Minimizing Resource Access and Management Disparities Between Desktop and Web Applications

by

Brett Allen Cannon

B. Philosophy, University of California, Berkeley, 2002
M. Computer Science, California Polytechnic State University, San Luis Obispo, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

The University Of British Columbia

(Vancouver)

January 2011

© Brett Allen Cannon, 2011
Abstract

Client applications, including both traditional desktop applications and modern Web applications, typically access and manage resources in order to perform their intended work. Unfortunately both approaches lack something the other has when it comes to resource access and management. Desktop applications typically do not provide enough security for resources to prevent malicious abuse of them. While operating systems provide access-control lists, they do not help facilitate enforcing the Principle of Least Privilege to minimize insecure resource access. Web applications, while they operate in a sandboxed environment which provides the necessary resource access restrictions, do not have a rich API for data storage management. While HTML5 provides two primitive APIs for a developer to use to manage stored data, neither approach allows for storing data in an object-oriented manner that a developer is used to. This thesis addresses the question of "can these two shortcomings in resource access and management be overcome in order to lessen the technological gap between desktop applications and Web applications?" For desktop applications an approach using aspect-oriented software design has been created which adds enforcement of the Principle of Least Privilege by using I/O to dynamically choose what resource permissions to grant the application. I show that this approach can tie into Java applications with rich user interaction and I/O to control resource access while providing a way for third-parties to provide the security code for auditing purposes. For Web applications, a library has been designed which introduces automatic object persistence to JavaScript without requiring any modifications of the browser. I show that my library is able to persist a thousand objects without a user-perceptible impact on application performance, all while having minimal requirements placed upon the developer to use the library.
Preface

This thesis draws heavily from two papers I have previously published:


Both papers are under the copyright of the ACM and for which I have permission to reuse.

I have changed the writing style to use a first-person voice even though both papers were co-authored with my supervisor, Eric Wohlstadter. The ideas conveyed in this thesis are of my own creation, as is all of the code written and discussed. Eric’s contributions to the two papers were that of guidance, editing, and help with refining the ideas presented.
# Table of Contents

Abstract ................................................................. ii

Preface ................................................................. iii

Table of Contents ....................................................... iv

List of Tables ........................................................... vii

List of Figures ........................................................... viii

Glossary ................................................................. x

Acknowledgments ........................................................ xii

Dedication ............................................................... xiii

1 Introduction ........................................................... 1
   1.1 Secure Resource Access for Desktop Applications .......... 3
       1.1.1 The Problem ............................................. 6
       1.1.2 The Solution ............................................. 6
   1.2 Simplification of the Storage Resource for Web Applications . 10
       1.2.1 The Problem ............................................. 11
       1.2.2 The Solution ............................................. 12
   1.3 Contributions .................................................. 15
   1.4 Thesis Statement .............................................. 16
   1.5 Outline ......................................................... 17
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Harnessing User Actions to Control Resource Access</td>
<td>18</td>
</tr>
<tr>
<td>2.1</td>
<td>Motivating Example</td>
<td>19</td>
</tr>
<tr>
<td>2.1.1</td>
<td>GUI I/O</td>
<td>19</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Network and Disk I/O</td>
<td>21</td>
</tr>
<tr>
<td>2.1.3</td>
<td>User Prompting</td>
<td>22</td>
</tr>
<tr>
<td>2.2</td>
<td>Architecture Details</td>
<td>22</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Developer Roles</td>
<td>23</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Runtime Architecture</td>
<td>27</td>
</tr>
<tr>
<td>2.3</td>
<td>Implementation Details</td>
<td>34</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Network and Disk I/O</td>
<td>34</td>
</tr>
<tr>
<td>2.3.2</td>
<td>GUI I/O</td>
<td>43</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Deletion</td>
<td>51</td>
</tr>
<tr>
<td>2.4</td>
<td>Soundness</td>
<td>52</td>
</tr>
<tr>
<td>2.5</td>
<td>Case Study</td>
<td>54</td>
</tr>
<tr>
<td>2.5.1</td>
<td>RSSOwl</td>
<td>55</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Lobo</td>
<td>60</td>
</tr>
<tr>
<td>2.5.3</td>
<td>Performance</td>
<td>64</td>
</tr>
<tr>
<td>2.5.4</td>
<td>Summary</td>
<td>65</td>
</tr>
<tr>
<td>3</td>
<td>Automatic Object Persistence in JavaScript</td>
<td>66</td>
</tr>
<tr>
<td>3.1</td>
<td>Overview</td>
<td>69</td>
</tr>
<tr>
<td>3.2</td>
<td>Detecting Mutations</td>
<td>70</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Detecting New Objects by Reachability in JavaScript</td>
<td>71</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Detecting Dynamic Property Additions</td>
<td>72</td>
</tr>
<tr>
<td>3.3</td>
<td>Persisting Data Locally</td>
<td>74</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Garbage Collection</td>
<td>76</td>
</tr>
<tr>
<td>3.4</td>
<td>Persisting Data Remotely</td>
<td>77</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Browser-Side Logging</td>
<td>78</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Synchronization With A Server</td>
<td>79</td>
</tr>
<tr>
<td>3.5</td>
<td>Correctness</td>
<td>87</td>
</tr>
<tr>
<td>3.6</td>
<td>Evaluation</td>
<td>88</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Benchmarks</td>
<td>88</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Experience</td>
<td>99</td>
</tr>
</tbody>
</table>
4 Related Work .................................................. 102
   4.1 Having User Actions Control Resource Access ............... 102
   4.2 Automatic Object Persistence in JavaScript .................. 108

5 Conclusions .................................................. 113
   5.1 Unsolved Problems Between Desktop and Web Applications ... 115
   5.2 Future Work .............................................. 116

Bibliography ................................................... 119

Appendices .................................................... 131

Appendix A Manual Persistence Version of Moi .................... 132
Appendix B Automated Object Persistence Version of Moi ........ 142
List of Tables

2.1 Mechanisms used to support RSSOwl security. © ACM 2009 . . . . . . 56
2.2 Mechanisms used to support Lobo security. © ACM 2009 . . . . . . 61
2.3 Overhead in RSSOwl from parsing feeds for URIs. © ACM 2009 . . . 63
List of Figures

1.1 An example aspect which uses a pointcut to capture the join point where the main method of a Java program is called to dynamically grant access to the application for the files specified on the command line .................................................. 7

1.2 The two available approaches to storing data in browsers and my proposed approach, using the storage of contact information in an e-mail client as an example. ................................................. 13

2.1 Subscribing to a feed in RSSOwl ................................................... 20
2.2 A link element from RSSOwl’s configuration file ............................ 21
2.3 Dependency diagram of developer roles; developers depend on the person below them in the stack ................................................. 23
2.4 Architecture of mechanism to monitor resource access for a desktop application ................................................................. 29
2.5 XPath code policy rules for RSSOwl ............................................. 35
2.6 Grammar for XPath rules ............................................................. 36
2.7 The wrap method, written by a policy expert, is used in authority aspects to begin monitoring some point of I/O ................................ 39
2.8 HttpURLConnectionAspect network stream monitoring aspect written by a framework expert ................................................. 41
2.9 FileStreamAspect file stream monitoring aspect written by a framework expert ............................................................. 44
2.10 Abstract class for tracking text entered into the GUI, written by a policy expert .............................................................. 46
2.11 Abstract class for managing indirect GUI input, written by a policy
expert ......................................................... 49
2.12 Implementation of IndirectGUIAspect for SWT’s List widget 50
2.13 Implementation of DeletionAspect .......................... 52

3.1 Architecture diagram for automating the persistence of JavaScript
objects ......................................................... 69
3.2 Pseudocode for detecting new properties and persisting any new
parts of the object graph. ............................... 73
3.3 Pseudocode for the garbage collection step, using set notation . . 77
3.4 Pseudo-code for hot synchronization between the browser and the
server (note on syntax: [x, y] represents creating a list consisting
of x and y; for(x, y in z) represents iterating through
z which is a sequence of two-item lists) .................... 82
3.5 Pseudo-code for cold synchronization between the browser and the
server ....................................................... 85
3.6 Benchmarking the persistence of 200 new objects ............... 90
3.7 Benchmarking the persistence of 1000 objects ................. 91
3.8 Benchmarking the persistence of 5000 objects ............... 91
3.9 Benchmarking the stabilization of 200 objects. ............... 93
3.10 Benchmarking the stabilization of 1000 objects. ........... 94
3.11 Benchmarking the stabilization of 5000 objects. ........... 94
3.12 Benchmarking the garbage collection overhead of not collecting
200 objects. .................................................. 95
3.13 Benchmarking the garbage collection overhead of not collecting
1000 objects. .............................................. 96
3.14 Benchmarking the garbage collection overhead of not collecting
5000 objects. .............................................. 96
3.15 Benchmarking the use of the eBay Web service for 100 auction
results which are stored to local storage and converted to XML... 98
Glossary

**ADVICE**  A pointcut along with code to execute when the pointcut matches a join point

**AOP**  Aspect-Oriented Programming

**AOSD**  Aspect-Oriented Software Development

**ASPECT**  A collection of advice

**ASPECTJ**  An aspect-oriented programming language designed to work with Java

**AUTHORITY ASPECT**  Aspects which manage granting permissions to the running program

**AUTHORIZING CONTEXT**  In a running program, a context where input has resources identified in it which may require authority to access

**CROSS-CUTTING CONCERN**  An aspect of a program that exists throughout a program beyond the typical compartmentalization of code such as classes or modules

**CSS**  Cascading Style Sheet

**DOS**  Denial of Service

**DSL**  Domain-Specific Language

**GUID**  Globally Unique Identifier

**JDK**  Java Development Kit
JOIN POINT  A point in the execution of a program where a pointcut can match

JSON  JavaScript Object Notation

JVM  Java Virtual Machine

MIME  Multipurpose Internet Mail Extensions

OPML  Outline Processor Markup Language

ORM  Object–Relational Mapping

POINTERCUT  Specification of execution points in a program

POLP  Principle of Least Privilege

REST  Representational State Transfer

RSS  Really Simple Syndication

SLOC  Source Lines Of Code

SSB  Site-Specific Browser

SWT  Standard Widget Toolkit

URI  Uniform Resource Identifiers

WAN  Wide Area Network

XHR  XMLHttpRequest
Acknowledgments

Various people have helped me academically during my Ph.D. At UBC, the members of the Software Practices Lab have tolerated my occasional outbursts of frustration and helped in providing feedback in practice talks. My supervisory committee — Norm Hutchinson and Rachel Pottinger — provided invaluable feedback on earlier drafts of this thesis. My supervisor, Eric Wohlstadter, provided guidance from beginning to end on not only this thesis but the publications that this thesis stems from.

Outside of UBC, several groups deserve thanks. The POLA community, especially those from HP, helped me learn a great deal about software security. My various intern hosts at Google — Dan Bentley, Neal Norwitz, and Guido van Rossum — helped fund my education through my internships as well as providing breaks from my academic work. Both Aaron Keen and Michael Haungs of Cal Poly SLO deserve special thanks for continually encouraging me to continue on for my Ph.D while working towards my masters degree.

Finally, the group that has provided the most support over the longest term in my goal of attaining my Ph.D. is that of the Python community as a whole. The support of the community through feedback on my research and writings has greatly helped. The Python core development team deserves special recognition having taught me how to develop world-class software while leading to numerous friendships.
Dedication

I have been very fortunate to have a large number of people support me and make me a happy individual throughout my life. Thanks to the wonderful teachers I have had over my lifetime who have gotten me here. Thanks to all of my friends I have made from my extensive years in school: Bryan, Phil, Steve, Wilson, (An)drew, Anna, Bem, Channing, Debbie, Gabe, Gary, Henry, Jackie, Jeff, Jer, John, Ken, Randy, Sandy, Steve, Susie, Carol, Hilary, Matt, Shaw, Alex, Andrew, Arjun, Brad, Brian, Davey, Disha, Emerson, Heather, Meesh, Mike, Paul, Richard, and Thomas (plus everyone’s spouses). Thanks to the friends I made at Google during my internships and have managed to keep in touch: Alon, Ann, Brett, Damon, DeWitt, Joey, John, Rafe, Tommy, and Will. Thanks to all of the friends from the Python community who have listened to me ramble on about whatever over the years and make PyCon fun: Alex, Andrew, Anna, Antoine, Barry, Collin, David, Dino, Doug, Eric, Frank, Fred, Gregory, Georg, Guido, Ivan, Jack, Jacob, James B., James T., Jeffrey, Jeremy, Jerry, Jesse, Martin, Michael, Neal, Phil, R. David, Raymond, Richard, Steve H., Steven B., Tarek, Terry, Thomas, Tim, Titus, and Van (plus everyone I forgot to list). Thanks to Eric for working with me instead of against me as my supervisor and tolerating my occasional ranting against the academic establishment.

My family deserves a lot of thanks as they have been there since the beginning: The Moranns (including Aunt Carmel), Alexandra, Todd & Whitney, Dad & Richard, and Mom.

