLS policies are beholden to informal descriptions of the constraint system or experimental explorations of its behavior in various cases. Consequently, there is also no way to prove the soundness or completeness of this system, because no formal model exists as a standard against which to compare it.

Our goal is not just to provide a logical specification, however, but also to use this specification for analyzing real policies. This cannot reasonably be accomplished by hand. In the next section we give some policy analysis algorithms that use these specifications and implement these algorithms in PALMS, described in Section 6.5.
6.4 Analysis

Understanding the semantics of the SELinux MLS policy is useful for various purposes. For example, it is important, for a given policy, to be able to determine whether all data classes and modes are constrained by the policy. Determining whether the policy faithfully implements basic information flow goals such as the simple security condition and \( \star \)-property is also important. There are also some practical systems reasons for analyzing the information flow properties of a given policy. In distributed systems, a system service may need to determine whether two MLS policies are compliant [JMS+06]. In cases that a MAC-based OS needs to trust an application to handle multiple levels of data, it is important that the OS can determine whether the application’s information flow policy complies with its own.

Policy compliance is important in a distributed system when labels are being communicated over sockets and an SELinux machine wants to be certain that the machine to which it is sending its data will be compliant with its own policy. For example, when machine A connects to machine B over a socket with MLS label \( s_2 \), will machine B honor the policy of machine A and not leak data passed through that socket to a lower level, such as \( s_1 \)?

Another application of this analysis could be for applications running in a particular OS. In some cases, it is necessary for an application to handle multiple levels of data inputs and outputs. If the application’s flows obey a particular security lattice, can those flows be tested for compliance against the host OS’s MLS policy?

Throughout this section, we refer to the SELinux MLS reference policy, version 20 using \texttt{mls-strict}. This is the policy that is distributed with latest version of SELinux. That MLS policy contains about 350 lines of policy statements ranging over 40 different kernel object classes, which can be accessed in 50 different modes. Thus, it is not feasible to evaluate by hand the functions we give in this section. For this reason, we have implemented these functions in an analyzer presented in the next section.

In this section, we use the formal semantics defined in Section 6.3 to demonstrate how we can determine compliance of one policy with another policy. We give a formal presentation here, which we have implemented in Prolog. This section serves as both a formal description and also, because the Prolog code follows the formalism so closely, as an introduction to the implementation. First we give some general definitions of information flows and functions that operate on them, and then we give some algorithms for how we instantiate these functions for SELinux MLS policy.

6.4.1 Finding all information flows

Definition 6.4.1 (Information Flow Policy). A policy consists of a set of security levels arranged in a lattice with partial order \( \subseteq \) and a set of statements determining each subject’s read/write permissions for a given object based on the security levels of the subject and object (and possibly also on other factors such as the class of the object).

Consider a typical military MLS information flow policy with no categories. In such a policy there are four security levels. Typically, military policies have permissions which implement the simple security condition (\( ssc \)) and \( \star \)-property:
Example 6.4.2 (Military MLS policy).

\[
\text{levels}(\text{Mil}) = \{\text{unclassified}(UC), \text{confidential}(CO), \\
\text{secret}(S), \text{topsecret}(TS)\}
\]

where \( UC \subseteq CO \subseteq S \subseteq TS \) and reads and writes obey the following properties:

**Simple security condition:** For a subject labeled \( l_s \) and an object labeled \( l_o \), the subject can read from the object iff \( l_o \sqsubseteq l_s \).

**\( \star \)-property:** For a subject labeled \( l_s \) and an object labeled \( l_o \), the subject can write to the object iff \( l_s \sqsubseteq l_o \).

We define an information flow in the following way:

**Definition 6.4.3 (Information flow).** An information flow from \( l_1 \) to \( l_2 \) exists in a system when a single process can read from a resource labeled with \( l_1 \) and write to a resource labeled with \( l_2 \).

**Example 6.4.4.** For the military policy given in Example 6.4.2, there is an information flow \((UC,S)\), because for a subject at level \( CO \), there is a valid read of an object at level \( UC \) and a valid write of that object out to \( S \). (Note: there are also other ways to generate this information flow, with a subject at level \( UC \) or \( CO \), but not at \( TS \).)

Next we define a function that is important for proving compliance, \( \text{AllFlows} \).

Here we give only a generic definition of what this function should do. Later, we will instantiate it for the Mil policy and the SELinux policy.

**Definition 6.4.5 (AllFlows).** The function

\[
\text{AllFlows} : \text{Policy} \rightarrow \wp(\text{levels}(\text{Policy}) \times \text{levels}(\text{Policy}))
\]

returns all information flows allowed in a given \( \text{Policy} \) with levels, \( \text{levels}(\text{Policy}) \).

To instantiate this function for the Mil policy, we must find all information flows, such that the \( \text{ssc} \) and the \( \star \)-property are preserved.

**Example 6.4.6 (AllFlows\text{\_Mil}).**

\[
\text{AllFlows\text{\_Mil}} = \{(l_1, l_2) : l_1, l_2 \in \text{levels}(\text{Mil}) \land \exists l_s \in \text{levels}(\text{Mil}). l_1 \sqsubseteq l_s \sqsubseteq l_2
\]

which would give the set

\[
\{(UC,UC), (UC,CO), (UC,S), (UC,TS), (CO,CO), (CO,S), (CO,TS), \\
(S,S), (S,TS), (TS,TS)\}
\]

6.4.2 Comparing policies

In addition to determining the information flows that are allowed by a given policy, it can also be useful to compare MLS policies. In a distributed system, for example, it is important to know how the policies of two operating systems compare, before they start exchanging labeled data.
When comparing two information flow policies, we require a mapping from the levels in one policy to the levels in the other. The mapping need not be defined for every level, but must map the levels in policy A to a subset of the levels in policy B. All levels which are not shared between policy A and policy B are mapped to ⊥ (undefined). In the following, we define both the renaming of a single level and the renaming of a flow (overloading the name rename).

**Definition 6.4.7 (rename).** Renaming levels and flows

\[
\begin{align*}
\text{rename}_{A \rightarrow B} : & \text{ levels}(A) \rightarrow (\text{levels}(B) + \bot) \\
\text{rename}_{A \rightarrow B} : & \text{ levels}(A) \times \text{levels}(A) \rightarrow \\
& (\text{levels}(B) + \bot) \times (\text{levels}(B) + \bot)
\end{align*}
\]

**Definition 6.4.8 (Shared levels).** A level l is said to be shared between two policies A and B iff \( \text{rename}_{A \rightarrow B}(l) \neq \bot \)

Compliance can then be defined for two policies by comparing the flows allowed in one policy with the flows allowed in the other. Specifically, we are interested the flows between levels shared by the two policies.

**Definition 6.4.9 (Compliance).** An information flow policy A is said to be compliant with an information flow policy B, iff

\[
\text{Flows}^\prime_A \subseteq \text{Flows}_B
\]

where

\[
\begin{align*}
\text{Flows}_A &= \text{AllFlows}_A(A) \\
\text{Flows}_B &= \text{AllFlows}_B(B) \\
\text{Flows}^\prime_A &= \text{rename}_{A \rightarrow B}(\text{Flows}_A)
\end{align*}
\]

Although the definition of compliance implies that all flows in both policies should be determined, in order to determine whether the flows in policy A are a subset of policy B, only the flows of policy A need to be exhaustively determined. Then each flow allowed by A can be checked to see if it is also allowed in policy B. This can lead to some performance improvement if policy B is significantly larger than policy A (as in the case when B is an OS policy and A is only an application policy).

### 6.4.3 Information flows for SELinux MLS

When implementing these information flow functions for SELinux policy, we must make some adjustments. The first consideration is that SELinux policy parameterizes MLS access rules based on object classes \((c)\), as described in Section 6.3. Thus, an information flow can occur using multiple classes, such as by reading from a public file and then writing to a secret \(\text{ipc}\). This requires us to define information flows by iterating over all possible object classes.
The second consideration is that the policy also parameterizes accesses based on the possible modes for that class. So, continuing the previous example, information could be read from a public file using the `getattribute` mode and written to a secret `ipc` using the `open` mode. We follow other systems [GHR05, SSS04] in grouping modes into “read-like” and “write-like” modes. Some modes fall into both categories, such as `dir create` which certainly is “write-like”, but is also “read-like” because it will reveal whether the directory already existed. We extend our formal semantics to include the functions, `readlike(p)` and `writelike(p)` which return true if the mode `p` is read-like or write-like, respectively.

The algorithm `AllFlows` can be instantiated for SELinux MLS policy by using the constraint $\gamma_{MLS}$ and the accessors, `classes`, `modes`, `ranges` from our formal semantics given in Section 6.3. The function is divided into two checks corresponding to two different ways that information flows can occur. The first way is by reading (in some mode) from some class at one level and writing (in some mode) to some class at another level. The second way is by simply relabeling an object from one level to another level.

Although we are not primarily concerned about general security contexts (including user, role and type) for our analysis of the MLS policy, $\gamma_{MLS}$ does require that the full security context of the subject and object be provided. This is because, generally speaking, the subject might have some special privileges that affect the MLS constraints. For this analysis, we are concerned with the most basic scenario and so we fix our subject and object to have a vanilla type $t$ with no extra privileges and to have insignificant user and role fields. For a more thorough analysis, our MLS analysis could be combined with existing analyses [Tec, JEZ03, SSS04, GHR05] that consider information flows introduced by type enforcement. The orthogonality of TE policies from MLS policies, however, facilitates the approach we have taken. The only additional interaction that could be considered is when a type transition might move the subject into a state in which it has some additional MLS privileges. We leave the consideration of this fringe case to future work. Thus, the set of flows can be found by unioning these two sets together as follows,

**Algorithm 6.4.10** ($\text{AllFlows}_{\text{SELinux}}$).

$$\text{AllFlows}_{\text{SELinux}}(\text{Policy}) = \{(l_1, l_2): \exists c_1, c_2 \in \text{classes}(\text{Policy}).\exists p_1, p_2 \in \text{modes}(\text{Policy}).\exists l_s \in \text{ranges}(\text{Policy}).\text{readlike}(p_1) \land \text{writelike}(p_2) \land$$
$$s = (u, r, t, l_s) \land o_1 = (\text{sys}, \text{obj}, t, l_1) \land o_2 = (\text{sys}, \text{obj}, t, l_2) \land$$
$$\gamma_{MLS}(s, o_1, c_1, p_1) \land \gamma_{MLS}(s, o_2, c_2, p_2)\}$$
\[ \bigcup \{(l_1, l_2) : \exists c \in \text{classes}(Policy). \exists l_s \in \text{ranges}(Policy). s = (u, r, t, l_s) \land o_1 = (sys, obj, t, l_1) \land o_2 = (sys, obj, t, l_2) \land \gamma_{\text{MLS}}(s, o_1, c, \text{relabelfrom}) \land \gamma_{\text{MLS}}(s, o_2, c, \text{relabelto}) \land \gamma_{\text{MLSvt}}(o_1, o_2, s, c) \} \]

In the next section, we describe the Prolog code that implements these functions and give an example of determining whether the SELinux reference MLS policy meets the \text{scc} and \text{*}-property, by determining if it is compliant with the Military MLS policy we have described throughout this section.