And an exuberant thanks to my wife, Andrea. She loved me (and allowed me to love her) while experiencing the “joys” of living with me as I wrote this thesis and finished off my PhD.
Chapter 1

Introduction

Developing a successful client computer application requires various pieces of technology. One of those pieces of technology that is common to client applications is the need to use resources. In a computer program, a resource is an asset that has a physical manifestation to it, e.g., storage, network connections, etc. Whether it is access to a data resource such as the file system to store data between executions or a network resource to communicate with a server, a client application needs to access resources in order to operate. That access needs to be done in a secure fashion so that the user’s machine does not become infected with malicious code or accidentally perform an action it should not be allowed to do on a resource, whether from user or developer error. Using resources also needs to be simple and flexible, else the complexity or rigidity of the API for resource access will become a development burden for application developers.

There are roughly two ways to develop a client application: as a (traditional) desktop application and as a Web application. In the first case, the operating system provides UI toolkits so developers have resources to allow interactions between the user and the application. The local file system gives the application a resource for data persistence so that work can be paused between executions. A network resource is provided so that online communication with a server is possible. All of these resources are typically easily usable thanks to abstractions being developed over the decades by operating system vendors. But while the operating system also provides rudimentary security to control resource access, desktop applications
on their own have typically not implemented the additional security needed to be trustworthy on their own. This has led to the point that desktop applications have ended up with more reported security vulnerabilities than operating systems [8].

Another approach to creating an application is as a Web application. The browser provides a stable run-time which exposes the various resources a Web application might need in a secure fashion thanks to the same-origin policy that browsers enforce upon Web pages and their associated JavaScript code. This policy makes sure that a Web page can only make dynamic connections to the same domain that the page originated from. HTML provides the GUI in a way that works across browsers. An API called XMLHttpRequest (XHR) provides a network resource to communicate with a server on the Internet. High availability of the application is attainable by allowing for offline access to Web applications through the HTML5 application caching specification [49]. But even though HTML5 introduces browser-side storage (both key/value and SQL-based), it is not as inherently flexible as an actual file system which is available for desktop applications.

Until fairly recently, even with the typical weakness in the security provided for resources, desktop applications were considered technologically superior over similar Web applications unless deployment across nascent platforms was required. But within just the last couple of years, a set of technologies have gathered to allow Web applications to come within striking distance of a large set of application domains traditionally reserved for desktop applications. Much of this is thanks to focused efforts to improve the performance of JavaScript along with the new resources being provided by HTML5. Web applications have improved so much that Site-Specific Browser (SSB) applications now exist that allow a user to create a desktop shortcut that launches a browser instance for a single website, hiding all generic browser UI elements to make the Web application appear like a desktop application. It has reached the point that three of the four top browsers have SSB capabilities either built-in[1] provided through a browser extension[2] or provided through a stand-alone desktop application[3].

With the technological gap between desktop applications and Web applications

closing in regards to resources, the shortcomings of desktop applications become more glaring. But I would argue that neither the shortcomings for Web applications nor for desktop applications are insurmountable. This thesis will consider the following two shortcomings and investigate whether they can be overcome in ways that do not introduce new burdens upon developers:

1. For desktop applications, can resource access be secured at the application level such that the application cannot overstep reasonable bounds for what resources it should have access to?

2. For Web applications, can storage resources such as those introduced by HTML5 be simplified in order to make data persistence easier to use?

The research question to be answered is whether the differences in resource access and management between desktop and Web applications can be removed such that resources no longer act as a reason to choose to develop an application as a desktop or Web application.

1.1 Secure Resource Access for Desktop Applications

Desktop applications are a popular form of software which interact heavily with both local and remote resources, such as local disk and network resources. This is both a benefit in terms of the rich features desktop applications can provide, but also a potential security risk. Resources must be handled by the application appropriately to ensure confidentiality and integrity of a user’s sensitive information. Unfortunately the default security policy of applications is typically not tight enough to prevent misuse of resources. Creating a way to introduce a security policy to manage resource accesses that cuts across an entire application would be beneficial in tightening security for applications.

Aspect-Oriented Software Development (AOSD) can be used to address security concerns in software in such a modular fashion [55]. Briefly, AOSD allows a developer to (conditionally) inject code into an application at selected points without modifying the original points being affected based on some aspect or concern of the application. The points where you can inject code are called join points, such as at every method call, variable assignment, etc. The specification of where
to inject code is a pointcut (a pointcut can be seen in Figure 1.1 in lines 7-9). An advice is a pointcut paired with code to execute when the pointcut matches a join point (lines 15-21). An aspect is a collection of pointcuts and advice (all of Figure 1.1). This leads to Aspect-Oriented Programming (AOP), using languages such as AspectJ (what Figure 1.1 is written in) to introduce AOSD into languages such as Java. All of this allows code to be defined that accesses a myriad of locations in the original code; such code is known as a cross-cutting concern as it cuts across the entire application instead of being restricted to only a specific module or class. All of this allows AOSD to introduce an event-driven architecture so that, when a pointcut matches a join point, an advice is fired and executed.

The canonical example of using AOSD is to add logging to an application. A pointcut is written which matches all join points where an action which is to be logged occurs, e.g., all method calls. The pointcut is then used with an advice which introspects on what method call the pointcut matched. Knowing what method was called with what arguments, the advice can then call the logging library appropriately. This provides a separation of concerns by having logging code which applies across the application separated from the application itself, cross-cutting the application where logging is expected to take place.

The benefits of AOSD for security work are numerous. By factoring out security code into aspects it becomes centralized. This makes it easier for an expert to perform security audits on the code. It also makes it easier to maintain. Perhaps most importantly, AOSD allows security code to be automatically applied consistently throughout an application based on pointcuts. This allows a developer to make statements about what should and should not be allowed in a single, cohesive manner, e.g., all file accesses must first be cleared through a security manager to make sure no improper file access occurs.

Popular middleware platforms (e.g., the Spring Framework\(^4\)) commonly support mechanisms such as role-based access control and encryption using pointcut-based deployment. Other research has proposed even more advanced scenarios where AOSD is applied to address security [23, 36, 43, 51, 67, 71, 79, 99, 100, 103]. However, most existing tools and research focus on the protection of server-side re-

\(^4\)http://www.springsource.org/
sources. There is little existing work which focuses specifically on the protection and privacy of end-user client-side resources. This is important as a report has identified “a significant growth in the number of client-side vulnerabilities ... [in] desktop applications” [8].

Resource misuse can arise in desktop applications in several ways. First, a virus, worm, or other kind of malware could introduce malicious code. This code may, for instance, try to open a back-channel such as connecting to a server to report on a user’s usage habits or to reveal sensitive information. Alternatively, it may simply attempt to corrupt or destroy a user’s files. A plug-in might be used by the application which is not as well developed as the application itself, such as the Adobe Flash plug-in for Web browsers in which security holes have been reported (e.g., Technical Cyber Security Alert TA09-204A\(^5\)). The plug-in could easily misbehave and accidentally delete or improperly write files. Finally, an application developer could be malicious or sloppy in the use of resources, especially if they are not security experts.

A common methodology to mitigate resource misuse is to follow the Principle of Least Privilege (POLP): “Every program ... should operate using the least set of privileges necessary to complete the job” [81]. This could be done by sandboxing a desktop application and its resource accesses; then the application could only use those resources necessary for its operation. In the case of a security issue, any negative effects would be limited to only those resources the application needed access to and not any arbitrary resource.

The traditional approach for protecting client resources is to use code access security [58]. Platforms such as Java and .NET allow each application to be controlled by a policy that details which resources code from various components (e.g., a “jar” file) are allowed to access. Essentially, the resource access concern is separated from the code into a policy. This provides the modularity necessary for system administrators and (security-savvy) users to control and reason about resource access for third-party applications.

\(^5\)http://www.us-cert.gov/cas/techalerts/TA09-204A.html
1.1.1 The Problem

Unfortunately it is just not possible to know statically what resources will be accessed by an application at run-time. This is because resource identity can depend on dynamic program I/O. The needs of an application, from a resource perspective, can also vary between executions in a way that cannot be predicted. So to cover all possibilities, many applications are simply given access to all of a user’s resources. This is obviously not an option for ensuring a high level of trust in a deployed application [58, 65, 81].

For example, as noted by Wagner [96], the UNIX program `cp` (i.e., copy) should only have access to the files specified on the command line to read from and write to. However, in its common implementation, `cp` inherits all the privileges of the current user, giving it access to more than just the two files it should be restricted to working with. To avoid this, it would seem reasonable to sandbox `cp` based on its I/O so that the file paths given to the program are the only resources `cp` has access to. Making security decisions based on the interaction of the user with an application is known as the Principle of Explicit Authorization [102]. A shortcoming of the principle, however, is that it only takes into account explicit actions performed by the user, instead of the overall intent of what the user wants. Consider the example of downloading and rendering a Web page. When a user clicks a link to view the Web page, this action implies “download the HTML for the Web page” while the user’s intent suggests “download the Web page and all resource dependencies in order to render the Web page properly”. This thesis attempts to address this disconnect between a user’s action and their intent.

1.1.2 The Solution

Notice that to address the example of the `cp` program, a pointcut could be used to identify the join point where command-line arguments are passed to the application from the console. Then an aspect could be written which dynamically grants permissions to access only those files specified on the command line; Figure 1.1 shows such an aspect. Thus an advice could grant the permissions necessary to access the files specified on the command line for `cp` dynamically. If you combine the advice with `cp`, initially executing with no permissions to access any files and
public aspect ReadCommandLine {

    /*
     * Capture the join point where the main() method is implicitly called at
     * the start of execution.
     */
    private pointcut commandLine:
        execution(void main(String[])) &&
        !cflow(call(* *.*(..)));

    /*
     * An advice which executes code when the commandLine pointcut matches the
     * join point for the execution of the main() method.
     */
    void around(String[] cmdArgs)
        commandLine() && args(cmdArgs) {
            /*
             * Grant access to files specified on the command line as found in
             * cmdArgs...
             */
            }
}

Figure 1.1: An example aspect which uses a pointcut to capture the join point
where the main method of a Java program is called to dynamically
grant access to the application for the files specified on the command
line

adding the ones necessary at run–time, you have caused cp to execute with the
least permissions needed and thus met the requirements for the POLP.

The concept of using pointcuts to enforce the POLP also allows extending the
Principle of Explicit Authorization take a user’s intent into consideration and not
just the explicit action. Returning to the example given in Section 1.1.1 of when
a user downloads an HTML Web page to be downloaded; here, pointcuts can be
used to detect when the HTML is downloaded by the application. Advice can then
be utilized to parse the downloaded HTML for references to dependent resources.
Subsequently, permission can be granted for accessing the resource dependencies
implicitly so that the users’ intent as started by triggering the download of HTML
can be met by the application.

Based on these motivations, I have devised a more dynamic form of code ac-
access security than typically used by desktop applications by leveraging AOSD. Us-
ing this approach, an application is monitored at run-time by what I call authority
aspects. These aspect components contain code which are given permission delegation rights to dynamically grant new permissions to the base application components through API calls to the Java security manager when certain pre-conditions are met. Authority aspects are then guided by an extended form of code-access policy to allow the dynamic granting of permissions to the application based on I/O as the application runs as an extension of the Principle of Explicit Authorization. The use of (I/O) monitoring allows for the user’s intentionality of what permissions should be granted to be inferred to drive security decisions. Although the application may be written by an untrusted party, these aspects could be implemented by a trusted third-party such as a middleware vendor, virtual machine vendor, or open-source consortium and woven (i.e., merged) into the application by the application user themselves to allow them control over the security policy that controls the application.

For this approach to be used on an application, an end-user would obtain authority aspects which are specific to some I/O class library frameworks and possibly the application itself. These I/O frameworks include implementations of GUI widgets (such as Swing or SWT) and implementations of application protocols (such as HTTP protocol libraries). Then an end-user can make use of applications that rely on these frameworks for I/O. In this way, the aspects will serve as a neutral third-party to bridge a trust gap between the user and the desktop application developer. The approach is not intended as a generic application sandbox which can be used to retrofit arbitrary applications, as some knowledge of what toolkits are used along with how the UI interacts with code is typically needed for extensive security support, but it should be possible to add in security to a desktop application without having to modify the application’s own code directly. This introduces a new architecture for security where various stakeholders at different levels of trust are allowed to participate in the security process [Section 2.2.1].

Instead of listing static resource identifiers in a policy, policies should be able to list authorizing contexts. An authorizing context identifies black-box inputs to an application (e.g., file, network, or GUI input) and properties of data (e.g., XML)

---

6I use the generic term “context” rather than the specific term “join point” because sometimes properties of data (e.g., (XML)) need to be accounted for, in addition to program execution points as supported by AspectJ.
tags) at those inputs as resources which serve references to other resources (e.g., URIs) which require authority to be accessed. Consider img tags in an HTML file downloaded over the network. An authorizing context can be used to automatically grant permission to download the images as specified by the img tags so that the page can render properly. This is done without a user having to specifically clear the images for download as the act of downloading the HTML page implicitly grants permission to fetch the images. The Principle of Explicit Authorization, while using a user's actions to drive security decisions, does not extend beyond the explicit resource the user is accessing. The addition of authorizing contexts extends this to the resource dependencies that are implied by the user. So while the Principle of Explicit Authorization would grant access to download the HyperText Markup Language (HTML) file, the resource dependency extension introduced by this work also grants the permission to download the embedded img tags in the HTML. Other researchers have cited my work as a way to blend security into an application without adversely affecting the GUI thanks to these authorizing contexts [54].

Authorizing contexts are identified through pointcuts and XPath\(^\text{7}\) rules. These rules provide a declarative way for users to describe which Uniform Resource Identifiers (URI)\(^\text{8}\) should be accessible to the application. For example, a rule such as /html/body//img/@src would provide the application permission to download HTML images. Authorizing contexts are specified separately from the application source code, so it possible to separately audit what kinds of resources the application will be able to access. Aspects can then be written to read the policy and dynamically enforce it.

Based on these concepts I have implemented a framework of AspectJ abstract aspects to help enforce the POLP through an extension of the Principle of Explicit Authorization. An abstract aspect is much like an abstract class; an aspect meant to be subclassed and specialized as needed. The abstract aspects can be specialized for distinct application requirements where necessary. The framework also comes with several concrete aspects to support common scenarios. Currently there is built-in support for common usages of GUI toolkits (SWT and Swing), protocols

\(^{7}\)XPath is a file-path like notation for identifying elements of XML documents.

\(^{8}\)A W3C standard for referencing resources, including file paths, URLs, etc.
(HTTP) and formats (XML/HTML). This allows for a desktop application to be secured by using both aspects designed to work with pre-existing libraries as well as application-specific aspects.

Both the abstract aspects and their subclasses are provided to a user as source code. The user is then able to audit what a developer decided statically and what occurs dynamically in order to decide whether the application is behaving properly in terms of resource access. Thus the security framework can be used to provide peace of mind to the user that the application uses resources properly based on I/O, all without requiring the user to actually trust the various providers of the aspects.

The goal of this part of my work is to see if it is possible to add security for resources to desktop applications in a way that is not prohibitive to either the developer or the user. By separating out the security concern into an aspects rather than having it embedded in an application, a user is provided with a way to verify and audit that an application behaves as it should from a permissions perspective. Using AOSD practices also benefits the developer by allowing security-specific code to be cleanly separated from an application’s main code, such that a trusted third-party can provide the security code for an application if needed. This separation allows for an architecture where stakeholders at varying levels of trust can contribute to the security infrastructure of an application. It also builds on the Principle of Explicit Authorization so that a user’s intent — as shown through I/O — not only drives security decisions for the resource that is directly accessed but also for implied resource dependencies. All of this should allow desktop applications — which typically have access to all the resources that the user has — to gain a level of security which more closely meets the security guarantees that a typical Web application provides — access to only the resources for the domain the Web application originates from — on a user’s computer.

1.2 Simplification of the Storage Resource for Web Applications

Although most browsers have historically had the capability to cache some components of a Web page [34], such as HTML and media resources, this has not included the ability to store the intermediate results of computations. As the desire to store
data that persists between executions has grown to become a more integral part of application functionality, this deficiency has made the deployment of complex, feature-rich, Web applications difficult, especially when there is a desire for the Web application to work offline.

New features introduced as part of the HTML5 standard [49] attempt to address the historical shortcoming of Web applications not being able to run without an Internet connection or store data locally for future use. Offline Web applications are a new class of application that are now possible thanks to HTML5 which can support page browsing and execution of scripts when disconnected from the Internet. Some popular examples of this new style of application include Gmail, WordPress, MySpace and Zoho.

Offline Web applications allow for data created and managed by Web applications to be stored in a resource locally in the browser. This allows Web application processing to be suspended and resumed offline, and also for offline data to be synchronized on the Web at some time in the future once an Internet connection is re-established. However, unlike the automatic caching of page components by Web browsers, the persistence of intermediate computations unfortunately requires careful consideration and manual intervention by Web application developers.

1.2.1 The Problem

Historically in non–browser environments, distributed programming language environments [14, 60] have offered automated object persistence to shield the developer from complexities related to data persistence. The automated storage of data objects is beneficial over the typical method used by desktop applications of manually reading and writing data to/from the file system thanks to its brevity and transparency. Unfortunately JavaScript does not inherently have a file system storage mechanism nor an object persistence mechanism; HTML5 provides a key/value store [48] and a SQL-based database [47]. While a key/value store provides a very simple but useful resource for storing data – the usefulness of hash tables

9 http://www.gmail.com/
10 http://www.wordpress.com/
11 http://www.myspace.com/
12 http://www.zoho.com/
in programming demonstrates this – it is limited as it has no inherent structure or hierarchy to it. On the other hand SQL, while being very powerful, does not present the same level of simplicity as working with objects does. The existence of Object–Relational Mapping (ORM) libraries that hide the details of SQL databases behind an object facade is evidence that relational tables are not the preferred data abstraction for some situations.

Consider managing contacts in an e-mail Web application. Each contact needs to have at least a name and e-mail address stored. This data also needs to co-exist with other data such as what e-mail a user has in each inbox. In a key/value store, a developer would most likely either have a “contacts” key that stored a serialized version of all contact data or create a hierarchical key format which concatenated the term “contacts” with a contact’s name. For a SQL database, a contacts table would be created with at least a column for the name of a contact along with another column for the e-mail address along with a proper **INSERT** command to commit the data to the database.

### 1.2.2 The Solution

The two available approaches to storing data in the browser both require mapping a contact data object that the developer has already designed and used in the e-mail client to the key/value store or SQL database table. An automated object persistence approach would provide the easiest interface to persisting hierarchical data by removing the need for the developer to design such a mapping. Figure 1.2 shows how much more work is involved for a developer to store a simple data object in both a key/value store (7 SLOC of JavaScript as shown by lines 8-14) and a SQL database (20 SLOC shown by lines 21-41) compared to using an automated object persistence approach (3 SLOC shown by lines 48-50).

There is also the issue of interoperability. By externalizing data management into a library, browser incompatibilities can be hidden from the developer compared to working with the raw storage resource. Even for something as simple as a key/value store there are differences between browsers.\(^\text{13}\)

---

\(^\text{13}\)The realStorage project ([http://realstorage.googlecode.com](http://realstorage.googlecode.com)) was created by me for my thesis work specifically to work around incompatibilities amongst the various browsers implementing the HTML5 Web Storage specification [48].
var myContact = {name: "Brett Cannon", email: "brett@python.org"};

Storing into a key/value store.

-- Requires serializing the data object. */
var allContact = localStorage["contacts"]; if (allContact == undefined)
allContact = {};
else
allContact = JSON.parse(allContact); // De-serialize
allContact[myContact.name] = myContact;
localStorage["contacts"] = JSON.stringify(allContact); // Serialize

Storing into a SQL database table.

-- Includes creation of the table if needed. */
function changeVersion (db) {
  db.changeVersion("", "1.0", function (t) {
    t.executeSql("CREATE TABLE Contacts " +
"(contactId INTEGER PRIMARY KEY, name, email)")}, error)
}

function prepareDatabase (ready, error) {
  return openDatabase("contacts", "1.0", "contacts storage", 5*1024*1024,
  changeVersion);
}

function insertContact (db, contact) {
  db.transaction(function (t) {
    t.executeSql("INSERT INTO Contacts (name, email) VALUES (?, ?)",
[contact.name, contact.email], function (t, r) {/** Success */},
  function (t, e) {/** Error */});
})

prepareDatabase(
  function (db) {insertContact(db, myContact)}, function (e) {/** Error */
);

Storing using an object persistence approach.

-- Includes checking that the root contacts object exists. */
if (persist.root.contacts == undefined)
persist.root.contacts = {};
persist.root.contacts[myContact.name] = myContact;

**Figure 1.2:** The two available approaches to storing data in browsers and my proposed approach, using the storage of contact information in an e-mail client as an example.
For performance, direct access to the storage resource will always be faster than having to pass through a library. This is especially true if you must perform some query that the library does not natively support compared to some other data store, e.g., a SQL query as supported by a SQL database. I have investigated whether the performance of an automated object persistence approach could be reasonable.

Data synchronization with a server also needs to be considered. Being able to store data remotely is a key strength of Web applications and so cannot be ignored. An automated object persistence library could automate the communication so that the developer does not need to worry about such details. It would require not only tracking what objects have been mutated within the browser, but also detecting when such a mutation has occurred. Then the state of the browser must be synchronized with the server to both gain updates from the server and to provide it with what has been updated in the browser. Conflicts in updates must be detected and conveyed back to the browser for reconciliation. All of this must be done with the browser initiating the network connection; XHR is a pull-only technology from the browser to the server, thus the server cannot initiate communication with the browser once the Web application is loaded.

Because of the stark simplicity of object persistence compared to what is already provided by the browser, along with the ability to ease communication with a server on behalf of the developer and hiding browser interoperability issues, I investigated whether a JavaScript library that implements automated object persistence in an unmodified browser is possible.

The common Web browser-based JavaScript execution environment is standardized through a set of standards such as ECMAScript 5 and HTML. While ECMAScript defines the core JavaScript language semantics, other standards such as HTML4 and the emerging HTML5, define sets of APIs that are available to Web application authors. Restricting myself to a solution of the proposed problem that uses only Web standards is challenging since it means I do not have the capability to make changes to the JavaScript interpreter layer itself, but must make automated object persistence work entirely at the application level.

JavaScript is a dynamic programming language where introducing a compilation step would be a shift in development for a typical JavaScript developer. This has led me to use a different approach from previous work which typically relied on
compiler modifications to introduce object persistence [16, 29, 50]. By harnessing a mark-and-sweep algorithm in conjunction with property accessors, I have developed a solution to the transparency problem which works for dynamic programming languages which lack meta-programming abilities beyond property-specific accessors.

JavaScript’s model and support for concurrency is different from most modern programming languages. Unlike the thread model used by most languages, JavaScript is purely event-based in a single-threaded environment. This is by design to ease difficulties with concurrent programming. Programmers in general view this as a strength of the programming environment within a Web browser. This has a direct consequence for my purposes: it is not possible for a separate thread of control to be used for managing persistence. Operations which manage persistence must either suspend application progress or delay until the application is idle in a way that prevents visible (UI) response delay for the user.

These issues taken together provide a research challenge that this thesis attempts to solve. Using emerging Web standards I have created an object persistence library that works in the browser without having to modify the JavaScript interpreter or the browser (which also applies to other dynamic programming languages with similar meta-programming restrictions). The performance of the library is also acceptable when persisting thousands of data objects in a browser’s single-threaded environment. By providing a mechanism that does not require browser modification and providing a simple way to persist hierarchical data, this work addresses the shortcoming of Web applications when it comes to data storage.

1.3 Contributions

This thesis describes several contributions to the field of computer science.

- Desktop applications
  1. Extend the Principle of Explicit Authorization to implicitly grant permissions for accessing resource dependencies
  2. Monitoring the I/O of an application to grant security permissions automatically based on authorizing contexts by utilizing AOSD
3. Create an architectural design and process for the POLP to allow security contributions by various stakeholders at different levels of trust
4. Design of a mechanism to automatically enforce the POLP through the concept of authority aspects

- Web applications

5. An accessor/barrier approach combined with a mark-and-sweep algorithm for detecting the mutations of persistent objects in dynamic languages providing only property-specific accessors
6. The design and evaluation of an object persistence library to pause updates between a client and server for later completion in a single-threaded environment
7. A stateless, connectionless server design for an object persistence system
8. Synchronizing persistent objects in an eager fashion between client and server using a continuous, pull-from-server consistency protocol

1.4 Thesis Statement

Traditional desktop applications and Web applications each have unique shortcomings when it comes to accessing and managing resources; I claim that these shortcomings can be overcome. To substantiate this claim, I answer two questions with this thesis. For desktop applications, can resource access be secured by applying the Principle of Least Privilege [81] through enforcement of the Principle of Explicit Authorization [102], using I/O monitoring to drive the granting of resource access privileges dynamically during application execution? For Web applications, can the data storage facility provided by the browser be strengthened while being simplified such that managing storage resources for an application is automatic through introducing object persistence to JavaScript through a user-level library?
1.5 Outline

Chapter 2 goes into depth about securing resource access for desktop applications. Chapter 3 discusses the simplification of the storage resource for Web applications. Chapter 4 covers the related work for the thesis. Chapter 5 concludes what the thesis covered and potential future directions for the work.
Chapter 2

Harnessing User Actions to Control Resource Access

As discussed in Chapter 1, desktop applications typically secure access to resources insufficiently when compared to Web applications. Ideally desktop applications would have exactly the authority they need to access resources on the computer, but no more; the Principle of Least Privilege (POLP) says as much. Unfortunately it is typically the case that a desktop application does not follow the POLP as this principle requires that permissions be dynamically granted and revoked while an application is running. This must be dynamic instead of statically declared before the application is executed because the needs of the application can vary between executions, leading to excess permissions being granted which violates the POLP.

One reason why desktop applications do not apply the POLP is that its enforcement can be difficult. The dynamic granting or revocation of permissions can be difficult to implement correctly and one must take into consideration many aspects of the running system, including configuration files, user interaction, and various pieces of I/O. Capturing all of this information in order to drive security decisions requires tying into a wide portion of an application in order to not miss any relevant I/O coming from a trusted code base.\(^1\) My research has involved developing a cross-cutting concern where the POLP for a desktop application is enforced using various types of I/O when making security permission decisions.

\(^1\)Code which is assumed to act properly on behalf of the user.
2.1 Motivating Example

To motivate the work, I use an example where my approach is applied to an open-source application, RSSOwl [73] (in Section 2.5 I present more details of RSSOwl). I will refer to the identity of a resource as being represented by a URI regardless of whether it is a network or file-based resource. A such a URI should be considered encompassing the realm of URLs.