### 6.5 Implementation

We implemented an analysis framework based on the analytical model presented in the previous section. This framework allows us to evaluate the MLS properties for a real SELinux policy. We implemented this framework by encoding the logic into Prolog, using the XSB Prolog implementation. Although the tabled resolution provided by XSB was not essential, it does serve to improve performance. Using Prolog was beneficial for multiple reasons. One is that the program encoding is a direct analogue to the logical model presented in Sections 6.3 and 6.4, making it trivial to determine the correctness of the implementation. Another is the simplicity of the Prolog code. Prolog is ideal for implementing search algorithms, because backtracking and unification are inherent to the language. Thus, merely expressing the rulebase for the SELinux policy along with some simple description of the searches is enough to implement the analysis. Only 20 lines of code are required to implement the functions described in Section 6.4 (the code for implementing the semantics in Section 6.3 is longer, about 150 lines, but need not be changed to vary the queries). Thus, it is easy to make slight modifications to the code to check different properties of the policy. Finally, because the analyzer should only be run infrequently, time is not a limiting factor (although, in fact, XSB Prolog is highly optimized and the time is not prohibitive for the kinds of queries discussed in Section 6.4).

The implementation of the MLS semantics in Section 6.3 can be implemented in Prolog in a straightforward way. By way of background, variables in Prolog which begin with capital letters denote \textit{logic variables}. These variables are gradually instantiated through unification as Prolog processes a query. For cases in which the variable could be instantiated in different ways, Prolog inserts a backtracking point and tries all possibilities. In this way, for example, we can implement the \textit{ranges} function from Section 6.3 by using the predicate \texttt{valid_mls}. The predicate \texttt{valid_mls(L)} is true when \( L \) is bound to any valid MLS range.

We encode MLS labels as a 4-tuple containing the low sensitivity level and low category set followed by the high sensitivity level and the high category set. Thus, to denote the label \( s_0-s_3: c_0 . c_1 \) we write the following:
To expand this into a full security context, we use the functor \( \text{sc} \), giving,

\[
\text{sc}(\text{system_u}, \text{object_r}, \text{user_t}, \text{mls}(s0, [], s3, [c0, c1]))
\]

This particular example describes an object labeled with the type \( \text{user_t} \) and the MLS label given above.

The \textit{AllFlows} function follows the definition in Section 6.4, with the slight modification that it calls an auxiliary predicate \textit{hasFlows} to find a single flow and depends on the higher-order relation, \textit{findall} to find all possible flows. The code is given in Figure 6.1. It demonstrates the close correspondence of the implementation to both the semantics in Section 6.3.2.3 and the algorithms in Section 6.4.

### 6.5.1 Example

One useful application of our analyzer is for automatically determining compliance between an SELinux policy and another information flow policy. We give an example here which shows that the current reference policy for MLS complies with the \( \ast \)-property and simple security property. We do this by limiting the SELinux policy slightly and showing it complies with the military MLS policy given in Example 6.4.2\(^1\). Since this military policy is defined according to the \( \text{ssc} \) and \( \ast \)-property, if the SELinux policy is compliant with it, we have, by implication, that it is compliant with these properties.

For our analysis, we use all the constraint rules from the reference policy, but for clarity of presentation, we modify the available levels slightly. While the reference policy has 16 sensitivity levels, we reduce this to the four military levels. Also, for simplicity of presentation, we ignore category sets (note that our analyzer handles both of these correctly). A more important consideration is that the security properties we are interested in verifying do not consider MLS ranges. We can still carry out the compliance check if we limit the analyzer to check only single levels.

To summarize, we use the following renaming predicate

\[
\text{rename}(s0, \text{UC}). \\
\text{rename}(s1, \text{CO}). \\
\text{rename}(s2, \text{S}). \\
\text{rename}(s3, \text{TS}).
\]

Finally, we can run \textit{all_flows} to get all possible flows in the SELinux policy, as shown in the following sample XSB execution.

\[
?- \text{all_flows(}L\text{Set}). \\
L\text{Set}=[(s2, s3), (s1, s3), (s0, s3), (s1, s2), (s0, s2), \\
(s0, s1), (s3, s3), (s2, s2), (s1, s1), (s0, s0)]
\]

\(^1\)This limitation is only for demonstration purposes. Using all 16 sensitivity levels and all category sets only increases the analysis time, not the fundamental result.
Listing 6.1. The Prolog code for finding all information flows in a given SELinux policy.

```
all_flows(LSet) :-
    findall(L,
        (L=(L1,L2), has_flow(L1,L2)),
        LList),
    list_to_set(LList,LSet).

has_flow(L1,L2) :-
    valid_mls(LS),
    security_class(C1),read_like(C1,P1),
    S = sc(user_u,user_r,user_t,LS),
    O1 = sc(system_u,object_r,user_t,L1),
    O2 = sc(system_u,object_r,user_t,L2),
    gamma_mls(S,O1,C1,P1,true),
    security_class(C2),write_like(C2,P2),
    gamma_mls(S,O2,C2,P2,true).

has_flow(L1,L2) :-
    security_class(C),
    valid_mls(LS),
    S = sc(user_u,user_r,user_t,LS),
    O1 = sc(system_u,object_r,user_t,L1),
    O2 = sc(system_u,object_r,user_t,L2),
    gamma_mls(S,O1,C,relabelfrom,true),
    gamma_mls(S,O2,C,relabelltoto,true),
    gamma_mlsvt(O1,O2,S,C,true).
```
After renaming the flows given in $\text{LSet}$ and reordering them, we can see that the set is equal to $\text{AllFlows}_{\text{Mid}}$ in Example 6.4.6.

### 6.5.2 Applications

In building the analyzer, we found it useful for analyzing SELinux policy in other ways as well. As one example, it is not easy to tell by inspection that the constraint rules for the MLS policy cover all possible object classes and access modes, and since the policy specifies a default-allow, this is an especially critical property. In fact, as we ran our analyzer, we discovered some strange flows (from top secret to unclassified, for example) allowed by the policy. Isolating these flows, we re-ran the analyzer to recover how these flows took place and discovered they were enabled through such write-channels as `socket/open` and `process/sigchld`. Upon closer inspection, we discovered in comments that the makers of the SELinux policy intended for these permissions to be ignored. Further inspection revealed that they are coupled with `write` permissions which are not left unconstrained. Had there been other classes/modes left unconstrained, however, our analyzer would have caught them.

Another important application for PALMS is in determining compliance of an STL application with SELinux, as described in more detail in Section 5.4.3. In fact, such an analysis is necessary for making STLs portable to different environments, particularly MAC OSs that support MLS. The ultimate goal is to be able to show that the OS and the application together enforce a high-level security policy, i.e. to bridge the semantic gap in security policy and show that high-level security policy semantics are implemented in the operational semantics of systems. Logical specification of policy semantics and analysis of real policies play an essential role in this process.

More policy analyses are possible for showing how given policy systems enforce high-level security goals. We have put forth a method in this chapter that can provide a blueprint for developing future analyses. The logical specification of the MLS model in SELinux constitutes a re-usable contribution that can serve as the foundation for future analyses. The analysis tool, PALMS, provides a starting point for implementing future analyses.
Chapter 7

Policy-Driven Runtime System Infrastructure

As noted in the introduction, the two policy decision points in an STL application are in labeling inputs and outputs at application boundaries and in re-labeling data as it passes through the application through delegation and declassification. The STL compiler then ensures that all intervening code, including assignments, conditionals, method calls, exceptions, etc. will enforce the policy implemented at those points. In Chapters 3 and 4, we focus primarily on the re-labeling that must take place within applications, introducing our JPolicy infrastructure. In Chapters 5 and 6, we solve some of the systems-interface challenges in porting applications to diverse environments such as MAC OSs.

In both the PKI setting and the MAC setting, we made changes to the Jif Runtime in order to accommodate different environments (contrast the PKI-based environment in Chapter 3 with the MAC-based environment in Chapter 5). A critical element needed for implementing security policy using STLs has been neglected, however. Thus, we have implicitly recognized, but not explicitly handled the infrastructure needed for this critical policy decision point. The machinery necessary to re-instrument applications to work with a PKI vs. MAC, e.g., have been developed in largely ad hoc ways. To handle this more effectively, in a way that will improve the portability, expressiveness and separability of STL policy, a principled approach to customizing runtime systems is needed.

The runtime system is responsible for mediating all communications between the information-flow world within the application and the non-information flow world outside the application. Without runtime infrastructure to handle dynamic labeling of I/O, STLs can only be effective for analyzing closed systems (with no inputs or outputs).

Real programs are seldom closed systems, however. They must interact with a world outside themselves, receiving data from and sending data out on various channels. An email client must interact with a remote mail server as well as a local user, various files and databases. Another example is a network firewall that must interact with the networking subsystem and may also save audit logs to files. In a web browser, data is read from and stored into various system resources, including files and databases, interactions with applets, various user inputs, and of course the communications with various web servers.
STLs can perform a critical service in provably ensuring noninterference within an application. They cannot, however, automatically reason about the security of data as it crosses the application boundary to and from the system. A runtime system is needed to handle these security decisions, governing whether the data arriving on a socket should be labeled secret or public or \texttt{alice-data} or \texttt{bob-data} or determining whether secret data may safely be output to a particular file. In fact, these are critical decisions; the rest of the label checking is moot if this step is not handled correctly.

This has been recognized, de facto, in the construction of existing STL applications [AS05, Tse07, CVM07], as well as our own (presented in Chapters 3 and 5). Each implemented its own runtime infrastructure to cover its own particular policy for labeling I/O. It is notable that these runtime infrastructures could not be reused from one application to the next. This is because the labeling of I/O can be quite subtle, dependent on several factors, such as what system resources are used by the application, what system security mechanisms are available for these resources and what authentication protocols are used to identify the resources along with the data and policy they carry. In other words, runtime system infrastructure must be specialized for different environments and applications.

This specialization must not be ad hoc, however. Because correctly handling the labeling of I/O is critical for enforcing security policy, it is necessary to adopt a principled approach for developing customized runtime infrastructure. Furthermore, to maintain the systems principle of separating policy specification from implementation, the security decisions implemented through the runtime system must be configured through high-level policy.

In this chapter, we give principles to guide the development of application-specific runtime system infrastructure for STL applications. Our principles provide for modularized, specialized runtime systems that can be configured and controlled through high-level policy. As a manifestation of these principles, we provide a Channel abstraction for Jif. The Channel facilitates the mapping of policy onto I/O channels and thus serves as the basic building block of a runtime system. It can be implemented and extended in different settings according to the individualized needs of different applications. We also provide a high-level policy infrastructure for activating and configuring Channels.