The primary function of RSSOwl is to maintain a list of news feeds that a user is subscribed to. When a user selects an RSS news feed, it is downloaded, checked to see if any new items in the feed exist, and the headlines of the items are displayed. Naturally, RSSOwl will require access to a large set of resources such as: configuration files, news feed hosts, software update sites, etc.

The URIs naming file and network resources that RSSOwl needs to access are specified in three ways: in a security policy file, through the GUI, and through disk and network I/O. A security policy file is used to grant access to resources needed for RSSOwl to simply begin execution. This includes its configuration file and the URI to check for new versions of the application. However, in many cases the URIs of resources that a user wants to access can only be determined by the interaction of the user with RSSOwl at run-time; such as when a user enters a new feed subscription into the GUI or identifies a file to read. In this case the GUI framework\(^2\) is given the URI that the user wants to access. A URI can also originate from the network or the disk by the user initiating an interaction with either resource. The key point is that RSSOwl can receive URIs from various places and sources, but no matter the origin, RSSOwl must be granted the proper permissions to access each URI. An example of a GUI widget taking a URI as input is shown in Figure 2.1 with the text box accepting the URI labelled “A”.

2.1.1 GUI I/O

When a URI is entered by a user, the GUI framework passes that string unmodified to RSSOwl. Placing the GUI framework (but not the application’s GUI itself) within the trusted code base means one does not need to worry about the URI being modified between the user inputting the string and the point where it enters

\(^2\)A software toolkit used in helping to create a GUI for an application.
the application code. If a user is told upfront that any URI entered into the GUI is
assumed to be for a resource they wish to access, a user can be trusted to not enter
URIs which represent resources she does not want to grant permissions for. Trust-
ing the GUI framework and the user makes it safe to grant permissions to access
any URIs entered into an application’s GUI. This is considered “direct input” as a
URI entered through the keyboard can safely be understood to be aacsURI with-
out requiring any application-specific interpretation. Certain other GUI interactions
such as a file dialog box are also unambiguous regardless of what the application’s
purpose is.

My approach leverages authority aspects which capture input in these autho-
rizing contexts where a user enters a URI into the GUI and grants the proper per-
missions to the application. Both authority aspects and authorizing contexts were
introduced in Section 1.1.2.

“Indirect input” can also happen when the mouse is involved. When a user
clicks on an HTML link, it is obvious to her what that action is meant to represent,
but in terms of an authorizing context it is not necessarily obvious. In this instance if the application developer can state what permissions an action will cause to be granted then the ambiguity is removed. If a user is then told upfront through some interface what will occur, she has the option to find out what consequences the indirect GUI interaction with an application will cause before actually performing the action.

2.1.2 Network and Disk I/O

In some cases, resources which are needed by the application are not directly requested by a user but instead are indirectly referred to by other resources requested by the user. In these cases, a desktop application would first receive permission to read some resource from the file system or network. Then, based on URIs in the resource data, some additional permissions would be needed. Depending on application semantics, it may be useful to transitively grant permissions for these contained URIs without user intervention. Otherwise the application might need to needlessly prompt the user to grant clearance to resources, annoying the user.

An example is illustrated in Figure 2.2 which shows a link tag from RSSOwl’s XML-based configuration file. The XML file specifies a link tag for each feed subscribed to by the user in previous sessions. The file resource for RSSOwl’s configuration file thus contains URIs which a user would seemingly want to grant RSSOwl permission to access on her behalf as she added those news feed URIs herself in previous sessions. If a user can specify what resource the configuration file is and how to find URIs in the file, then granting permission to read the configuration file should implicitly grant permissions to access the URIs listed in the configuration file.

In these cases, authority aspects are used to monitor file and network streams for URIs which appear in a context that implies a legitimate need for the resource.
Since the semantics of URIs appearing in different contexts is application-specific, support is provided for policies which can be tailored to different deployment scenarios.

2.1.3 User Prompting

Finally, in a remaining number of cases, neither a code access policy or dynamic monitoring provides appropriate grounds from which to base access control decisions. In these cases my approach falls back to explicit user prompting as a last resort. This ensures the approach is fail-safe but could be considered impractical if too much user prompting is required; in general users dismiss incessant prompting dialog boxes without giving much thought to their content. Section 2.5.1 shows the number of instances within RSSOwl’s UI that require changes with and without my approach to lower the amount of prompting.

In this section I used an example to demonstrate that certain data flowing into an application from trusted sources can be used to guide resource access control decisions. In order to take advantage of this information coming into the application, some code describing the semantics of the sources must be provided. Unfortunately, the application developer cannot be depended upon to provide the code which describes these input sources, since it is their code which may be suspect. The challenge then is to devise a methodology and mechanism that minimizes the amount of explicit user prompting, while securing the application. The remainder of this chapter describes an approach that allows this code to be implemented and distributed by a trusted third-party, separately from individual applications.

2.2 Architecture Details

I use an aspect-oriented approach which is designed to leverage knowledge that is provided by three developer roles in the construction of secure desktop application software (discussed in Section 2.2.1). This distribution of work allows experts in their respective areas to contribute as appropriate to their skill set rather than requiring a single individual to work in areas they may not be familiar or comfortable with. It also allows varying levels of trust of the participants playing these three developer roles based on their skills and trustworthiness.
These participants provide code for four sources of protection state (discussed in Section 2.2.2). The various sources provide varying levels of automated granting of permissions based on the situation being affected.

### 2.2.1 Developer Roles

There are three developer roles in the construction of secure desktop application software are: a trusted POLP policy expert (policy expert), a trusted I/O framework expert (framework expert), and (optionally) a potentially untrusted authority-aspect aware desktop application developer (application developer). These three roles devolve from the typical software development process where an application developer creates software by relying on pre-existing code created by a framework expert, e.g., a GUI library or a standard library included with a programming language. The security expert fits into the development process by being a domain expert for the issue being worked on. The concept of the security expert in the
design process, while not unheard of, is not typical and thus an addition of my own choosing. The role allows for someone who has the proper domain expertise create the base software for the security framework to provide a solid foundation for the other two roles to work from. While these roles could potentially be filled by the same person, at least the code developed by the policy expert and the framework expert must be trusted by the end–user. The dependencies between these roles is shown in Figure 2.3; roles listed higher in the stack depend on the roles below them.

**Policy Expert**

The policy expert designs abstract aspects which reify a cross-cutting POLP policy in terms of abstract pointcuts, advice, and/or abstract methods (these various terms were introduced and defined in Section 1.1). The details of abstract pointcuts and abstract methods can be left to be filled in by one of the other two developers. This is equivalent to the policy expert providing a superclass for the other participants to subclass as needed.

In the case of an abstract pointcut, exactly what join points are to be matched is left to be filled in by the framework expert by the pointcut she defines. For the abstract methods, the framework expert decides exactly how to handle the information that is captured by the advice which uses the now-defined pointcut to specify what I/O to capture. Advice contained within the abstract aspect uses the now-defined pointcuts to match join points of interest. The now-defined abstract methods call methods provided by the abstract aspect to perform security actions, e.g., granting permissions for a specific URI.

This promotes reuse of the policy on more frameworks and applications. The POLP policy expert can mark security-critical pointcuts or methods as “package protected” to ensure that only a trusted I/O framework expert should be allowed to override them. If the abstract aspect includes pieces which are safe to be filled in by an untrusted application developer, these pieces should be marked “protected”, allowing them to be overridden by any sub-aspect. The security expert is thus able

---

3 Points in execution where code can be injected.
4 Although packages are open in the Java programming language by default, the Java security policy language allows package membership to be open only to classes from certain binary components.
to provide abstract authority aspects which can be used by a framework expert or application developer without allowing either of them to gain abilities that the security expert does not explicitly provide (how this works is discussed in lower-level detail in Section 2.2.2). For RSSOwl, the policy expert designs nothing specific, but instead creates generic authority aspects for the framework expert and application developer to use.

**Framework Expert**

The framework expert subclasses an authority aspect to tie a platform-specific implementation for some type of I/O to the aspect to grant permissions as necessary. The connection is performed by implementing abstract methods and/or pointcuts which require platform-specific I/O API domain knowledge to implement correctly. This allows the abstract policies specified by the abstract authority aspects to be available to more I/O frameworks used by application developers. Such frameworks would include various GUI platforms and application-layer protocol stacks. For example, the framework expert would take an abstract authority aspect which grants file access and implement the details for using the file access dialog box which the GUI framework presents to a user when they want to open or save a file.

Authority aspects need to be provided for the various entry points of I/O into an application. This includes not only the visual I/O of the GUI but also the network and disk I/O that a user does not directly observe. In all cases, an authority aspect that has been designed to handle the special requirements for the various types of I/O is needed. For RSSOwl this means using authority aspects for SWT – a GUI framework – and various authority aspects for the Java Development Kit (JDK) as created by a framework expert. In the case of SWT, authority aspects would be needed which handle, e.g., the file dialog box API. For the JDK, authority aspects would be needed for, e.g., the various networking APIs. Examples of authority aspects like these are shown and discussed in Section 2.3.

**Application Developer**

The application developer develops a desktop application with the awareness that only those types of I/O that have a corresponding authority aspect should be used.
in order to avoid explicit user mediation. For some authority aspects, the application developer will be allowed to override abstract pointcuts/methods to include application-specific knowledge. This allows application-specific interactions to be able to participate in the decision of whether to grant the authority to access a resource while still allowing a user to make an informed security decision. Note, however, that the aspects the application developer subclasses contain advice which are designed to explicitly notify a user through the GUI of what permission an action will cause without granting permissions directly, allowing the application developer to continue to not be a trusted participant. For example, in RSSOwl the application developer provides an authority aspect that grants permission to access a URI when the user clicks on a news feed in the GUI after notifying the user of the permission to be granted upon clicking.

Through a combination of AspectJ’s abstract aspect mechanism and the existing Java security mechanisms, this approach allows for a separation of concerns between the three developer roles based on their individual expertise. The separation of concerns is paired with a corresponding separation of trust to ensure the safety of the final implementation.

The end-user is expected to obtain the binaries for some authority aspect(s) from the trusted framework expert and then use these aspects in conjunction with a desktop application that relies on any framework for which they have obtained the appropriate aspects. The aspects the developer provides can already be compiled into the application binary as a Java security policy file is used which only allows aspects supplied outside the application to grant permissions (this does require that the abstract authority aspects are provided externally to the application). In order to make the process of applying the authority aspects to an application simple for the end-user a shell script is provided that weaves the aspects into the application at load-time.

**Summary**

As a high-level overview, the security model is structured such that:

- The application developer is not trusted to grant permissions as the developer may be malicious, although she can participate in a secure fashion if desired
by providing “hints” as to what actions should provide what permissions

- Authority aspects are trusted
- By default, all resource accesses trigger an explicit prompting of the user for clearance
- A code access policy can be provided to grant authority known statically to be needed to mitigate prompting
- The application developer may provide subclasses of the authority aspects that define authorizing contexts to dynamically grant authority in an application-specific way to further mitigate prompting assuming the authority aspects are audited for security purposes

2.2.2 Runtime Architecture

The runtime architecture of a desktop application being monitored by authority aspects is shown in Figure 2.4. The End User (top left) interacts with the Application (large box on the right), through the User Interface (screenshot with a green border). The goal of the monitoring mechanism as introduced by authority aspects is to ensure that the application has sufficient access to resources to meet the needs of the user based on the user’s interaction with the application, but no additional resource access.

The heart of the mechanism is the Java Security Manager (shown centre). It acts as a resource mediator, to determine whether a resource request by the application should be allowed.

In order to be useful, the security manager needs instructions regarding which resource accesses to allow and which to block. These instructions are shown in the figure flowing into the security manager (shown as dotted lines labeled Add Permissions).

In this approach, an application’s security is derived from four sources:

1. Explicit user prompting
2. Code access policy
3. Authority aspects

4. Aliasing

Here I describe each in turn; in Section 2.3 I focus on further details of the authority aspect framework and other aspects of Figure 2.4 since this is my primary contribution.

Explicit User Prompting

The easiest way to ensure that no resources are accessed against a user’s wishes is to explicitly prompt the user [102]. These prompts are used, for example, in some client-side firewall programs. A typical prompt will ask: “The application is trying to access resource: allow once, allow always, or deny?”, where resource refers to a sensitive or suspicious resource. This is demonstrated in Figure 2.4 by the dotted line between the end user and the security manager.

Explicit user prompting provides total resource mediation but also requires the explicit involvement of the user. This can easily be done by tying into the security manager and having all failed authority access requests trigger a prompting. So on the one hand, user prompting provides mediation for every single resource access attempt by the application and is totally dynamic. On the other hand users may become frustrated or annoyed by frequent prompting. This is described in the literature as psychological acceptance [81] as “the human interface [is] designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly”. This is such a well-known problem that it was the subject of a television advertisement. 5

The security framework is intended to ensure psychological acceptance by minimizing the amount of user prompting necessary and making security decisions on behalf of a user based on the I/O she drives through the application. This is done by monitoring contexts which imply that a user has already expressed her intention to access the resource. For example, if a user enters a URI to add a RSSOwl news feed, there is really no need to ask one more time if she wants to access that

5See the Apple security ad from 2007 at http://www.cs.ubc.ca/~drifty/macsecurity.html
Figure 2.4: Architecture of mechanism to monitor resource access for a desktop application
resource. Section 2.5 evaluates how much prompting is necessary for two medium-size desktop applications after the security framework is used.

**Code Access Policy**

In managed execution environments such as Java and .NET, resource access can be mediated through a code access policy. In Java, for example, a software component is usually identified by a “jar” file which contains various packages and classes. A code access policy in Java can then specify permissions that apply to all classes defined within a “jar” file, e.g., restrict which classes can read a specific file to those contained in a specific “jar” file [2]. This is because some software components may be trusted more than others. Although the details vary, the important point for the purpose of this discussion is that these code access policies are specified statically and do not consider dynamic program input.

In my approach, authority aspect binaries compiled by the user from the code provided by the policy expert, framework expert, and application developer are distinguished in a code access policy provided by the policy expert so that these components may dynamically grant new authority to undistinguished base application components, which have no authority by default. Sometimes configuration files or directories are also listed so that they can be loaded without issue, but as it will be pointed out in Section 2.5.1 it is not always the case that configuration files may be granted authority upfront.

A whitelist is “a list or collection of people or entities that are known, trusted, or explicitly permitted”. A code access policy acts as a whitelist of resources that, amongst other uses, specifies what resources the application can access upon startup. This makes it very easy to perform a security audit of what resources will be accessed by the application starting at execution time.

**Authority Aspects**

Monitoring input provides the leverage needed to control resource access dynamically. This is because input data often implies the need for resource access, but

---

6 A collection of compiled Java source code.
output data does not. For example, HTML read as input to a Web browser creates a burden on the application to retrieve additional resources such as images. Consideration for output is only necessary to ensure that safeguards are not circumvented, e.g., URIs written to the application’s configuration file were specified by the user and not arbitrarily by the application. These details are described in Section 2.3.

Monitoring input to authorize resource usage cross-cuts the architecture of a desktop application. As is typical, input comes from three primary sources: the disk, the network, and the GUI. The security framework leverages AspectJ [56] to weave security code into applications written in Java. Authority aspects are used to identify URIs which appear in authorizing contexts.

At the lowest level, desktop applications read from disk and network resources through input streams (represented at the bottom of Figure 2.4). Desktop applications often use standards such as HTTP to communicate over streams using other standards like XML to send structured data. When structured data is read from streams, aspects monitor the read operations. As the structured data is read, authorizing contexts are searched for. When found, URIs contained in the authorizing contexts are extracted by advice and the application is granted permissions by calls made to the security manager.

The GUI is typically how a user directly interacts with a program. Users often request access to resources through direct GUI input via the keyboard and indirect GUI input via mouse usage. GUI inputs range from file choosers and text fields to mouse events involving tree views and tables. Because GUI events typically trigger a callback that leads to code interpreting the GUI input, the code locations where GUI input is read can be spread throughout an application’s code base, emphasizing the cross-cutting concern that it is. When data is input through the keyboard and mouse, aspects are used to monitor GUI inputs that imply the need to access a resource. URIs that appear in these authorizing contexts are then cleared through the security manager.

A user needs to be aware of what consequences their actions may cause. When directly inputting text into the GUI, she must know what they are typing. However, when a user selects items such as an HTML link using the mouse, she needs to be told what resource access her selection will require.
Aliasing

One detail that differentiates permissions originating from disk and network input compared to GUI input is the inherent lack of control a user has over triggering disk or network usage. Because of this, the idea of permissions that are potentially but not necessarily required needs to be considered when dealing with input from disk or the network. The situation can arise where it is best to capture resource identities from disk/network input, but not to grant the permission to access those resources immediately. Instead, an association can be made between some GUI interaction and the granting of permissions for a resource originally identified through the network or disk. This is done through associating an alias for a resource and the permissions desired. With the alias displayed in the GUI, interactions with the alias can properly grant permissions for the associated resource.

An example of this is anchor tags in HTML. When an HTML page is downloaded, it contains links to other pages that would require permissions to access if the links were clicked. One would not want to grant permission to access all URIs listed on a page simply because they are there, but instead based on being selected. But a URI is not always as descriptive as the text that comprises the link. Making an alias to the URI from the linked text can allow for a more rich and easily understood association of what permissions are to be granted for something in some cases.

Some GUI widgets do not have the user enter any information directly, but instead select from various options. Tree and table views (e.g., a traditional file/folder tree representation or a grid of sales data) epitomize these types of GUI widgets where the user interacts with information instead of entering it. Using aliases allows the semantic connection between what the user sees in a selection widget and what resource it is connected to. This association is communicated with the user through a Status Bar (shown in the left side of Figure 2.4) which updates the user as to what, if any, authority will be granted if they select a GUI widget. This is much like a status bar in a Web browser that displays what URI a hyperlink points to when a user hovers their mouse over the link text. Technical details are explained in Section 2.3.2, but briefly this is done through an authority aspect that is provided by the application developer.
When a user either single- or double-clicks on an item that is an alias for a resource, permission is then granted for that resource if it has not already been granted. This allows a user to control when permission is granted for resources that are specified by e.g., downloaded XML that is being presented to a user in a non-XML format by allowing a user to view the Status Bar to see what permissions will be granted when they (double-) click in the GUI.

Consider an RSS feed full of news items. After making an association between each news item’s title and the URL associated with it, the news items can be displayed by title in a table view. When a user hovers over a title in the table view, the status bar alerts her that clicking on that item will grant the permission to connect to the URI associated. And if she does decide to click on the item in the table, then the proper permission is granted.

When input originates from disk, typically the application generated the input itself or the user explicitly chose the file on the disk. But for input from the network, neither the application nor the user have control over what the computer at the other end of a socket connection will send back. Thus it is prudent to consider resources specified by network input as leading to potential permissions. Through either the resource identity itself or an alias tying the resource to a human-readable name, GUI input from the user can then be used to grant the potential permission and thus keep the user in control and not some third-party sending data to the application.

**Summary**

In spirit, my architecture is like the very simple cp example discussed in Section 1.1.1. The architecture takes the interaction of a user with an application along with the I/O triggered by those interactions, and then uses that information to grant permissions on behalf of the user. By relying on what a user explicitly does with the application only the permissions needed for the application to work as the user desires will be granted; this is the essence of the POLP. The research challenge is to determine if this approach can scale to realistically-sized applications which may use many input and output channels.
2.3 Implementation Details

The I/O for an application originates from different resources. Stream-based I/O from the network and disk is one point of origin. The GUI provides a user-facing origin of I/O. While both provide input which can contain URIs that require permissions, neither explicitly provide for the need for the deletion of permissions which must be handled separately.

The various types of I/O which help drive dynamic security decisions are listed below for convenient reference and are discussed in the rest of this section.

- Network (Section 2.3.1), e.g., HttpURLConnectionAspect (Figure 2.8)
- Disk (Section 2.3.1), e.g., FileStreamAspect (Figure 2.9)
- Application configuration file (Section 2.3.1)
- Text (GUI) widgets (Section 2.3.2), e.g., DirectGUIAspect (Figure 2.10)
- Mouse GUI widgets (Section 2.3.2), e.g., IndirectGUIAspect (Figure 2.11)
- Deletion (Section 2.3.3), e.g., DeletionAspect (Figure 2.13)

2.3.1 Network and Disk I/O

A malicious developer or piece of code could either attempt to: force authority aspects to monitor streams they should not, or hide from aspects streams that should be monitored. The first case is not possible since the user controls which files and network hosts can be accessed, based on their GUI input and XPath policy which builds up what streams to monitor, starting from nothing beyond what is specified in the security policy file. In the second case, hiding a stream from the aspects would only result in some permissions not being granted as the user would be prompted for those permissions rather than having them cleared by the authority aspects. This could only serve to annoy the user which may cause them not to use the application. Regardless of whether the developer is malicious or not, providing the user with an application that is hard to use would not provide any advantage for some malicious adversary. This acts as a motivation for application developers to
provide authority aspects which properly work with network and disk input used by the application.

System administrators or users can control a policy file consisting of authorizing contexts specifying which URIs are authorized based on their location in some XML document from disk or network. The XPath policy file consists of a list of XPath rules. Each rule specifies if it applies to a specific document (e.g., configuration file), all network input, or to all documents selected by the user through the GUI. The rule then contains location steps which point to a specific XML tag.

Consider RSSOwl and its configuration file (discussed in Section 2.1). The rule to grant permissions for the feeds the file contains is listed as the first rule in Figure 2.5. The rule for RSSOwl's configuration file format acts as an authorizing context saying that the application should have access to any URI that is given in a link tag that is nested below the tags rssowl and favorites in the document located at configFile.

Supposing this was the only rule in some policy file, the user would safely be assured that the only URIs which the desktop application will have access to are

Figure 2.5: XPath code policy rules for RSSOwl
those identified by that rule or given explicitly through the GUI. The user would not have to be prompted to give explicit permission for all of these resources which are listed in the configuration file.

Desktop applications interact with network and disk resources using stream buffers. In Java, the class InputStream serves as the superclass for network and disk-bound byte streams (shown as Application Streams in Figure 2.4). Authority aspects monitor information read from InputStreams in standard formats in order to capture URIs appearing in a context that implies a user’s intention to allow resource access by the application (shown as Aspects surrounding Application Streams in Figure 2.4). This intention is described using XPath-based policies that are written by the policy expert (shown being fed into the aspects of Figure 2.4).

The grammar for the rules is shown in Figure 2.6. Rules either target a document or a MIME type. For documents, wildcards can be used to match all files of a specific type (e.g., *.xml for all XML files). MIME types allow for identifying formats that are received from over a network using HTTP. Figure 2.5 lists the XPath rules needed for RSSOwl: its configuration file format, OPML, Atom, RSS, and HTML.

An application developer can use the security framework to force their application to follow the POLP by providing pointcuts which identify InputStream objects that should be subject to monitoring. The aspects which the application developer would use are typically provided by a framework expert, thus not necessarily requiring knowledge of AOSD. The aspects should also either be audited before compilation or provided by a trusted third-party to prevent tampering by the application developer.

The objects which are to be monitored must be instances of classes from a trusted code base, in this case the JDK classes. This is because the security frame-
work must be sure that the stream is actually connected to the file system or network and not some fraudulent resource which is simply feeding carefully crafted data to the authority aspect in order to gain arbitrary resource access. This requires monitoring streams in general, and network and disk streams specifically.

**Stream Monitoring**

Typically the first way that structured input flows into an application is through the disk and network as streams of data. Independent of whether it is from a configuration file on disk or some data downloaded from the internet, structured input does flow into an application through ways other than the GUI. Therefore, the security concern must monitor disk and network activity for authorized resource identifiers. The abstract approach taken allows the developer or user to write data-specific code to pull out the resources desired from supported structured data formats (currently XML and HTML). Having to specify per-format code for input coming from the network or disk is an issue faced by others in the security community who have tried to come up with a general solution to dealing with the myriad of data formats out there [74].

The monitoring of streams is achieved through wrapping a decorator\(^8\) [35] around a stream object. An advice then sends data read from the stream to both the code in the advice and the application itself. After returning from the advice, the desktop application would read bytes from the wrapped input stream without awareness that it is being monitored. However, inside the decorator, whenever some bytes are read, they are also copied and passed on to a content handler\(^9\). This handler executes in a separate thread, so the application is not required to block while the data is processed by the handler. Essentially, every byte that is pulled from the stream by the application is also simultaneously pushed to the content handler thread by the decorator.

To know how to process the stream, the content handler can also detect when an XML file is being read based on whether an XML parser can parse the data coming from the file or introspecting on the data. Potential aliases between data (e.g., the feed title and the URI in RSSOwl) can be made so that when the GUI presents a

---

\(^8\) An object which adds new/modified behaviour to an existing object, typically transparently.

\(^9\) A method which processes data.
URI by alias alone, the proper granting of permissions for the URI can be done. This also extends to other formats if the wrapping stream could discern between varying formats when the data format is not known upfront. Typically, though, this automatic dispatch of processing based on introspection of the data is not needed as network protocols specify their formats in their header packets (e.g., HTTP and MIME types) and files usually have a file extension (e.g., .xml for XML files).

One final implementation detail needs to be handled for this approach to be sound. Since the desktop application and content handler execute in separate threads, there is a race condition where one thread finds URIs in the stream first. This is a problem if the desktop application requests access to a URI that it finds before the content handler has time to find and authorize the URI. For this reason, the security manager must be locked until all content handlers executing at the time of the resource request are finished.

This approach is used rather than forcing a sequential execution of the desktop application and content handler because it is unlikely the desktop application will ever beat the content handler in some race. That is because the content handler is only programmed to find URIs and pass them to the security manager, whereas the desktop application is a complete application, programmed to process all the data in a stream and apply arbitrarily complex program logic on the data. So, even though the lock is not likely to be necessary, it must be included to ensure correctness. In cases where it is not needed, the content handler and desktop application are free to run concurrently.

The code which handles the wrapping of a stream is the wrap method (Figure 2.7). The method takes an InputStream captured at some join point in desktop application execution along with the format of data being read in from a stream. Different authority aspect specializations of stream handling (file and network) rely on this method to initiate the stream monitoring process.

The stream is wrapped by a decorator to enable monitoring of the data read from the stream (the stream is an argument to the method as shown on line 10). After duplicating the stream so that two copies exist (the PipedInputStream and PipedOutputStream on lines 11-13), the decorator (line 14) forwards information read from the stream to a content handler which parses either XML or HTML data (lines 16-19). This handler attempts to match rules from the XPath
Method to duplicate a stream so that the application can read from the stream while it is also processed for URIs.