The Channel was designed to accommodate a wide range of challenges we faced in developing STL applications throughout this work. Some key aspects of the Channel base class are that it provides a way of coupling input streams with output streams; it facilitates authentication protocols through maintaining state in the Channel itself; it can query an external mechanism for handling data-specific labeling; and it provides an API for determining what protections are offered by the output channel. We provide hooks to control and configure an application’s channels through our high-level policy infrastructure. We call this runtime infrastructure, both Channels and policy system, the SPolicy infrastructure.

We evaluate our approach by using our SPolicy tools to build FlowWall, a basic, packet-filtering network firewall [CBR02]. As part of FlowWall, we construct a new Channel, a PacketChannel, which handles the inputting and outputting of all network packets. The PacketChannel is constructed in such a way that in order for a packet to pass through the FlowWall, it must flow from its source to its destination address.
The Jif compiler provably ensures that all flows obey the label semantics defined through our JPolicy tools (discussed in Chapter 4). By automatically compiling this application flow policy such that it is isomorphic to the firewall policy, we are able to implement a firewall that provably enforces its firewall policy.

The result of our investigation shows that the Channel is effective for implementing high-level security policy in the runtime infrastructure. The Channel integrates well with STLs because it can be incorporated into a separable STL policy infrastructure; it can be configured and controlled apart from the application code. It also facilitates expressiveness and portability, because it utilizes a pluggable interface for various protocol-based, authentication-based and other data-specific labeling mechanisms. Finally, it aids STL software engineering, because it provides an intuitive interface for handling a problem that repeatedly faces STL programmers.

In this, we explore the practical needs of STL development and identify the need for a runtime system that can mediate and manage the security concerns on all data as it passes into and out of an STL application. We provide principles for developing this runtime infrastructure in STLs in general and we provide extensions for Jif to assist in the construction of customized runtime infrastructure. These extensions include an abstraction for I/O channels and a policy infrastructure, SPolicy, for activating and configuring the policy enforced by these channels. We demonstrate these extensions through the construction of a provably secure firewall, the FlowWall.

The principles for runtime infrastructure we discovered in this investigation are (1) the need to isolate dynamic labeling to channels, (2) the importance of limiting the runtime API only to channels that are actually used the application, (3) the value of customizing the semantic granularity of inputs, i.e., the amount of data structure expected, and (4) the ability to configure the runtime system through high-level policy. Following these principles when creating customizable runtime infrastructure facilitates a high-level understanding and configuration of critical policy decisions enforced by STLs. Understanding the policy applied to application I/O is critical for determining whether an application fulfills its high-level policy and thus bridging the semantic gap in security policy.

7.1 Security for STL Applications

As we have seen, establishing overall security properties in security-typed applications depends on the security policies established in three separate parts of the application: 1) the dynamic labeling of inputs and outputs, 2) the static labeling of code in the application, and 3) the relationships between labels (label semantics). In preceding chapters, we have focused primarily on the importance of (2) and (3) with only cursory, ad hoc treatment of (1).

The power of the STL compiler consists primarily in its ability to automatically restrict the labels that may be placed statically on code (2) such that they must honor the labels in (1) with respect to how they may licitly flow, as defined by (3). In other words, so long as the security analyst can verify that inputs and outputs are properly labeled (1) and the label relationships are correctly established (3), the STL compiler will automatically check the rest. The problem is that without careful design of the
runtime system and without an accompanying high-level policy, these three dimensions of the security policy may be scattered throughout the code. This would violate the systems principle of policy separability that we are striving to ensure throughout this dissertation.

7.1.1 Automatic analysis performed by STL compiler


```java
1 public class Stats[label L] {
2     int[L] sshCount;
3
4     public Stats() {
5         sshCount = 0;
6     }
7
8     public void checkPacket[L](Packet pkt) where {pkt} <= L {
9         if (pkt != null && pkt.destPort == 22)
10            sshCount++;
11     }
12 }
13 }
```

One of the compelling features of STLs is their modularity [SM03]—they can ensure security through composition of secure modules. Modules can be separately type-checked for security and compiled, then later combined to make secure applications. The more generally the module’s security properties can be expressed, the more widely it can be used in different applications.

Developing a basic module Consider the Jif code in Listing 7.1. This small module can be used to keep statistics on network packets as they pass through an application such as a network firewall. It is parameterized by a label L with the annotation [label L] on line 1 and has a single member variable, sshCount on line 2, which is guaranteed to be protected at the level of L (indicated by the annotation {L} on sshCount’s type). In this context, “protected” refers to information flow properties such as confidentiality and integrity. This variable keeps a running count of the packets that are being sent to port 22.

Recall that parameterizing a class with a label allows (but does not require) that label to be used within the class. The label parameter must be instantiated when an object of the class is created. For example, the programmer may instantiate a Stats object that is visible only to the firewall administrator as follows:

```java
Stats[{admin:}] statsObj = new Stats[{admin:}]);
```

A key advantage offered by parameterized classes is separability. They serve to separate policy from the class; the class makes no assumptions about L, i.e. it can be specialized with any policy when it is used. An assumption is only introduced when calling the method checkPacket. When called, information about the parameter pkt will
Listing 7.2. A faulty first attempt at extending the Stats class with a method for printing the statistics.

```java
public void printStats()
{
    System.out.println("SSH count: "+this.sshCount);
}
```

implicitly flow into sshCount, therefore Jif requires the programmer to place a restriction on checkPacket to ensure sshCount will protect the information in pkt. This restriction is expressed on the method header with the constraint, where `{pkt} <= L` (line 8). This restriction then limits the call site for checkPacket. At the call sites for checkPacket, where L and `{pkt}` must already be instantiated, the Jif compiler ensures this constraint holds for those particular label instantiations. Whether the constraint holds is determined by the label semantics encoded in the principal hierarchy, defined in a high-level policy external to the application.

Note the advantages of compositionality here. Jif will ensure programs are secure by construction. Stats can be designed apart from any particular application, and when it is inserted into an application, the STL compiler ensures it will not change the security properties of the application. It may restrict the flows in the application, but it cannot introduce any leaks.

Adding system I/O  On the other hand, when system inputs and outputs are added to code, they can modify the security policy of the application. Let us consider the challenges this introduces. It is natural to add a print method to Stats to output the current sshCount. Consider the code in Listing 7.2 extending Stats with a means for printing the current sshCount. Should this be legal? No. The programmer explicitly ensured that the sshCount label (Listing 7.1, line 2) was secret enough to protect information about packets. The method printStats cannot simply print the value to standard output, without discerning whether standard output is secret enough to protect the value.

Labeling I/O with the process owner  A straight-forward fix is to retrieve standard output from a runtime system that implements a particular policy. For example, a reasonable policy defines the standard output to be as secret as the UNIX user who ran the program. This gives rise to the code in Listing 7.3. This is not a bad solution. The Runtime keeps track of the process owner (initialized at program start up) and stores a corresponding user label. When retrieving the standard output stream (line 3), it requires the stream to be parameterized by the user label, userL (indicated with the code `PrintStream[userL]`). Whenever something is printed on the stream, a dynamic check is made, querying the label semantics to be sure that userL is sufficiently secret to protect the data in sshCount, i.e. that L <= userL (line 5). This approach is essentially the approach offered by Jif’s default runtime system.

Notice how policy is encoded in the construction of the Runtime class in Listing 7.3. Making standard I/O as secret as the user running the application is an approximation of an information flow policy. However, this approximation may not hold in different
Listing 7.3. A valid approach to extending the Stats class with a method for printing the statistics.

```java
public void printStats() {
    final label userL = Runtime.userLabel();
    PrintStream[userL] out = Runtime.stdout(userL);
    if (out != null)
        if (L <= userL)
            out.println("SSH count: " + this.sshCount);
}
```

settings: the terminal window may be in plain sight, in which case it should be considered public. In other settings the secrecy of the terminal window may be determined from the windowing system. This facet of runtime infrastructure (indeed all runtime infrastructure) implements application security policy and should be configured based on the security goals and assumptions of the particular application.

The process owner approach is also limited in another way. It prevents the possibility that the output stream could be authenticated to a higher secrecy level. For example, although the application is run by alice, she might have special privileges allowing her to see information on packets. To use those privileges, however, she must dynamically authenticate herself by providing a password. This requires the addition of an input stream. Furthermore, the data retrieved through the input stream must be able to change the security label on the output stream to reflect a valid authentication. This gives rise to the final version of printStats using more advanced data structures to handle labeled I/O channels.

Listing 7.4. Robust approach using Channels.

A new construction

```java
public void printStats() {
    Channel stdio = Policy.getChannel("stdio", null);
    Policy.authenticate(stdio,"stats");
    final label outL = stdio.getNextOutputLabel();
    if (L <= outL)
        if (stdio != null)
            stdio.put("SSH count: " + this.sshCount, outL);
}
```

Listing 7.4 gives our solution to this challenge. In this final version, policy decisions are deferred to a Policy module. The Policy provides methods for retrieving system channels, each in its own Channel object (line 2) and authenticating them (line 3). The Channel abstraction contains a pair of input and output streams. The input stream delivers LabeledObjects which package together the next object on the input stream with its label. The output stream can be queried to determine what level of protection it can ensure for the next object. This label can be used to determine whether it is possible to
output a given object on the stream. The Channel.put method requires that the label on
its first parameter (the object that will be output) be dominated by the second param-
eter (the label it expects for its next output, which may always be retrieved by a call to
Channel.getNextOutputLabel() as on line 4).

As a result, the Jif compiler requires the programmer to guard the call to put
with the dynamic check if (L <= outL) (line 5). We present the Channel, Policy and
LabeledObject classes in more detail in Section 7.2.

The key advantage to our solution is that it adds a layer of indirection, isolating
policy decisions to a policy module that can be configured external to the program. In a
spirit similar to the policy handling in the checkPacket method, the labels and channels
need not be determined here, but only checked at runtime for certain relationships.
The Jif compiler ensures all needed runtime checks are in place. Policy decisions such
as whether to allow standard I/O to be used at all, how it will be labeled, and what
authentication to allow for this channel, are lifted out to a separate module that can be
controlled and configured along with the rest of the policy for the application.

7.1.2 Runtime system principles

To accurately determine the security properties of the entire system in which
this code is executed, it is necessary to analyze the labels dynamically placed on sys-
tem objects (the packets and output stream in this case) and the relationships between
labels (which will determine the result of the dynamic policy check if (L <= Lout)). If
the dynamic labeling of system objects or relationships between labels do not properly
reflect the system security context, the system will fail to meet security goals, despite
its automatically checked objects.

These critical security decisions should not be hidden in application code, but
isolated into separate modules and configured as part of an external, high-level policy.
To this end, we have found the following principles to be effective for moving security
policy decisions regarding the dynamic labeling of I/O out of main application code
and into runtime modules governed by high-level policy. These principles serve as the
requirements for our runtime system and its use. We provide further rationale for the
principles as we explain the additional infrastructure we provide (Section 7.2) and we
return to these principles after presenting our FLOWWALL application (Section 7.3) to
show how they were effective. For now, we simply list them.

Principle 1 *Isolate dynamic labeling* by placing the code for dynamic labeling of system
objects (inputs and output channels) into the runtime system infrastructure.

Principle 2 *Limit the runtime API* such that it is carefully controlled and as minimal
as needed for the applications.

Principle 3 *Customize the semantic granularity* of dynamic system labelings by ensur-
ing that the security context determined for inputs and outputs corresponds with
the desired granularity of control in the application.

Principle 4 *Configure runtime labeling through high-level policy* by governing what
Channels and authentication may be used based on high-level policy.
We define the term *semantic granularity* in Principle 3 to refer to the amount of semantic structure an object has. For example, a stream of bytes has less semantic granularity than when those bytes have been assembled into IP packets or emails. This is not a strict measurement but intended to reflect the insight that the security properties of inputs and outputs can often depend on the semantics of the data, and a datum’s semantics cannot be understood until the data is parsed to a higher semantic granularity.

### 7.2 Runtime system

In this section, we present a new runtime system infrastructure, the SPolicy infrastructure, designed and implemented according to the principles presented in Section 7.1.2. Figure 7.1 provides an overview of our infrastructure for compiling policy into a Jif application. In this system, the programmer is responsible for developing Jif application code. If the application requires specialized runtime components (our FLOWWALL requires a component to interact with the network packet stream using a special libipq library, e.g.) for communicating with the host system, these must also be provided along with Channel interfaces for using them. The Jif flow policy, JPolicy, and runtime system policy, SPolicy, can be customized by the application deployer. The SPolicy determines what channels can be activated as well as (optionally) configuring how they handle labeling and authentication.

Listing 7.5. Jif signature API for Channel abstraction.

```java
public abstract class Channel[label L] {
    public abstract LabeledObject[L] getNextObject(L);
    public abstract label[L] getNextOutputLabel(L);
    public abstract void put(*lbl, Object{*lbl} obj, label{*lbl} lbl);
}
```

Our two key contributions are the Channel abstraction and the SPolicy compiler; they work in concert with the Jif compiler [MZZ+] as well as the JPolicy compiler, presented in Chapter 4. The SPolicy compiler produces an SPolicy object based on the policy it has been given. The Channel abstraction can be extended to implement different kinds of channels. The SPolicy class controls what Channels can be used when executing the application and may configure some labeling and authentication schemas used by the Channels. The SPolicy class should only be generated automatically from a high-level policy and then linked into a final application when it is executed. In this way, the channels and authentication used by an application can be controlled through high-level policy while still providing for separate compilation of application modules.

#### 7.2.1 Channels

The Channel abstraction and policy infrastructure were designed in light of years of experience in building STL applications. The two key goals guiding their development were that they would facilitate the separation of security policy specification from
implementation (allowing control through high-level policy) and that they would be sufficiently expressive, yet still general enough to capture the runtime interactions required by diverse applications.

The basic Channel API is shown in Listing 7.5. Channels cannot be created directly (the API disallows this); they can only be instantiated through the SPolicy class. The SPolicy class is configured using high-level policy and may include or exclude the methods to instantiate various Channels. This serves to separate policy specification from its implementation in the application.

A Channel delivers labeled objects from the system to the application (inputs) and from the application to the system (outputs). Channel.getNextObject returns an object packaged with its label in a LabeledObject. For outputs, Channel.getNextOutputLabel returns a label and Channel.put only accepts outputs with lower security requirements than expressed by that label.

Past experience building STL applications exposes the main challenges for developing a channel abstraction that is both sufficiently expressive and sufficiently general to be useful in a wide variety of settings. These challenges include that 1) labeling of I/O depends on the security mechanisms offered by specific environments (contrast SELinux mandatory access controls with UNIX ACLs on files with authenticated sockets, etc.). 2) Labeling of data sometimes depends on the structure of the data itself (the To: address on an email or the source address of a packet, e.g.) or 3) on a series of data exchanges (an authentication protocol, e.g.). 4) The protection offered on an output channel may depend on data (like an authentication token) that has been received on a companion

Fig. 7.1. Infrastructure for compiling security policy into Jif applications for provable enforcement of policy.
input channel. 5) As the protection on an output channel may change over time, the output channel API must include a means to determine its current label.

These challenges guided the design for the Channel. Simultaneously, we ensured that the Channel would help the programmer meet our principles for sound runtime system development. Firstly, a Channel maintains state between uses. By keeping track of some previous inputs and outputs, the Channel can track authentications and modify labeling policies based on transmitted data. This is critical for handling data-specific labeling (2) and authentication protocols (3). Also, when the system must be queried after a stream is opened to determine the security context (1), this context can be saved as part of the state of the Channel. Secondly, input and output streams are coupled together in a single Channel to allow input data to affect the labeling of outputs (4). In our experiments, we did not find a need for multiple inputs to be coupled with a single output or vice versa, although some channels could naturally have an input (or output) such as a read-only (write-only) file with no corresponding output (or input).

Another innovative quality of Channels is that they can be queried to determine what protection they offer for the next output. A Channel connected to a file may return a label indicating that it can protect data according to the security properties on the file. Here are three examples. A world-readable UNIX file could return public. A Channel that encrypts its output may return a label indicating the security properties of the encryption it uses. A terminal window output Channel may return the label for the user for which it is authenticated. As noted in (5) above, since this can change (e.g., standard output may revert to public when the user logs out; a file may be upgraded through a chmod operation), the label must be retrieved for each output. Then the application is responsible for converting the label on its data before outputting it on the Channel. Likewise, each input read from a Channel is packaged with its label in a LabeledObject type that we have provided for this purpose. This label need not be known a priori (in contrast to previous Jif systems).

Channels can operate at the level of semantically expressive Objects rather than only streams of bytes. They provide a specific object type as input and expect a specific object type (a String, a Packet, or some other data structure) as output. Hence, Channels can adjust I/O according to the proper semantic granularity demanded by the application. This is an important feature for handling the challenges of data-specific labeling (2).

Finally, a key design goal of the Channel abstraction was to make sure that Channels could be activated and configured by high-level policy. We have included a level of indirection for activating each kind of Channel, by not allowing Channels to be instantiated except through SPolicy. Another level of indirection allows customization of the labeling policy for each Channel input and output. Finally, external classes implementing authentication protocols can be activated to customize Channel instances in different settings.

### 7.2.2 Example

The policy given in Listing 7.6 defines the semantics for the standard I/O channel given in Section 7.2.1. It starts with public labels and allows the user to authenticate
Listing 7.6. Policy entries for standard I/O channel.

```
channel policy "stdio" {
    channel pol.StdioChannel
    authentication "stats"
    labeling "flowwall"
}

authentication policy "stats" {
    pol.StdioAuth.pwdauth ["stats-pwds.txt"]
}

labeling policy "flowwall" {
    init pol.StdioChannel.setPublicLabel
    inputs pol.StdioChannel.getCurrentLabel
    outputs pol.StdioChannel.getCurrentLabel
}
```

Listing 7.7. Policy class generated automatically from the policy in Listing 7.6.

```
public class Policy {
    static public void authenticate(Channel channel, String authType, Label l)
        throws PolicyException {
        if (channel instanceof StdioChannel && authType.equals("stats"))
            StdioAuth.pwdauth ((StdioChannel)channel, l, "stats-pwds.txt");
    }

    static public Channel getChannel(String channelType, Object params, Label l)
        throws PolicyException {
        if (channelType.equals("stdio"))
            return StdioChannel.getInstance();
        else return null;
    }
}
Listing 7.8. A simple use of standard I/O.