@param stream the data stream to monitor
@param format the format of the data coming over the stream
@return delegate which sends duplicate data from the original stream

```java
static T wrap(T stream, Format format) {
    PipedInputStream in = new PipedInputStream();
    PipedOutputStream out;
    out = new PipedOutputStream(in);
    T delegate = getDelegate(stream, out);
    if (format.equals(Format.XML))
        // Parse XML from stream for URIs.
    else if (format.equals(Format.HTML))
        // Parse HTML from stream for URIs.
    return delegate;
}
```

Figure 2.7: The `wrap` method, written by a policy expert, is used in authority aspects to begin monitoring some point of I/O

policy. The content handler starts consuming input in a separate thread from the application (details are not shown but are straightforward). This allows the desktop application and the content handler to execute concurrently. When a URI is matched in a context specified by the policy, the URI is passed to the Security Manager to have the proper permissions granted for the URI (as shown by the Add Permissions edge in Figure 2.4).

Having a general framework in place to handle streams, it can be used to build solutions specific to the type of stream. Disk and network streams are the two that are worked with inside the security framework.

Network Input

Since the user controls which URIs are accessed, this prevents the use of network I/O in some malicious manner (e.g., launching an arbitrary Denial of Service (DOS) attack to a site of its own arbitrary choosing). As long as the user never deviates from trusted URIs then they will never be put in a position of untrusted I/O driving permissions for the application. To help enforce this, permissions at the level of
URIs are used, not just hosts as the Java security manager natively supports as the user might want to visit a domain which hosts services for many various developers. For example, the user may trust the personal website at www.cs.ubc.ca/~drifty/ but not that at www.cs.ubc.ca/~wohlstad/ which is at the same domain but is a different URI.

By doing network permissions at the URI level I am able to prevent certain potential attacks the desktop application could perform. Consider visiting a website. To properly display a page, you need to download the content specified in link, script, img, iframe, and frame tags. If any of these tags (or the page currently being viewed) led to a domain which provided a mechanism for generating Web pages (e.g., a message board), the desktop application could gain clearance to visit the host in general, generate the page it wanted at the same domain, connect to the created page, and then gain the permissions it wanted. One could simply not provide XPath rules for HTML, but that would quickly deteriorate into more user prompting than one would like as many websites store static content (e.g., images) at another host for performance reasons. Plus the user would have to be advised that visiting any domain that contains Web pages that could generate other pages dynamically is dangerous, which I deemed an unacceptable restriction. My approach avoids these pitfalls by not granting permissions for URIs which the user has not explicitly visited or URIs implicitly required to properly render the Web page being currently viewed. This prevents a malicious desktop application from arbitrarily choosing to direct a user to a malicious URI at a domain which just happens to also host a trusted URI she chose to visit.

The current implementation supports desktop applications which make use of the HTTP protocol for application-level transport. Figure 2.8 shows an aspect which supports an HTTP library included with the JDK. Support for third-party HTTP libraries could be added by a framework expert familiar with the third-party library. The pointcut identifies the input stream objects which are subject to monitoring (lines 10-11). The protected pointcut restrict (line 17) can be used by a desktop application developer to further restrict the set of connections that are monitored. This is useful for optimization if an application developer has application-specific knowledge that some HTTP connections will never be used to read in resource identifiers.
Monitor an HTTP connection for resources.

```java
/**
 * Pointcut which captures all accesses to an HTTP stream.
 */
public abstract aspect HttpURLConnectionAspect
    extends StreamAspect<java.io.InputStream> {
    /**
     * Pointcut that, when overridden, specifies what HTTP streams *not* to
     * monitor.
     */
    protected pointcut restrict();

    /**
     * Advice which wraps an HTTP stream so that all data from the stream goes
     * both to the application and to a method to process the data for
     * resources.
     *
     * @param that an HTTP stream to process for resources
     * @return original HTTP stream for the application to read from
     */
    InputStream around(HttpURLConnection that) : target(that) && monitor() {
        String uri = that.getUrl().toString();
        if (SecurityManager.isURISafe(uri)) {
            SecurityManager.allowProceed();
        }
        InputStream stream = proceed(that);
        if (that.getContentType().contains("html"))
            return wrap(stream, Format.HTML);
        else if (that.getContentType().contains("xml"))
            return wrap(stream, Format.XML);
        else
            return stream;
    }
}
```

Figure 2.8: HttpURLConnectionAspect network stream monitoring aspect written by a framework expert
Connections are handled by the around advice (lines 27-41). The aspect allows the application to open a new stream only if the URI specified has been previously cleared (lines 29-31). As was described, this clearance would have been made by some other authority aspect.

The advice is used to deduce whether HTML or XML was received (lines 35-40). The advice monitors the HTTP Content-Type header to make the determination. Based on this information a call to the wrap method, provided by the policy expert, is used to begin monitoring the content as described in the previous section.

Disk I/O

A desktop application will typically read configuration data into memory when the application is launched. Then, during program execution, the data in memory can be mutated by the user through a GUI. When the program exits, the new data is written back to disk. For example, RSSOwl saves previously added news feeds in its configuration so they are available to be read at launch.

As mentioned previously, rules in the XPath policy file can be rooted at a specific document. For these rules, the specified documents are read upon the start of the security manager and before the application begins. This allows execution-critical permissions required for the application to begin functioning to be created before the application needs them.

During program execution, aspects monitor the use of the GUI for any additional URIs to be cleared or URIs removed (described in Section 2.3.2). Thus, the security manager state reflects both the URIs from configuration data and changes during execution. Still, you need to save this state in some way, so that it can be used the next time the application is launched.

Ideally, since the application will save its own state in its configuration files, it may seem possible to simply leverage the application’s behaviour. That way the security manager would not need to persist its own state; when the application starts up again, the protection state could be reconstructed by interpreting the configuration files. However, this approach is difficult to apply in practice directly.

The problem with this ideal approach is that it may not be able possible to trust the data written by the application to disk. Even if the software component which
writes the data could be trusted, the data itself may have become tainted by some other malicious piece of code.

To deal with this problem, the security framework leverages the application’s own behaviour for saving configuration data, but also performs additional run-time verification. This is achieved by post-processing the configuration files written by the application. Each configuration file is read by the security manager during program exit\(^\text{10}\) to determine if it matches the state of the security manager. If it does not, a warning is issued to the user.

Post-processing of configuration files is achieved using the same content handlers which are used to interpret the reading of file streams during regular program execution. The only difference is the way in which URIs are handled when they are discovered. In this case, instead of authorizing URIs which are discovered, the security manager ensures that the URI already exists as part of its state.

A FileStreamAspect, similar to HttpURLConnectionAspect is used to monitor file I/O (Figure 2.9). When a file is opened for reading, the path is checked against the files that have been flagged for generating permissions (lines 28-30). If the path matches, then the InputStream gets wrapped so that the file can be parsed for permissions as data streams in (comment at lines 34-35), just like I do for network input (as discussed in Section 2.3.1). I make use of another subclass of StreamAspect for file input which works similarly to the HTTPStreamAspect previously shown.

**2.3.2 GUI I/O**

In addition to monitoring information from I/O streams, authority aspects are also used to monitor the GUI (shown as Application GUI Widgets in Figure 2.4). This requires covering not only widgets for entering text, but also widgets controlled through the mouse. Both types provide their own unique challenges to capture the URIs that they may contain.

\(^{10}\)Java provides for the execution of a shutdown hook which is executed whenever a Java Virtual Machine (JVM) terminates.
public abstract aspect FileInputStream
extends StreamAspect<java.io.InputStream> {

    // Pointcut to capture the creation of a FileInputStream.
    pointcut monitor(): call(FileInputStream.new(...)) && !restrict();

    // Abstract pointcut to allow the skipping of processing certain
    // FileInputStream instances.
    protected pointcut restrict();

    // Advice to verify that a file stream should be opened, duplicate the
    // stream, and process the stream for URIs.
    @param that file stream to process
    @param arg File argument to create stream
    @return stream that is being processed
    InputStream around(FileInputStream that., File arg): args(arg) && monitor()
    {
        String uri = that.getAbsolutePath();
        if (SecurityManager.isURISafe(uri)) {
            SecurityManager.allowProceed();
        }
        InputStream stream = proceed(that);
        return stream;
    }

    // Advice to do the same as above, but handle the case of when the
    // FileInputStream is created with a String argument.
    @param that file stream to process
    @param arg string argument containing the path to the file
    @return stream being processed
    InputStream around(FileInputStream that., String arg): args(String) && monitor()
    {
        String uri = that;
        /* Same as advice above from this point forward. */
    }
}

Figure 2.9: FileStreamAspect file stream monitoring aspect written by a framework expert
Text I/O

Text can enter from the GUI in various ways. This can be from using a text field directly or a file chooser that leads to a file path being returned. The key point is that the intended use of a GUI in relation to URIs cannot be determined from inspecting GUI widgets.

By crosscutting over all GUI widgets that return a string as I/O into the application, any potential resource identity that is directly typed in by the user can be caught entering the application. This helps to alleviate the problem of not knowing the meaning of various GUI widgets that directly take input from the user. Thus if a text field prompts a user to directly type in a file path or URI, my approach does not need to know that fact explicitly for the specific text field in question.

Although the security framework can trust the user who enters text into the GUI, most text-based widgets also support values being fed into the widgets programmatically. For example, many widgets include a method such as `setText`.

This is a problem if you cannot trust all code which might influence these values. Consider that a virus may modify the fields arbitrarily or a plug-in may be able to influence the value through data-flow from its interaction with a plug-in API. This means that text set programmatically is considered “tainted” and inherently unsafe and thus should not be trusted as representing what the user meant to identify as a resource to access [6, 42, 94].

The XPath rules that are not anchored to a specific file read are applied to data obtained from any URI granted file read access through a text widget. As mentioned previously, the user needs to be aware of this possibility and thus not enter a path to any file that could potentially grant permissions they don’t want granted. Otherwise the user must make sure that no XPath rules are defined that would grant permissions that are not desired. Blacklisting is not supported as that leaves the user open to exploitation if they are not diligent in keeping the list up-to-date.

To monitor URIs entered as text strings by the user, I created the `DirectGUIAspect` aspect as a policy expert, in Figure 2.10. This abstract aspect is parameterized by the type of the widget, for example, `JTextField` (Swing), `Text` (SWT), etc. (line 6). Concrete specializations of this aspect must
Provide an authority aspect for a GUI widget type.

@param T widget type to support

abstract aspect DirectGUIAspect<T> {  

    What URI the widget contains.

    @param widget to be inspected
    @return URI contained in the widget

    abstract String toURI(T widget);

    Pointcut that captures join points that signal the widget should be
    examined.

    abstract pointcut textReady();

    Advice which authorizes use of the URI contained by the widget as long
    as the widget is not tainted.

    @param that widget being examined

    after (T that): target(that) && textReady() {
        if (that.tainted)
            return;
        SecurityManager.authorize(toURI(that));
    }

    Pointcut which captures join points signaling that the application has
    programmatically set text in the widget, thus tainting the data.

    abstract pointcut catchTainting();
    boolean T.tainted;

    Advice which flags the widget as tainted.

    @param that widget that has been tainted

    before(T that): target(that) && catchTainting() {
        that.tainted = true;
    }

}  

Figure 2.10: Abstract class for tracking text entered into the GUI, written by
a policy expert
be provided by a framework expert because these aspects must be trusted.

The method `toURI` (line 14) is overridden by a framework expert to provide the `URI` which is currently input to the widget (or `null` if none is available). The pointcut `textReady` (line 19) is used to signal when the widget should be checked for text inputs (typically when a method used to read from the widget is called) so that the authority aspect does not need to poll the widget to know when there is input to read. At this point, the `URI` is read from the widget (line 31) and cleared by the security manager unless the widget has been `tainted` (line 28).

The aspect considers any text widget to be tainted if its value was set by the application (lines 46-48). The pointcut `catchTainting` (line 38) is triggered at these points, marking a widget as not to be trusted. A taint flag (line 39) is added to the widget from an `inter-type declaration`, which is an AOP feature where new attributes can be added to a class. A potential solution for applications that want to set default text is discussed in Section 2.3.2.

The `DirectGUIAspect` includes some details which are framework-specific but would not be safe to be specified by an untrusted application developer. Since the `DirectGUIAspect` receives text input from some widget directly the `toURI` method is package protected. Essentially, a framework expert can “vouch” for the integrity of some widget implementation by providing a sub-aspect which binds the abstract pointcuts to that widget.

Concrete aspects for various widgets from the Swing and SWT toolkits are currently implemented. Unlike stream data where users could specify specific elements to be monitored, no way is provided for the user to specify which widgets are monitored for input. A user who makes use of the security framework is advised that any `URI` she enters may lead to the application having access to that `URI`.

**Mouse Input**

Not all input in a GUI is through a direct specification of `URI` text. In many cases, mouse-driven GUI widgets represent some `URI` resource indirectly. Because of this, proper support needs to be given to map a user’s interaction to the `URI` that some mouse-driven GUI widget may represent (what I call an indirect GUI widget and
introduced in Section 2.1.1).

Consider, for example, hyperlinks in a Web page. Each of those hyperlinks represent a URI to a site that the user wants to visit. Permission should not be granted to the various links automatically as that would allow the application to essentially crawl the Internet. However, if a user clicks on a link then the permission to connect to the URI should be granted.