```java
final Channel[] stdout = Policy.getChannel( "stdio", null, new label{} );
if (stdout != null) {
    final LabeledObject[] obj = stdout.getNextObject();
    if (obj != null) {
        String str = (String) obj.getObject();
        stdout.put("You entered " + str, obj.lbl);
    }
}
```

herself as having “stats” privileges. The StdioChannel contains the standard input and standard output streams. It prints Strings to standard out and retrieves Strings from standard in. Firstly, the channel must be enabled by adding a policy entry to the high-level policy file. The channel line of this policy (Listing 7.6, line 1) indicates that the StdioChannel should be enabled in the SPolicy and can be selected in the application with the String "stdio" (Listing 7.7, line 10). Without such a policy line enabling StdioChannel, it could not be used in the application.

The initial label on inputs and outputs is established by the method `pol.StdioChannel.setPublicLabel` as defined in the stdio’s labeling policy’s `init` field. This method establishes standard I/O as being public initially (this will correspond to the desired policy for the FLOWWALL in which no one is allowed to see statistics on packets until authenticated as having that privilege). The labeling policy for this channel is quite simple—the method `getCurrentLabel` merely returns the current label (which is kept as part of the state for the channel). To raise the security of the channel, an authentication module may be used in the application code, in this case causing `pol.StdioAuthenticate.pwdauth` to be called with a parameter, "stats-pwds.txt" indicating the location of the password file.

A StdioChannel may be used as previously shown in Listing 7.4 or as shown in Listing 7.8.

7.2.3 A principled runtime system

A design goal for the Channel and SPolicy classes was to guide programmers in implementing the principles given in Section 7.1.2.

**Principle 1 Isolate dynamic labeling** This is achieved by pushing all the dynamic labeling decisions into the Channel classes. Each object is labeled as it enters the application, as determined by the semantics of the Channel class. Likewise, the Channel class limits the objects that can be output from an application, based on the object’s label.

**Principle 2 Limit the runtime API** Because the runtime API is governed by what Channels can be retrieved from the SPolicy class, the Channels used by an application can be easily limited to what is needed.
Principle 3 Customize semantic granularity  An alternative design for Channel is to restrict Channel only to read and write individual bytes. This fails to accommodate the needs each application has for a specific semantic granularity of inputs and outputs. On the contrary, our design requires Channel to get and put Object, freeing the developer to design the semantic granularity as appropriate to the application.

Principle 4 Configure runtime labeling through high-level policy  This principle is met through our policy infrastructure which allows high-level policy to be compiled into a SPolicy class that is specialized for each application. The SPolicy class governs which channels can be used in an application and how the channels can be authenticated. It also allows some configuration of these decisions, specifying credentials repositories such as a keystore or password file.

In the following section, we present FLOWWALL, focusing on how the Channel class and policy infrastructure were extended to handle the unique demands of this application.

7.3 FLOWWALL

To evaluate our design principles, we apply our approach to a real-world application, a network firewall. Network firewalls are a well-known part of the security infrastructure of almost every computer. Ensuring that a network firewall properly implements its policy is not always an easy problem [KFS+03], however, and benefits from automated assistance. The task of a network firewall is, essentially, to prevent illicit network packet flows across a particular boundary (a particular computer, an enterprise router, etc.). Based on the firewall policy rules, packets should be accepted or dropped, as shown in Figure 7.2.
Listing 7.9. IPTables firewall rules

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iptables -A FORWARD</code> -d 192.168.1.20 -j DROP</td>
<td></td>
</tr>
<tr>
<td><code>iptables -A FORWARD</code> -s 192.168.1.0/24 -d 192.168.2.0/24 -j ACCEPT</td>
<td></td>
</tr>
<tr>
<td><code>iptables -A FORWARD</code> -s 192.168.2.0/24 -d 192.168.1.0/24 -j ACCEPT</td>
<td></td>
</tr>
</tbody>
</table>

A basic firewall policy signifies this with rules as in Listing 7.9. Rules may contain a \(-s\) flag to indicate the source address (including optional port number\(^1\)), a \(-d\) flag to indicate the destination address (including optional port number) and a \(-j\) flag to indicate whether the packets should be accepted or dropped. The rules are evaluated top-down. In Listing 7.9, if a packet matches the first rule (i.e. its destination is IP address 192.168.1.20, any port), it is dropped and further processing stops. If it does not match this rule, it is processed by the last two rules. The last two rules match on a 24-bit subnet for both source and destination. The ranges that this notation represents are 192.168.1.0 – 192.168.1.255 and 192.168.2.0 – 192.168.2.255. In the next section we describe how to implement this application in an STL.

7.3.1 An information flow policy for a firewall

In contrast to normal software development processes, the first step in developing any security-typed application must be to determine what kind of information flow policy it will enforce. This design phase is critical in security-typed languages and constitutes one of the greatest challenges to security-typed application development. Haphazard setup results in extremely difficult programming, because it leads to many labeling conflicts, which usually induces a cascading of relabeling throughout the application. For the FlowWall we want to maintain a simple information flow policy:

*Packets arriving from a given source address may only flow to their destination address if allowed by the firewall policy rules. All other packets are dropped.*

As described in Section 7.1, application security analysis depends on (1) how system objects are dynamically labeled, (2) how labels are propagated on code throughout the application and (3) the relationships between labels (label semantics). We illustrate the basic structure of the FlowWall in Figure 7.3 such that these three areas can be identified. To determine whether this policy is fulfilled by FlowWall, it is necessary to determine (1) that input packets are labeled with their source addresses, (2) that the application type-checks and (3) that the relationships between source addresses and destination addresses are in compliance with the firewall policy rules. Comparing Figure 7.2 (in which the entire application must be hand-checked to determine compliance with policy) with Figure 7.3 displays the advantage of STLs: the STL compiler will

---

\(^1\) Port numbers are left out here for simplicity of presentation, but we handle them in the natural way in FlowWall.
automatically ensure that all code inside the inner box is secure, i.e. that it honors the labels on inputs and behaves according the FlowWall policy. This requires the security analyst only to check by hand the three API methods for the PacketChannel.

As other functionality is added to examine or process packets, the Jif compiler will ensure that they maintain the security properties for packets as we showed earlier for Stats.checkPacket (Section 7.1.1). In this regard, we can treat additional modules as black boxes since they are guaranteed to sustain (or possibly somewhat restrict) the security properties established on data at their call sites.

We now present the design of FlowWall, focusing on the three elements which impact its overall security, the runtime system, application code and the high-level policy.

7.3.2 Runtime system: Labeling inputs and outputs

The FlowWall needs a way to retrieve packets and a way to output packets. Packets that are not output are dropped. In light of these requirements, FlowWall maps naturally onto our Channel abstraction.

The key insight for implementing FlowWall is to use the source and destination addresses on packets as the security labels that will govern how packets can flow through the FlowWall. In light of this insight, the PacketChannel’s input channel delivers packets labeled with their source address. When a packet is retrieved from the PacketChannel, the channel’s internal state reflects this by changing the label for the PacketChannel’s output channel—it will only accept a packet labeled with the proper destination address. This corresponds to the expected firewall policy that a packet can only be accepted if it can flow from its source address to its destination address. Having set up the runtime system I/O labeling, the final step is to configure the label semantics. The label semantics must reflect the firewall policy. If a source address $s$ may flow to a destination address $d$ in the firewall policy then the Jif label corresponding to $s$ must be dominated by the Jif label corresponding to $d$ in the label semantics (controlled by JPolicy). This facilitates the code in Listing 7.10, which demonstrates the central processing loop of the FlowWall.

As expected, the API for packet channels meets the principles set forth in Section 7.1.2, because it utilizes the Channel pattern we established. Namely, Principle 1 is fulfilled by isolating all dynamic labeling of system objects (only the packets in this case) in the runtime infrastructure. Principle 2 is fulfilled by limiting the runtime API to include only the needed interface—for packets in this case. Principle 3 is fulfilled by basing the dynamic labels on source and destination addresses for the packets, which is the granularity of control desired for the application (not on data in the packets or any other criteria). Principle 4 is fulfilled by describing all possible address ranges for packets in the FlowWall policy and giving the licit flows by relating source and destination addresses. We describe these last two points, the granularity and configurability now in more detail.
Listing 7.10. Main loop for getting, processing and accepting or dropping packets.

```java
final Channel[] pktChannel =
    Policy.getChannel("packet", null);
...
while (true) {
    // code to handle one packet
    if (pktChannel != null) {
        final LabeledObject[] obj = pktChannel.getNextObject();

        // any processing of packets ...

        final label[] destL = pktChannel.getNextOutputLabel();
        if (obj != null) {
            Object pkt = obj.getObject();
            // read "<=" here as "may flow to"
            if (obj.lbl <= destL)
                pktChannel.put(pkt, destL);
        }
    }
}
```

7.3.3 High-level policy infrastructure

As shown in Figure 7.1, the two parts of high-level policy include an RTS policy governing the dynamic labeling of I/O and a Jif flow policy, defining the label semantics (and thus, the legal flows). The SPolicy for FLOWWALL must at least activate the PacketChannel with the labeling described above. The Jif flow policy must be a faithful encoding of the IPTables firewall policy. Then we implement the policy through the JPolicy compiler (described in Chapter 4) and our SPolicy compiler. Both policies compile into bytecodes that are linked into the application when it is executed. We now consider the advantages of this approach with regards to our principles, especially focusing on the last two principles: semantic granularity and high-level policy configuration.

7.3.3.1 Semantic granularity

Principle 3 asserts that the dynamic labeling provided in the runtime system should be targeted to a semantic granularity that is appropriate to the application. This refers to the amount of security context which must be determined before a label can be applied. One level of semantic granularity would be to consider all data input from the network merely as a stream of bits and label each bit the same way. This might be useful for enforcing a simple information flow policy that prevented network data from being saved to disk or printed to the screen, for example. Labeling network streams, disk streams and stdout with three different labels, {network}, {stdout} and {disk} for example, and then ensuring that {network} $\nleq$ {stdout} and {network} $\nleq$ {disk} would be sufficient to achieve this information flow goal. In this case, there is no need to recover the semantic granularity of packets, source addresses and destination addresses.
As another example, we might be interested in a different level of semantic granularity which abstracts packets into SSL channels. In an instant messaging chat client, for example, we may want to label streams of String data with a label indicating the public-key certificate and certificate authority used when establishing the SSL connection.

Because the semantic granularity of labels depends on the security goals of the application, this cannot be achieved in a general way. Rather, the dynamic labeling runtime system for applications must be specialized based on the security goals of the applications and what level of information flows they seek to control. This motivates the important design goal of our policy and infrastructure: to simplify the specialization process and improve reusability of RTS components.

For the FlowWall we are concerned with labeling packets as they enter the application and ensuring they have proper labels as they leave the application. Thus, labeling packets based on the context of their source and destination addresses is the best choice for the semantic granularity of labels. Namely, the security context of incoming packets consists of the source address and the security context of outgoing packets consists of the destination address. Whether a packet can flow from source to destination is governed by the firewall policy and enforced by the relationship between source labels and destination labels (label relationships are discussed in more detail below).

7.3.3.2 Establishing relationships between labels

Establishing the relationships between labels is critical for enforcing security policies over applications. The labels determine the expressiveness of the security policies that can be enforced. The relationships between the labels govern the ways that information can flow through various methods and objects in an application. For example, in a traditional military security lattice [BL75], it is acceptable for unclassified information to flow into secret documents, but not vice versa. In this context, the label \{unclassified:\} is dominated by the label \{secret:\} allowing information flows from unclassified to secret. Relationships must be established between labels; for the sake of security analysis and separation of policy specification and implementation, this should be done as much as possible in policy separate from the code, as stated in Principle 4.

The IPTables firewall policy must be isomorphic to the Jif policy used by the application. To ensure this is the case, we provide a specialized firewall policy compiler, just for use in FlowWall, that automatically generates the Jif policy from the IPTables policy.

The conversion is quite simple. As an example, consider the policy in Listing 7.9. For this policy, we want to ensure that source address 192.168.1.0/24 (any port) may flow to destination address 192.168.2.0/24 (any port). To express this in Jif, we compile this into two composite Jif principals, s_192_168_1_0_255_any and d_192_168_2_0_255_any and add a delegation to the source principal which establishes the policy s_192_168_1_0_255_any ≤ d_192_168_2_0_255_any.

Additionally, because these principals express the policy of (any port for) a range of possible source and destination addresses, we need a conversion from a single source address and port number, e.g. 192.168.2.15:22, to the composite address.
s_192_168_2_0_255_any. The code for this conversion is hooked into the PacketChannel through the high-level policy. It handles the firewall policy rules in a top-down fashion, sometimes needing to break up regions into multiple parts when multiple rules apply.

To model the semantics of “drop” rules in the IPTables policy, the policy must be compiled from bottom up, removing flow arcs when a drop rule is encountered. For the sample policy, the first rule requires breaking the composite rule 192.168.1.0/24 into three principals, s_192_168_1_20_any for the single IP address 192.168.1.20, s_192_168_1_0_19_any and s_192_168_1_21_255_any. Rule 2 adds an arc to s_192_168_1_20_any, but when Rule 1 is processed, that arc is cut.

7.4 Evaluation

7.4.1 Analyzing FlowWall’s security

A security analysis of the application is driven by the high-level policies for the RTS and the label semantics. The security goals for FLOWWALL were that it would accept or drop packets exactly in accordance with its IPTables firewall policy. Clearly, this depends on the application inputs, outputs and intermediate flows. The advantage of our approach is that all application inputs and outputs can be determined from the RTS policy. This reveals that packets enter and leave through the PacketChannel.

In order to determine how packets can flow through the application, we must know 1) how they are initially labeled, 2) how data with that label can flow through the application and 3) what labels must be on outputs. For determining input and output labels, we consult the SPolicy, which directs us to some code. This code must be carefully checked to ensure that packets are labeled with their source addresses. The SPolicy also directs us to code that reveals that the acceptable output labels for PacketChannel are always derived from the destination address of the last input. Finally, checking the Jif flow policy compiled automatically from the IPTables policy, we complete our analysis: 1) input packets are always labeled with their source addresses. 2) A packet labeled with its source address can only be relabeled to destination addresses allowed by the Jif policy and the Jif policy allows relabelings isomorphic to the IPTables firewall policy it was compiled from. 3) Only packets labeled with the destination address of the last input packet can be output; the rest are dropped.

In short, presuming the correctness of our compilers, we have a packet-filtering firewall that provably enforces its policy. This can be determined almost entirely without examining the application code, through inspection of the high-level policy and manual inspection of some labeling runtime infrastructure indicated by the policy. Inspection of the application code is only needed to ensure basic correctness: that the application actually reads any packets at all and does not just drop all packets.

This demonstrates our goal to show that security-typed languages significantly aid programmers in bridging the gap between high-level policy specification and code-level enforcement of that policy. At the same time, we have shown how runtime infrastructure plays a critical role in that implementation. Furthermore, using principled design, exemplified by our Channel pattern, we can allow the bulk of policy decisions to be deferred to an external, high-level policy.
7.4.2 Performance

<table>
<thead>
<tr>
<th></th>
<th>Latency</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>No firewall</td>
<td>29.15 µs</td>
<td>4.18 Gbps</td>
</tr>
<tr>
<td>Accept all</td>
<td>55.71 µs</td>
<td>1.80 Gbps</td>
</tr>
<tr>
<td>FlowWall</td>
<td>83.66 µs</td>
<td>1.29 Gbps</td>
</tr>
</tbody>
</table>

Table 7.1. Performance measurements.

Because performance is not our central consideration, we refer the reader to a more detailed performance evaluation reported elsewhere [Mis07]. Our tests were run on an Intel 2.4 GHz Core 2 Duo with 2 GB of RAM, running Ubuntu Linux. In summary, FlowWall demonstrated performance expected of a Java-based application without any effort spent on optimization. At the same time, it demonstrated processing throughputs sufficient for an average Ethernet setting using gigabit Ethernet cards. In Table 7.1 we compare packet processing latency and throughput of FlowWall to a minimal C version, both using an “accept all” policy.

Despite the notable reduction in bandwidth for both firewalls, they were both still capable of processing significantly more than 1 gigabit per second. This performance would be more than sufficient, even if the host were using gigabit Ethernet cards for all network connections. As a result, we conclude that the performance of FlowWall is sufficient with respect to bandwidth for deployment in a production environment.

7.4.3 Evaluating prior applications

The number of realistic applications is still very small, but we each would benefit from a principled, configurable runtime system as given in this paper. We now evaluate each of them (including our own) for the way they implement the four principles we provide for runtime system development. We summarize our evaluation in Table 7.2.

**Mental Poker** This application uses a very simple policy to test a cryptographic protocol for playing poker without a trusted third party. The system consists only of two Jif applications. The inputs and outputs of each application are limited to communication via a shared file. The file is treated as publicly visible and so all sensitive data is encrypted before writing to the file. This application definitely limits the runtime API to what is minimally necessary, capturing Principle 2, but only employs Principle 1 in part. Some dynamic labeling of inputs occurs at the point of decryption which happens within the application code, not in the runtime system. Likewise, an encryption-induced declassification labels all outputs to public. The input stream and output stream which are retrieved from the runtime system are both labeled public. Because Principle 1 is only partially implemented, more code within the security-typed application must be checked and less code within the runtime system. Neither Principle 3 nor Principle 4 are
Table 7.2. P1: isolate all dynamic labeling to runtime methods. P2: limit API functionality to what is strictly necessary for the application. P3: specialize the semantic granularity of I/O to match security goals of the application. P4: ensure that dynamic labeling is configurable through external policy.

<table>
<thead>
<tr>
<th>Method</th>
<th>P1 Dynamic labeling</th>
<th>P2 Limited API</th>
<th>P3 Semantic granularity</th>
<th>P4 High-level policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mental poker</td>
<td>Partial</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>JPmail</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Partial</td>
</tr>
<tr>
<td>SELinux RT</td>
<td>Yes</td>
<td>Partial</td>
<td>Partial</td>
<td>Yes</td>
</tr>
<tr>
<td>SIF</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Inference</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>FLOWWALL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

utilized which makes it difficult to tell what the information flow policy is that is automatically checked by the STL compiler. Much human reasoning is needed to establish this.

**JPmail** The original implementation of JPmail, described in Chapter 3, controls the flow of emails through a high-level policy. Network channels are considering publicly visible, so emails are encrypted before being sent out and decrypted on receipt. One novelty of JPmail is the use of a high-level policy, JPolicy, for trusted declassification (described in Chapter 4). It allows control over code-level declassification decisions through an external policy. This takes a step towards Principle 4, although the high-level policy does not specify any dynamic labeling or principal authentication.

JPmail has complicated interactions with the runtime system. Like mental poker, all network channels are treated as public and all dynamic labeling of encrypted emails occurs within the code. The code that handles dynamic labeling is the most complicated section of code in the application, resulting from bad matching of the semantic granularity with the application security goals (Principle 3). Ideally, the parsing and labeling of emails would happen in the runtime so that the policies on the emails could then be automatically ensured throughout the application.

JPmail also has various other interactions with system objects, including files for storing passwords, certificate authorities, keystores and standard I/O. The labeling of these inputs and outputs is not made explicit through an external policy, making it difficult to determine what system-wide security properties are implemented by the application. JPmail handles orthogonal principals including UNIX users and PKI identities. Labels based on the UNIX user are used for files and standard I/O, while PKI identities are used on emails. It utilizes the runtime system to perform an authentication in order to connect these two principal sets.

**SELinux runtime** The SELinux runtime API used for `logrotate` and an MLS version of JPmail, described in Chapter 5, has some unique advantages in that it is specialized for
a particular runtime environment, but re-usable for multiple applications. All files and
network sockets have specialized labels controlled by a mandatory policy that enforces
information flow goals. Thus, the mapping between system security mechanisms and
STL information flow policy need not be approximated but can be achieved directly.
Furthermore, because the applications are designed to handle data at the same semantic
granularity as these MAC labels (e.g. all data in a file is considered to have the same
label as the file and all data in a network socket has the same label as the network
socket), the application development was simpler and the security properties are clearer
than in the original JPmail. In accord with Principle 1, all dynamic labeling occurs in
the runtime system. The dynamic labels that can be applied are captured in a high-level
policy (as per Principle 4).

This application of STLs benefits from a significant advantage in being able to
leverage fine-grained, mandatory access control (MAC) labels on system objects. The
result is that with little code or complexity, a rich security context for system objects can
be recovered just by examining their MAC labels. So long as applications can operate
at this semantic granularity (i.e. not requiring files to be parsed or emails to be re-
labeled based on their data), this runtime system can be re-used quite effectively for
providing applications with strong, automatically checked security properties that can
be understood and configured through external policy.

SIF SIF [CVM07] provides a general runtime system for handling Jif web servlets. SIF
provides a controlled environment in which Jif servlets can be executed. Servlets receive
low integrity action commands from web clients delivered through the SIF framework.
Servlets may also receive confidential information from the web server through back-end
databases. These inputs are labeled dynamically through SIF and the Jif servlet provably
enforces the policies provided by dynamically labeled inputs (preventing low integrity
inputs from being trusted) and the limitations placed on output channels (preventing
the leakage of confidential server data to clients, e.g.).

Although SIF provides a runtime API, it also requires each servlet to specialize
this API by providing some of its own dynamic labeling services. In this way, the
labels placed on input data in a calendar applet can be very different than the labels
placed on messages in an instant messaging service. Thus, they specialize the semantic
granularity (Principle 3) of I/O to suit each application. SIF is very careful to follow
Principle 2, limiting the broader API provided by the Java servlet framework to a more
controlled interface for Jif servlet inputs and outputs. Their major limitation is that the
labeling of inputs is not expressed in any high-level policy (Principle 4). This muddies
an understanding of what security properties are enforced by the servlets and requires
changing the code to re-configure the applications for different security properties.

Sj healthcare application Tse’s Sj language, developed concurrently to this work,
seeks to address many problems related to this work, especially with regard to dynamic
labeling and authentication-based policy decisions. The healthcare application is limited