Thus one must crosscut over these indirect GUI widgets to tie into the proper mechanism to pick up the interactions. The application can then be properly queried to find out what (if any) URI(s) are associated with the widget in question. But the user should also be presented with information as to what their action of clicking will do before they perform the action. This means that the user should be notified somehow.

When a user hovers over an indirect GUI widget that contains a URI, she is to be notified what URI would be granted permissions. If she does this and is told what will occur, that notification needs to be recorded so that the application does not change what URI has been entered into the widget between the time she hovered and when she clicked.

Interpreting a user’s intentions from mouse interactions can be more difficult than interpreting URI text input. This is because desktop applications typically present a user with representative references to URIs instead of URIs themselves. For example, the HTML anchor tag, `<a>`, is presented based on its content rather than the associated URI. Still, selecting the anchor should cause the associated URI to be retrieved. Essentially, this additional mapping of widgets to URIs must be dealt with in the framework.

Deletion of items stored in a configuration file is also typically done indirectly through the GUI. These GUI interactions tend to be application-specific. For this purpose, an aspect is provided to the developer which they can use to inform the security manager to delete some permissions. When the deletion of a permission has occurred, the user is notified through the Status Bar of the deletion. That way, if a user deletes something that she expects to lead to the removal of a permission, she can watch for the deletion notification. If the notification never occurs, she then knows the application is not acting in the way that is expected.

To monitor the selection of widgets that indirectly refer to URI resources, the
/** Aspect to handle indirect GUI input */
@Param T GUI widget to handle */
public abstract aspect IndirectGUIAspect<T> {

/** Pointcut to specify what join points represent use of the widget. */
protected abstract pointcut monitor();

/** Register a widget as being monitored. */
@Param widget widget to watch */
abstract void register(T widget);

/** Advice to register use of the widget. */
@Return widget now being watched */
after() returning (T widget): monitor() {register(widget);}

/** Introspect the GUI widget for the URI is contains */
@Param widget widget to be introspected
@Return the URI contained within the widget */
protected abstract String toURI(T widget);

String T.mappedURI = null;

/** When hovering, notify the user what permissions will be granted. */
@Param widget the widget being hovered over */
void hover(T widget) {
    widget.mappedURI = toURI(widget);
    SecurityManager.notify(widget.mappedURI);
}

/** Grant authority for the URI upon clicking as shown when hovering. */
@Param widget the widget that has been clicked */
void select(T widget) {
    String uri = widget.mappedURI;
    if (uri == null)
        uri = toURI(widget);
    SecurityManager.authorize(uri);
}

Figure 2.11: Abstract class for managing indirect GUI input, written by a policy expert
/**
   Aspect which grants authority through interactions through SWT’s list widget.
*/
public aspect ListAspect extends SWTAspect<List> {

    /**
     Start monitoring an SWT list widget when it is created.
     */
    protected pointcut monitor():
     call(List.new(..));

    /**
     Get the URI from the currently selected item in the list.
     */
    @param widget widget being used
    @return URI selected in the list
    protected String toURI(List widget) {
     return widget.getItem(widget.getFocusIndex());
    }
}

Figure 2.12: Implementation of IndirectGUIAspect for SWT’s List widget

IndirectGUIAspect aspect is used, as shown in Figure 2.11. This abstract aspect is parameterized by the type of the widget, for example, JList (Swing), List (SWT), etc. For the abstract authority aspect, two kinds of mouse events are monitored on these widgets: one to provide feedback to the user about the URI mapping and one to authorize mapped URIs. The monitor pointcut (line 7) is used to identify widget objects for monitoring. These widgets then have listeners added to them to catch these events.

The aspect includes an abstract method toURI (line 23) which is implemented by an application developer to provide the URI which some widget is mapped to. This mapping is application-specific so it must be provided by the developer of a specific application. Since one cannot assume the developer is trusted, special care is required in the handling of these widgets.

Mapping URIs to to a more representative name used by the application is achieved by querying the toURI method when a user places the mouse over some widget but before the widget is selected (lines 30-33). The URI retrieved from the developer’s implementation of the method is then presented to the user in the Status
Bar (as described in Section 2.2.2 and shown on line 32 of Figure 2.11). This way a user can review the mapping between the widget and a URI before they select the widget. If she finds no problem with the mapping she can select the widget and the URI is cleared by the authorizing aspect (lines 38-45). Using the Status Bar in this way is less intrusive than forcing a user to interact with a prompt after she has selected the widget. Browsers such as Mozilla and Internet Explorer use this technique. The aspect allows this technique to easily be added to any desktop application.

The abstract IndirectGUIAspect (Figure 2.11) is designed to reify one policy for granting resource access to a desktop application, based on mouse input. The policy includes the Status Bar feedback mechanism for end-users to evaluate which URIs will be accessed when a widget is clicked. This makes it safe to allow an untrusted desktop application developer to provide a widget to URI mapping through the protected toURI abstract method.

An example of a concrete IndirectGUIAspect implementation is shown in Figure 2.12. The aspect SWTAspect (not shown) extends from IndirectGUIAspect and provides an implementation of event listeners that make calls to the hover and select methods of IndirectGUIAspect. ListAspect then extends from SWTAspect (line 5) to provide List-specific implementation details. Here a developer maps a widget to a URI based on the item in a list which is the focus of mouse events (lines 19-21).

A similar approach can be used to “un-taint” programmatically set text when there is an associated submission button. If the application developer explicitly writes an authority aspect for a text GUI widget, they can then have the submission button show what permissions will be granted when the button is hovered over.

2.3.3 Deletion

When a resource is deleted, its permissions need to be removed. If the permissions were allowed to continue to exist then the POLP would no longer be met as un-needed permissions would still be available to the application. This can especially be a problem if the “deletion” didn’t destroy a resource but simply choose to no longer access it.
A security policy needs to be sound in order to properly protect the user from attack. From the perspective of the security framework, this means that false-positives for permissions must not be possible; permissions should never be accidentally granted. False negatives — when a permission is not granted that should be — should be eliminated if possible, but if they occur there should be a mechanism in place to handle them on-demand. By never accidentally granting improper permissions and allowing for needed permissions, the policy prevents an application from gaining permissions it should not have.

Soundness follows from 7 assertions, enforced by my framework, as described in the following paragraphs:

1. False positives cannot occur
2. False negatives may only occur if there is a mechanism to allow them to be rectified

3. User prompting prevents false negatives from being unresolvable

4. URIs entered into text widgets have proper permissions granted

5. Indirect GUI input only provides permissions if the user is made aware of through the Status Bar

6. XPath rules are applied to data coming into an application from disk and the network to grant permissions

7. Configuration files cannot gain permissions which did not exist at some point during program execution

(1, 2) The POLP, when fully implemented, prevents all false positives and false negatives. Based on the definition of soundness previously given, the POLP is sound along with supporting a stronger specification of security by the complete elimination of false negatives. Full implementation of the POLP is rare, though, as eliminating all false negatives can be difficult, which is why a sound implementation is considered acceptable.

(3) User prompting is a sound implementation of the POLP. No false positives are possible as a user is asked to clear all permissions first, preventing accidental permission granting. Prompting the user for all permissions also means that no needed permission will go unprovided, i.e., there are no false negatives.

(4) Thus the fundamental challenge was to minimize user prompting. The security framework monitors the GUI to detect when a user shows intentionality for accessing a resource. For obvious GUI widgets such as file dialog boxes, a framework expert is able to have the widget grant the proper permissions without the possibility of a false positive. For text input widgets the same is also possible as long as the user is told that any entered text can be interpreted as a URI.

(5) For indirect GUI input, the Status Bar prevents false positives when the application developer chooses to participate in the security policy. As long as a user takes time to evaluate whether the Status Bar is displaying a permission that should
be granted when (double-)clicking occurs on a GUI widget then false positives are prevented. User prompting, as always, prevents false negatives from occurring.

(6) When the security framework monitors disk and network I/O, XPath rules are used to decide what (if any) permissions should be granted based on the data coming into the application. The permission needed to access the disk or network in the first place would have not been granted if the user did not wish for it to occur and trust the resource data, so the resource supplying the data is considered trusted.

As data is being read from a trusted URI, XPath rules are applied to search for URIs that should have permissions granted to access them. If the data contains malicious or incorrect data which leads to false positives then the trust in the resource providing the data was misplaced. If an incorrect XPath rule led to a false positive then a mistake was made in the specification of the rule.

(7) Because configuration files are typically a special case of being an XML file which is both read and written to by the application, it must be monitored to make sure the application cannot unduly influence what permissions the configuration file will provide indirectly through monitoring. When the file is written the permissions the file would grant the next time it is verified to contain only permissions granted at some point during the current execution; otherwise the application could create arbitrary permissions for itself.

2.5 Case Study

For this case study I wanted to understand two properties of the approach better as well as measure the overhead. First, I wanted to know if the monitoring capabilities presented were sufficient to cover the implementation of realistic applications. It was anticipated that off-the-shelf applications may use some features that the approach would not be able to support. For example, it was known before work began on the case study that applications using the approach could not make use of native methods. Even though it is not the intention for the approach to be used as a way to automatically retrofit applications with security, it is useful to elucidate where some existing applications diverge from the methodology required by the approach to see whether standing developer practices would have to shift to accommodate the approach; this case study is not intended to test the hypothesis
of whether the security framework would work well for any randomly chosen off-the-shelf application. Second, I wanted to better understand the effort involved in applying the approach for a realistic application. By applying the approach myself, I hoped to report my experiences and some of the lessons that I learned.

I looked to select desktop applications that had no security in place but could benefit from restricting its permissions to access resources. Two such applications was evaluated: RSSOwl [73] and Lobo [5].

RSSOwl is a 46K Source Lines Of Code (SLOC) RSS reader written in Java. It makes use of the SWT widget toolkit, HTTP protocol, RDF and Atom XML formats, and an XML configuration file. Lobo is a 61K SLOC Web browser written in Java. It makes use of the Swing widget toolkit, HTTP protocol, and HTML format.

Coverage of each application’s features was estimated by using the GUI as a coverage criteria. Essentially each feature of the application was manually exercised that was induced by GUI input and noted what permissions were requested to perform the action by adding debugging code which outputted what permission was requested for what URI. Although it is believed that all features of the applications have been covered, it is possible I had somehow missed a feature that was not obviously accessible in the GUI. However, since any overlooked feature was not obviously accessible, it is probably not frequently needed by users. Therefore the fact that access control for the feature might have to be covered by explicit user prompting is not likely to be irritating for the end user.

2.5.1 RSSOwl

Table 2.1 shows the tallies of what kind of mechanism was needed to secure the features of RSSOwl tested. The first column, Authority, lists the name of authority aspects, user prompting, or traditional Java code access policy that was used to secure RSSOwl. For instance, the DirectGUIInput aspect from Figure 2.10 is listed because it is used in securing RSSOwl. These classifications and the results for each are discussed below.

The second column, Details, lists either a specialization of the authority aspect for a specific kind of widget, the XML format which is addressed by an XPath policy, or the number of cases the mechanism was used. Using DirectGUIInput
as an example again, two text widgets from SWT are listed as they needed to have an authority aspect written for them.

I now discuss each row of the table in turn. When referencing the table I start from the top going down, not counting the header row. Rows are counted based on the “authority” column.

<table>
<thead>
<tr>
<th>Authority</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Access Policy</td>
<td>1 use</td>
</tr>
<tr>
<td>DirectGUIAspect</td>
<td>swt.Text</td>
</tr>
<tr>
<td></td>
<td>swt.FileChooser</td>
</tr>
<tr>
<td>IndirectGUIAspect</td>
<td>swt.Table</td>
</tr>
<tr>
<td>DeletionAspect</td>
<td>2 uses</td>
</tr>
<tr>
<td>FileStreamAspect</td>
<td>RSSOwl XPath</td>
</tr>
<tr>
<td></td>
<td>OPML XPath</td>
</tr>
<tr>
<td></td>
<td>RSS XPath</td>
</tr>
<tr>
<td></td>
<td>Atom XPath</td>
</tr>
<tr>
<td>HttpStreamAspect</td>
<td>RSS XPath</td>
</tr>
<tr>
<td></td>
<td>Atom XPath</td>
</tr>
<tr>
<td>User Prompt</td>
<td>2 uses</td>
</tr>
</tbody>
</table>

Table 2.1: Mechanisms used to support RSSOwl security. © ACM 2009

**Code Access Policy**

The bootstrapping requirements of the JVM requires whitelisting certain things just to get the JVM to allow loading class files and get the application up and going. These various permissions tied to starting the JVM must be specified in a policy file [3]. With these basic permissions granted, application startup can continue as discussed in Section 2.5.1.

At startup RSSOwl connects to [www.rssowl.org](https://www.rssowl.org) to perform a version check to see if the user is running the newest version of the software (row 1). This URI is included by the application developer in a code access policy distributed with the application.
GUI I/O

Several GUI interactions in RSSOwl lead to the user directly entering a URI. For these situations the DirectGUIAspect (Figure 2.10) was used to monitor the interactions (row 2). A GUI interaction such as selecting File -> New -> Favorite pops up a dialog box that accepts a URI to a feed. The URI entered by the user should be accessible by RSSOwl without requiring any additional prompting of the user. The DirectGUIAspect was useful for 2 different SWT text widgets which has a combined 8 instantiations of the two widgets.