in size and complexity, although it addresses many realistic policy decisions, guided by
Anderson’s requirements [And96, And00] for healthcare policy. As this language was
developed quite recently [Tse07], further exploration will be necessary to understand the
advantages and disadvantages of their approach. One of their stated points of future work is to understand how their policy system can be implemented in Jif. What we describe in this chapter may provide the proper setting in which to carry this out.

**Thober’s inference** Although not a security-typed language application, Thober et al. deserve mention for producing a limited, general approach to converting Java programs into information flow secure programs. Thober’s automatic inference for runtime policy [ST07] shares our insight about the importance of labeling inputs and outputs for understanding the end-to-end security of the application (Principles 1 and 2) and they capture this labeling in an external policy (Principle 4). They claim that all application labels can be derived based solely on this input information. Their system fails to recognize, however, the importance of establishing the proper semantic granularity for labels (Principle 3). They presume that all inputs and outputs have constant, static labels that can be applied to all data that passes into and out of the particular channels. While their system has advantages for guaranteeing some information flow policies on applications, it fails to incorporate richer, more realistic policies and would be unable to express the kinds of policies found in JPmail, the SIF servlets or even `logrotate`. One possibility is that their high-level declassifiers and endorsement functions, attached to streams, might be able to simulate some richer labeling of I/O.

### 7.5 Summary

The importance of security-typed languages for developing strongly secure systems has gained much credibility in the research community over the past few years. The importance of having strongly secure components whose security can be automatically checked and easily reasoned about promises to aid greatly the overall security of complex systems. Various obstacles have prevented them from being utilized widely, however, including the need for runtime systems which must be specialized per application and operating system environment and the need for high-level policy infrastructure which supports separation of security policy specification and implementation.

In this chapter, we have identified the need for runtime system and high-level policy infrastructure, presented some principles that can be used to guide the development of such infrastructure for future applications and have demonstrated the utility of our principles on a small application, a network firewall. We have found these principles to be effective for evaluating past efforts as well as for developing the FlowWall.
Conclusion and Future Work

Saint Benedict’s policy for monks has been in use for over fifteen centuries and has, arguably, provided a significant formative influence over the development of western civilization. One of the central qualities of the rule is its separability. By not hard-coding all the rules and over-specifying behavior for each situation (contrary to the rule’s predecessor and other contemporary monastic rules), but rather leaving many policy decisions to the Abbot (the policy administrator and enforcer), Saint Benedict’s rule proved to be portable to every country and culture throughout the world over the course of centuries.

On the other hand, little exploration is necessary to show that the policy in Saint Benedict’s rule was not always enforced. Cursory examination of contemporary monasteries, as well as reading historical accounts quickly reveals that the Rule of Benedict, despite its flexibility, portability, expressiveness and practicality, was not verifiably enforced [Cha50, Law84].

In contrast, security-typed languages provide verifiable enforcement. Our investigation shows that the state-of-the-art falls short in incorporating some key systems principles needed for developing secure systems, namely separability, expressiveness, portability and modularity. Their capacity for verifiable enforcement provides a strong foundation. These systems construction principles are necessary, in turn, for achieving the ultimate goal of this work, to bridge the semantic gap between high-level security policy semantics and the operational semantics of applications. Here we revisit each of the four systems principles needed for bridging the semantic gap and then show how they combine to give the desired result of provable implementation of high-level security policy in applications.

8.1 Modularity, Expressiveness, Portability, and Separability

Modularity The high-level policy for JPmail was that the bodies of emails should not leak to unintended recipients. Implementing this policy in a real computer system requires email servers, file systems, and networks in addition to the JPmail email client. To build JPmail in an STL that cannot interact with non-STL components would require an operating system, network infrastructure and email servers all built in STLs.
This is unrealistic, at least for the immediate future. To remedy this, our work aims at constructing realistic, modular applications that can interact with existing systems. Handling this securely, however, necessitates STL principal infrastructure, like we provide, that roots principals in real-world identities such as a PKI, OS user database or IP address.

**Expressiveness** Software verification tools can only serve to bridge the semantic gap in security policy, if they can verify that an application’s operational semantics correspond with the semantics of real, high-level policies. To take this to its fullest extent would require verifiably translating natural language policies into high-level policy languages with formal semantics. At the least, however, it must be possible to verify that software enforces policy that is semantically close to the natural language policies. Our research shows that information flow policies based on noninterference are close to what is needed, but somewhat too strong. In JPmail, for example, it is necessary to introduce declassifications when placing information on a public network, preparing email for a remote recipient, and sending passwords over the network, to name a few. In other applications, such as logrotate and FLOWWALL, noninterference provided appropriate expressiveness without additional declassification technology, but implementing richer policies for these applications (filtering packets, sending log reports to system administrators, etc.) likely requires trusted classifiers.

**Portability** As we showed throughout, in order to understand the policy enforced by an application, it is necessary to know all the channels into and out of an application as well as how the channels apply labels to the data flowing through them. For example, without knowing that network packets can only flow into and out of the FLOWWALL through the networking subsystem, it is not possible to know the firewall provably enforces its policy. Additionally, it is necessary to know how the input packets will be labeled and what labeled packets may flow out of the application. The STL provides certainty that the application will obey these labeling policies, but they must be made explicit in order to bridge the semantic gap in security policy. While programming language models have typically handled STL applications as if they were closed systems, we support portability to diverse systems through the SPolicy runtime infrastructure.

**Separability** Separation of policy from mechanism is critical for bridging the semantic gap in security policy, because it reduces an application’s operational semantics to a smaller, policy-based semantics. The policy may then be compared reasonably to high-level, natural-language policies. Other STL projects made extensive searches through the whole code base to analyze each declassification [AS05, CVM07], and had to explain in natural language how I/O labeling took place in the runtime interface. In contrast, a guiding principle throughout our work was to isolate policy specifications to external policy files that could be automatically translated into code that could be automatically verified for security by the STL compiler. The JPolicy and SPolicy infrastructures, including compilers and code bases, serve this end.
Each of these criteria is crucial for bridging the semantic gap in security policy and provably implementing policy specifications in practical systems. We are now in a position to evaluate our original thesis.

8.2 Bridging the Semantic Gap in Security Policy

To bridge the semantic gap between the semantics of a high-level policy specification and the operational semantics of a real system, we implement principal infrastructure needed for building modular applications; we implement label policies that are expressive enough to capture the semantics of real, high-level policies; we expand runtime infrastructure to facilitate greater portability to diverse operating environments; and we maintain a policy that is separate from the application code. In this way, it is possible to hold up our policy specification to see that it expresses high-level policies. STLs are then able to verifiably implement our policies over a given application.

Alone, STLs can connect to high-level policy only through limited applications and with a limited policy, as shown on the left-hand side of Figure 8.1. By adding modularity to STL applications, they can be used in more robust, realistic systems. The JPolicy infrastructure serves this end, rooting application principals in real-world entities outside the application. The dimension of portability serves to continue expanding the scope of STL application power and interplay with real systems. Our SPolicy infrastructure fulfills this interaction, mediating all information flows that cross application boundaries and coordinating how applications work with diverse system security mechanisms. Altogether, STLs with realistic principals and dynamic labeling of I/O can be made into practical computer systems. This is illustrated in Figure 8.1.

To bridge the semantic gap, however, it is necessary to connect these applications with high-level policy. The separability of JPolicy and SPolicy from application code...
allows the application’s operational semantics to be captured in high-level policy and the static analysis of STLs ensures this mapping is sound. Furthermore, by improving the expressiveness of these policies, the high-level SPolicy and JPolicy specifications draw closer to capturing the high-level policies exactly.

In light of these developments, we can validate our original thesis.

The application of systems principles to security-typed languages enables the development of realistic, secure systems that bridge the semantic gap between high-level security policy semantics and the operational semantics of computer systems. The systems principles needed for achieving this goal are modularity, portability, separation of security policy from mechanism, and security policy expressiveness.

Setting forth from this foundation with renewed hope in this programming language technology, we explore some remaining questions that emerged from this work and will serve to guide further research.

8.3 Future Work

In this thesis we put forth architecture, infrastructure, analysis tools and software engineering tools for building secure systems that provably implement high-level policies. Thus far, we have constructed only proof-of-concept applications to explore the power and applicability of STLs. A major thrust in future work is to use the tools and infrastructure for building more realistic and robust secure systems. We expect that scaling up the complexity of applications and policy will uncover further needs for development. Another major thrust in future work is expanding the policy infrastructure such that policy can be stored and administered from a policy server separate from the applications that use it. This introduces challenges in both systems and theory, requiring an architecture for policy queries and also a model for policy administration. A third major thrust in future work is to investigate the interaction of STLs with additional systems security mechanisms for enforcing system-wide, high-level policies. We describe these and some other open areas of research in the following subsections.

8.3.1 Improving portability

In this work, we have shown how STLs can be used in multiple operating systems environments, interfacing with diverse systems security mechanisms and communication protocols. Future work remains, however, for extending this experience to leverage security mechanisms provided in more contexts. We believe that this should be quite achievable through the Channels abstraction and SPolicy infrastructure, but this must be explored in more realistic systems.

Also necessary is a formal, theoretical analysis of the security properties provided by Channels. Similar to Tse’s formal reasoning about interactions with the system through Sj [Tse07], some theorems on the security of Channels must be proven.
Furthermore, our and others’ experience of using STLs for developing applications in diverse settings raises the question of what systems infrastructure may be provided for the deployment of STL applications. Our investigation showed that MLS labels in MAC operating systems were particular useful for expressing system-wide security properties of data as it comes in from and flows out to the system beyond the application. Also important for deploying STL applications was a system service, SIESTA, for verifying some properties of the application before giving it special privileges for handling data. What other system services could be developed to support STL applications, helping them to cooperate with systems security to enforce end-to-end security goals? One promising example of further exploration is to see how STL applications interact with the Asbestos operating system \cite{EKV05} that also uses a labeled approach. Another promising avenue of future research is to see how STL applications can be used to extend database security properties.

These considerations raise the question of co-specification of policy. Rather than separately defining OS policy and application policy, is it possible to co-specify these in a single high-level policy and automatically generate OS and application policies to implement it? Furthermore, to what extent can such policies be carried beyond a single OS and application and enforced over a distributed system? Some research is currently investigating this question, seeking to solve it through a shared reference monitor, or Shamon \cite{MBC06,JMC06}. Our research promises to harmonize quite well with the goals of the Shamon system.

Finally, for STL applications to extend their security properties to a mobile, distributed setting, it is important to fill in the holes we left in SIESTA, namely to be able to verify the “Jif-ness” of an application and to be able to determine the integrity of its information flow policy. A technology such as proof-carrying code (PCC) would be especially helpful here. This requires that the information flow guarantees of an STL be able to be pushed all the way into the bytecode (or typed assembly language) and that the proof generated by the type checker be attached to the low level code. Banerjee, Rezk and Naumann have made some progress towards this goal \cite{BRN06}. A full PCC framework for STLs would be a great benefit for extending STLs to produce provably secure, mobile code.

8.3.2 Separability

A driving design goal throughout this work has been to maintain separation of policy specification from mechanism. Seeking to lift all policy decisions out of code and into a separate policy file guided design decisions for the principal and runtime infrastructures. Overall this has been quite successful. Two significant areas remain for future work, however: policy management and policy analysis.

To handle policy management, a policy server and some model of policy administration are both necessary. A motivating force behind the decentralized label model (DLM) used by Jif is to handle mutually distrusting principals through a decentralized policy. A decentralized policy could be stored on a common server and each user could update his own policy, while possibly also maintaining some mandatory requirements.
There has been much work on policy administration models in general, and some consideration for STLs in particular, namely with the Rx system [SHTZ06]. Rx builds on the considerable work in policy administration from the RBAC community combined with the substantial developments for distributed policy in the Trust Management community by utilizing RT labels. Implementing the Rx system, complete with policy administration and even dynamic updating tools, as well as a centralized or distributed policy server framework is an important avenue of future work.

Some work on dynamic updating [HTHZ05, SHTZ06] exposes some problems in Jif’s flexible principal interface. Although it is not part of Jif’s normal principal interface, it would be possible to introduce delegation revocation as part of a user-defined principal. This is problematic, because it can introduce unsafe flows into an application. Consider the following code. The semantics of the actsfor check is that the then-branch of the conditional should only be executed so long as the check holds. In this case, the policy on data2 is that data1 could only flow into it so long as p actsfor q. In this case, revoking the relationship between p and q during the then-branch should cause the code to exit the branch before assigning data1 to data2. Jif does not implement this semantics, however.

Listing 8.1. Problematic dynamic updating.

```c
int{q} data1;
int{p} data2;
if (p actsfor q) {
    MyPrincipal qq = q;
    qq.revoke(p); //revokes delegation
    data2 = data1; // Jif allows this contrary to the revocation
}
```

Another important area for improving flexibility regards policy analysis. We provide a policy analysis for information flow compliance between JPolicy and SELinux policy, focusing on MLS labels. It is important carry out a more careful analysis of the interaction effects between Type Enforcement policy and the MLS policy in SELinux. As noted earlier, this interaction is limited to some very specific cases, but a combination of TE analysis with our MLS analysis would produce some important results for full SELinux system security management. Due to the similarity of the frameworks, combining our analysis with that of Sarna-Sota and Stoller [SSS04] should be particularly fruitful.

Also under policy analysis, a more careful analysis of the MLS policy is called for, in light of the special privileges for declassification that can be introduced for trusted subjects and trusted objects. These privileges include attributes in the existing MLS reference policy such as mlsfilereadtoclr and mlsfilewritetoclr, which introduce additional information flows.

Beyond policy analysis for compliance, other forms of policy reconciliation may be fruitful as well, such as those expressible in Ismene [MP00] in the Antigone framework. More experience implementing STL applications to co-operate with secure systems is necessary to understand the issues or directions of future analyses.
8.3.3 Improving modularity and practicality

While JPmail shows that it is possible to develop applications with practical functionality, JPmail is not yet a truly practical email client. A critical missing element is a graphical user interface (GUI). An area of future exploration involves handling GUIs, which requires handling event loops and multiple sources of input (keyboard inputs from many different fields, mouse inputs from mouse coordinates, button presses, etc.). Each input source must be labeled properly and event loops undoubtedly introduce subtle implicit flows between high-security events and low-security events. This is coupled with the fact that GUIs utilize multi-threading, a feature not yet implemented for STLs.

The lack of multi-threading is a major deficiency limiting the practicality of STL applications. The subtle information flows caused through synchronization and termination of threads have been widely studied in the literature, but there are no implementations as yet. This feature is critical for implementing many applications that could greatly benefit for STL information flow guarantees, such as secure servers. Although the SIF project [CVM07] was willing to accept the small information leaks caused by their multi-threaded event-handling system, this is not acceptable in general.

Some IDE tools are still needed for practical STL development. A crucial tool that is needed is a debugger. Some basic debugging tools should be routine, although, they must be extended for exploring labels, label relationships and label constraints as they develop dynamically in an application. Yet more powerful would be debugging tools that could maintain the confidentiality and integrity properties on data, while allowing the debugger only limited exploration of data, based on what it is authorized to see in a particular setting. Imagine an operative with secret clearance debugging an email server that also handles top secret communications.

Along with debugging there is a general need for better development tools to provide better guidance to programmers. This is general problem for languages that perform complicated static analyses to ensure strong program properties [Hee05, Wan86, JW86, BS93, FJKA06, FFK+96, DS06]. A corollary to Milner’s claim that, “well-typed expressions do not go wrong” is that it can be hard to make well-typed expressions. On a human level, it is frustrating when a program gives wrong results, but it is even more frustrating when a program will not even compile. One avenue for addressing this is in providing better error messages and suggestions for problem fixes. Similar to the ongoing development in other programming language communities, tools for better blaming must be developed to help programmers understand more precisely where code has gone astray.

We have presented compelling evidence that many labels can be inferred so long as principals, principal relationships and I/O labeling is handled by the application developer. Jif seeks to facilitate compositionality by requiring the programmer to place conservative approximations of a method’s information flows on the method’s header. In some ways, this is the worst of both worlds. It requires potentially complicated annotations from programmers while also limiting the precision of the annotations that may be provided. Better interprocedural label inference could ease the programmer’s burden, while also providing more precise labels for methods and classes. There are at least two starting points for this in the literature [SBN04, ST07].
The practicality of a Java-based language is certainly limited. To improve practicality, security-type annotations must be extended to other languages, like C, that have more low-level, precise control of the hardware and memory. The problem is that such languages tend not to be type-safe and thus preclude security-type checking as a means of verifying security properties. Incorporating security typing with some projects such as Cyclone [JMG+02], CCured [NCH+06], or CQual [FJKA06] may be effective for improving the practicality of STL technology.

Finally, practicality ultimately requires evaluation of technology and further guidance through building more applications, implementing more policies and also carrying out user studies. Understanding how difficult it is both to teach and to learn the reasoning necessary for information-flow restricted programming is essential for developing more practical STL tools and deploying them for the development of real systems.

### 8.3.4 Expressiveness

STLs are particularly effective for expressing and enforcing confidentiality and integrity properties. Many other security properties can be derived from these. A few areas are also lacking. Some security properties that cannot be expressed through strict noninterference are authentication, auditing, temporal properties, and cryptographic guarantees.

The typical approach to providing for new STL security properties is to introduce a new declassification concept [SS05]. Another avenue is by using more expressive or more specialized labels. For example, it may be beneficial to add label models that can express SELinux’s Type Enforcement or Domain and Type Enforcement, Java’s Stack Inspection, or other access control systems. Currently this would be a significant undertaking in Jif, because the label model is too deeply integrated into the compiler, but future STLs may use a more modular approach, allowing different label models to be plugged in for different environments.

A danger that must be avoided is making the labels or the declassification mechanisms so complex that it severely hinders usability. The DLM, for example, provides for reader/writer lists to enable more expressive policies. These were used to good advantage in SIF applications, but we found them complex to handle, difficult to reason about, and for our purposes, completely unnecessary in all our applications. It may be possible to harness the power of such complex and unwieldy labeling models by leveraging our work with Channels. In particular, by relegating the labeling and checking of labels to Channel code, the complicated labelings could be left to the specialists who design the Channel labeling interfaces. That way, the guarantees could be achieved, but the general-purpose programmer freed from the complicated details.

Having them does provides for very complicated security properties. One possibility is that, if the complex labels can be isolated to channels, as in the two SIF applications, complex labels may not be too burdensome for the programmer.

**Authentication** in an STL is the process of dynamically placing a label on some data. This finds a natural expression in our Channel interface. It may require various credentials, such as a password, a private key, or some quality in the data that is being labeled (in the FlowWall, the source address “authenticated” the security context of
the packet and determined what label should be placed on the packet). Authentication
decisions are policy decisions and so we facilitate their placement in an external policy
file. These decisions are not evident in the labeling scheme nor can they be checked
automatically by the Jif compiler. There’s no right or wrong way to authenticate data—it
is a policy decision that must be handled on a per-application basis.

More complex authentication protocols, such as those for a mail server cannot be
expressed as an information flow policy label. These interactive protocols may involve
challenge and response, nonces, hashed secrets, etc. The STL compiler is unable to guar-
antee that this has all been designed and carried out correctly. Some limited guarantees
can be made with regard to whether secrets have been leaked (the STL compiler might
catch that a password is being sent over an insecure channel, for example, without being
hashed first, as illustrated in JPmail in Chapter 3), but a more robust analysis of the
security of these protocols is beyond the scope of security-type checking. Our Channel
class was built to facilitate such protocols however. The Channel can be used to model
different stages of protocols by applying different labels to the channel. For example, in
a password authentication, after the user enters her username, the secrecy of the input
could be upgraded to protect the password that follows. The secrecy of the output chan-
nel, however, should not be upgraded until the password is successful. This models the
way passwords are normally “X”ed out as they are entered.

Another limitation of STL policies is in regard to availability. STLs are partic-

ularly effective for expressing what must not happen, but not necessarily for expressing
what must happen in an application. For example, we can show that FlowWall’s
implementation of its firewall policy never allows bad packets to get through. Merely
inspecting the policy does not, however, show that any packets come through at all.
Information flow control better expresses the prevention of flows than the certitude of
arrival (or availability of data). It can express that flows may occur, but not that they
must occur.

In addition to the problems with demonstrating availability, there is also a lim-

itation in demonstrating temporal properties such as ordering. Again considering the
firewall, a property that is often important for a firewall is the ordering of packets. That
FlowWall implements this property is not apparent from the high-level information
flow policy. We know that certain flows can never occur, but it is not clear in what order
they will occur without a careful inspection of the application code. For FlowWall we
can determine this more easily, because it is actually a property of the PacketChannel.
Because the PacketChannel is only willing to output a packet with the destination address
of the last input packet, it will ensure that only one packet is processed at a time. It is
possible to implement automatically checked ordering properties, but it requires declas-
sification and the properties become less obviously enforced. For example, SIF [CVM07]
uses complex owner/reader labels and mandatory declassifications to enforce some or-
dering properties.

STLs have been unable to reason about the security properties provided by crypt-
ographic constructions. This is not surprising because these properties generally rely on
the unproved existence of one-way functions. All the same, they are fundamental build-
ing blocks for secure systems. Thus far it has only been possible to capture cryptographic
guarantees as exceptions by wrapping them in some form of declassification. Integrating
the guarantees provided by various shared-key, asymmetric key, hash-based, HMAC-based and other cryptographic constructions is a rich area of ongoing exploration.

8.4 From Theory to Practice

The Rule of Saint Benedict was the confluence of two separate fonts: the wisdom of the masters who preceded him and his own lived experience. Saint Benedict drew on at least two hundred and fifty years of monastic wisdom when he composed his “little rule written for beginners,” and yet he did not stop with a theoretical transcription of past wisdom. To the contrary, he shaped his policy system also according to his own experience. It is notable that this experience was particularly hard-won: his first monastic community attempted to poison him for his overly strict policies [tG49]. All the same, this experience appears to have taught him the moderation, compassion, practicality and flexibility for which he, and his rule, are so well revered.

To a much lesser, but still significant, extent, this work, too, combines the wisdom of the past with the lessons of hard-won experience. I believe strongly in the rich interplay between the ideal and the real, between theory and practice. The way that the ideal can set a course and propose a pristine vision of what should be provides hopeful energy and clear direction for the practitioner. The practitioner, in turn, provides essential data that must be incorporated into any ideal model that ultimately seeks the truth. Whether in matters monastic or matters computational, the practitioner must learn to resist disillusion, always willing to follow the vision of the ideal. The idealist, likewise, must never lose a firm commitment to reality, always seeking to anchor the ideal in fact. Both must be willing to suffer a bit, however, because “the ideal and the real only meet on the Cross.” (Fr. Silvan Rouse, CP, personal communication)

In this work, we have put theory into practice by using security-typed languages to build secure systems. That experience revealed the shortcomings of these powerful theoretical tools and inspired new infrastructure, policy systems, tools and analyses to fill the gap. Perhaps most importantly, it added a font of experience to the font of traditional wisdom. We hope the experience and the concrete results of this thesis will ultimately advance the cause of computer security, and in so doing will serve “to amend faults and to safeguard love.”
Bibliography


Vita

Fr. Boniface Patrick Hicks, OSB

Education


Penn State University, University Park, Pennsylvania. M.Sc. in Computer Science and Engineering, 1997; B.Sc. in Computer Science with Honors cum laude, minors in Mathematics, Statistics and German, 1997.

Professional Experience


Campus minister, Penn State Catholic Campus Ministry, Penn State, University Park, Pennsylvania. Fall 2004–present.


Teaching Assistant, Department of Computer Science and Engineering, Penn State, University Park, Pennsylvania. Fall 1997.

Courses Taught (at Saint Vincent College, unless otherwise noted)

CMPSC 201, Computer Programming for Engineers. TA (conducted lectures for recitation sections). Department of Computer Science and Engineering, Penn State. Fall 1997.