One unique thing about RSSOwl is that every GUI input that accepts a URI also accepts a file path. This leads to the need to clear both URIs and file read access at the same time for all GUI input where a URI can be entered. The same is not the case in reverse; there are certain GUI interactions (e.g., File -> Generate HTML) that only accepts a file path through a file chooser.

The IndirectGUIAspect (Figure 2.11) was used to define an authorizing context to inform users of the hosts for various news items in a feed (row 3). This indirectly includes the Status Bar informing the user what permissions will be granted. Aliasing was also used to map the feed title to the URI for the feed.

When a user deletes a feed, the permission to access that feed should be removed. A DeletionAspect was needed in two cases (row 4). Once for removing news feed categories and once for removing news feeds themselves.

Disk/Network I/O

RSSOwl uses an XML configuration file to specify the URIs for the feeds the user is subscribed to. Here the FileStreamAspect was equipped with authorizing contexts defined as XPath rules for the RSSOwl configuration file format (row 5). To create these rules the RSSOwl configuration file was inspected to understand which XML elements were used to encapsulate URIs that were needed by the application (Figure 2.5 contains the rule).

One must cut across disk I/O for configuration files in order to support the refreshing of all feeds on a set time interval. There is also the issue of using menu options such as Favorites -> Reload All Favorites which refreshes all feeds and thus operates just like when all feeds are refreshed during the initial
loading of the application. Regardless of the trigger, the feeds the user has specified during previous executions that have been stored to the configuration file require permissions to download the feeds to check for updates.

The second situation where disk I/O must be handled is when XML is read from outside the application’s configuration directory based on user input. This is exemplified by importing an Outline Processor Markup Language (OPML) file. OPML is an XML file format for storing directories of RSS feed providers. Similar to the previous example, the FileStreamAspect was used with OPML XPath rules (row 5). Since the semantics of the OPML file are standard, it was simple to create the rules which provided the necessary interpretation.

RSSOwl uses the network to download representative icons for the various feeds. The HttpStreamAspect was used to gather the authority needed to download the proper icons as specified in the various RSS and Atom formats (row 6).

For displaying the Web pages for new items, RSSOwl uses an SWT API\textsuperscript{11} to launch an external browser. I did not anticipate this use-case as it is not a traditional GUI widget used by most applications. Using the security framework, the developers of RSSOwl would need the framework to provide explicit support for this API in order to properly control its use. This support was not difficult to add, but would need to be added on a case-by-case basis.

User Prompting

Ignoring permissions needed for bootstrapping purposes that can easily be whitelisted (which are discussed in Section 2.5.1), the first step to startup for RSSOwl is determining what directory contains RSSOwl’s configuration data. RSSOwl uses an algorithm to create a URI for the directory, depending on the current machine’s operating system. On OS X it first looks at /Library/Preferences/.rssowl, followed by /Library/Preferences/rssowl (note the lack of a leading dot) as specified by the RSSOwlLoader.createWorkingDir method. The user is prompted to grant permission to read (and eventually write) to the directory (row 7). Since this path is determined at run-time — negating the possibility to specify the path in a code access policy file – and it does not depend on any input

\textsuperscript{11}The org.eclipse.swt.browser.Browser class.
— preventing the use of an authority aspect on some I/O — it had to be cleared through user prompting.

I felt this was reasonable since the application was going to determine on its own where to begin placing new files on the user’s machine. I think that in many cases the user would appreciate being notified of this fact instead of blindly allowing RSSOwl to select whatever directory it wanted to store files.

Granting the application access to its configuration directory then automatically clears permissions allowing for various bookkeeping steps such as writing to its configuration when something is changed through Edit -> Properties or View -> Customize Toolbar or denoting that a feed item has been read.

To prevent having to clear the same application directory at every launch, as it occurs after applying my approach, two options are available. One is to modify the user prompting code to add an option for storing cleared permissions somehow. This could either be done through a second whitelisting file or appending to the pre-existing one. Otherwise the user could manually add any permissions they so choose to the code access policy file. Help could be given by explaining to the user how to go about modifying the whitelist file to make the process as simple as possible.

The other use-case for user prompting during execution is when the application dynamically decides it needs a resource without any user interaction (row 7). A perfect case of this is Tools -> Search for Newsfeeds. This menu option takes a string that it passes to Google and searches for potential feeds for the string. This action leads to downloading various search results returned by Google. Because RSSOwl chooses what links to follow and not the user, there is no way to infer the intention of the user and thus what to grant permissions for. In this case, prompting the user is the only sane approach that does not require inherent knowledge of the application. Otherwise the interaction taken by the application needs to be specially handled.

Summary

As this work presents a new security policy for applications to follow, there is no direct comparison to use against pre-existing tools. But a comparison can be made
against how many permissions are handled by the framework instead of requiring the user to make a security decision. If the security framework had not been used, there would have been 96 requested permissions made of the Java security manager on behalf of RSSOwl. Add in 5 more prompts from the three situations discussed previously where the framework cannot alleviate the need for user prompting and you have 103 requested permissions. That is an over 93% reduction in permissions that the user would need to explicitly allow or deny because of the security framework’s ability to infer the intentionality of the user.

My framework came up short in three situations where prompting was required of the user. The two user prompting situations are application-specific and unavoidable. One of them would be intermittent based on using a specific application feature that would not come up in daily usage. The other would occur during every execution but could be avoided by adding the possible paths for the configuration file to the code access policy file.

The third shortcoming was my fault for not considering that a specific GUI toolkit widget would be needed. That would not have been an issue given enough time to go through and support as many GUI widgets as possible.

Excluding these three cases, all other application interactions between the user and RSSOwl were properly secured. This covered everything from adding new feeds, refreshing them, clicking on links, and all file system interactions. Considering that only one of the three shortcomings are systemic I would argue that the approach worked successfully for RSSOwl.

2.5.2 Lobo

Lobo is a Java-based Web browser written using Swing. In contrast to RSSOwl, Lobo has its own security manager. For the purposes of this case study, I disabled Lobo’s security manager as it did not manage resource access permissions but those to dynamically execute Java code only.

As an overview of my experience, Table 2.2 shows the tallies of what mechanisms were used to secure resource access for Lobo.
<table>
<thead>
<tr>
<th>Authority</th>
<th>Specialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Access Policy</td>
<td>5 uses</td>
</tr>
<tr>
<td>DirectGUIAspect</td>
<td>JFileChooser</td>
</tr>
<tr>
<td></td>
<td>JTextField</td>
</tr>
<tr>
<td>IndirectGUIAspect</td>
<td>cobra.HTMLLink</td>
</tr>
<tr>
<td></td>
<td>JMenuItem</td>
</tr>
<tr>
<td>HTTPStreamAspect</td>
<td>HTML XPath</td>
</tr>
<tr>
<td>FileStreamAspect</td>
<td>HTML XPath</td>
</tr>
<tr>
<td>DeletionAspect</td>
<td>1 use</td>
</tr>
</tbody>
</table>

| **Table 2.2:** Mechanisms used to support Lobo security. © ACM 2009 |

**Code Access Policy**

There are five specific uses of whitelisting for Lobo (row 1). The most important one is whitelisting the application directory. Since it is specified as `$\{user.home\}/.lobo` in the source code, the directory can be specified in the whitelist since system properties are supported by Java’s code access policy.

Three uses of using the whitelist were for the Page Services menu. The menu options use the current page as part of a query performed at three different websites (e.g., google.com). One can statically list those sites and thus allow the query to take place without requiring any prompting of the user.

Lastly, Help -> Project Home takes the user to the Lobo project home page. Much like with the menu option that takes the user to Google, whitelisting the Lobo project home page allows the initial connection to the site to occur.

**GUI I/O**

In Lobo, GUI I/O is used when a URI is specified in a traditional browser navigation bar. By monitoring what is entered in the bar, Lobo can be granted the permission to download the HTML for the page desired. Here the DirectGUIAspect was used and specialized for a Swing text field (row 2).

Alternatively, Web pages can be loaded from disk using the File -> Open -> File menu option. In this case the DirectGUIAspect was specialized for the Swing file chooser (row 2).
To support browsing behaviour through clicking on hyperlinks, \(<a>\) tags were monitored using the \texttt{IndirectGUIAspect} (row 3). Lobo did not include a status bar feature like traditional browsers so an aspect added this additional feature. The implementation originally assumed applications would only use the widgets provided by a GUI toolkit. Lobo uses a custom HTML renderer (named Cobra) which draws directly to the screen.\(^{12}\) To support this use-case in the security framework required adding lower-level support for Swing’s direct screen mapping. This could be useful for other applications if developed further for those with unique GUIs which are directly rendered to a window.

Finally, Lobo allows users to select bookmarked Web pages through menu items (as in traditional Web browsers). This indirection between the bookmark title and the bookmark URI was also handled by the \texttt{IndirectGUIAspect} similarly to the case for \(<a>\) tags (row 3).

\underline{Disk/Network I/O}

To properly display HTML downloaded from the Internet, certain tags need to have permission to access the URI the tag refers to. In some cases, images and scripts are loaded from a host name that is different from the host of the initial Web page. Since the user only authorized the download of the initial Web page, these auxiliary resources still need to be handled. Reading HTML files from both the network when they are initially accessed and the disk if the Web page is accessed from a cache must both be supported.

This can be handled through the use of XPath rules which automatically clear the hostname for the image or script URI (row 4 for network access, row 5 for disk cache access). If the user is careful to only visit trusted sites, then it is safe to clear connections to these image and script resources. I did not attempt to help the user choose which sites should be trusted or not as this kind of decision making is outside the scope of the security mechanism implementation.

\(^{12}\text{Direct screen drawing is done through} \texttt{java.awt.Graphics}.\)
<table>
<thead>
<tr>
<th>Feed</th>
<th>Size</th>
<th>Items</th>
<th>Original (ms/std. dev.)</th>
<th>Aspectized</th>
<th>Parsing</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNET</td>
<td>14 K</td>
<td>25</td>
<td>305/7</td>
<td>315/3</td>
<td>19/08</td>
</tr>
<tr>
<td>mozillaZine</td>
<td>38 K</td>
<td>12</td>
<td>143/2</td>
<td>149/9</td>
<td>20/08</td>
</tr>
<tr>
<td>MajorGeeks</td>
<td>6 K</td>
<td>20</td>
<td>97/3</td>
<td>98/2</td>
<td>8/02</td>
</tr>
<tr>
<td>freshmeat</td>
<td>103 K</td>
<td>68</td>
<td>256/2</td>
<td>303/3</td>
<td>79/46</td>
</tr>
<tr>
<td>PC Magazine</td>
<td>16 K</td>
<td>10</td>
<td>87/4</td>
<td>94/4</td>
<td>9/04</td>
</tr>
<tr>
<td>BetaNews</td>
<td>9 K</td>
<td>15</td>
<td>112/6</td>
<td>105/7</td>
<td>11/06</td>
</tr>
<tr>
<td>Slashdot</td>
<td>34 K</td>
<td>15</td>
<td>129/6</td>
<td>139/6</td>
<td>12/01</td>
</tr>
<tr>
<td>SnapFiles</td>
<td>18 K</td>
<td>48</td>
<td>133/3</td>
<td>135/7</td>
<td>7/01</td>
</tr>
</tbody>
</table>

Table 2.3: Overhead in RSSOwl from parsing feeds for URIs. © ACM 2009

Deletion

Lobo had a single use for the deletion of permissions to handle the removal of a search engine used to perform searches (row 6). If a search engine is removed from the list then the permissions to access it are revoked unless the user explicitly visits the website of the search engine.

User Prompting

Lobo had no need for user prompting. I was able to provide all needed permissions from monitoring input or from the code access policy.

Summary

Lobo required 58 permissions, all of which the security framework was able to handle. The framework only came up short in one instance for Lobo by not supporting its custom HTML renderer through the low-level graphics renderer. Much like one of the shortcomings in RSSOwl, if I had the time to secure every GUI widget in Swing then I would have not have any problems securing Lobo with my framework.
2.5.3 Performance

I wanted to test application performance with and without the monitoring mechanism to see if the added overhead of the security framework negatively impacted performance to the point to make an application unusable. The most performance-intensive part of the framework is certainly the additional parsing of stream input. I chose to test RSSOwl feed loading with and without authority aspects as it involved parsing, was easy to measure, and provided a variety of real-world test data.

Initially I planned to report performance numbers from Lobo, however I quickly realized that rendering HTML pages is more CPU-intensive than loading news feeds, with CPU costs masking the overhead introduced by the authority aspects, so I chose to study RSSOwl.

Table 2.3 shows the performance numbers for RSSOwl. The time that was measured is from when a feed is selected to when the news items are displayed by RSSOwl. The standard deviation comes from measuring each feed five times. All times were run on a 1.6 GHz Intel Core 2 with 3 GB of memory on Windows Vista Home using Java 6. All timings represent how long it took RSSOwl to load a news feed. The news feeds to test were chosen somewhat randomly as they are defaults in the Software category of RSSOwl. Each feed represents the news from the host indicated on September, 29 2008. The news feeds were saved to file and loaded from disk. This was done to remove effects of network latency from the measurements.

The original times are using an unmodified copy of RSSOwl. The aspectized times are running RSSOwl with my framework. The FileStreamAspect is used to monitor the feeds as they are loaded. The parsing times show just the time spent parsing the data by the content handler and not the rendering of the news feed. The aspectized times include the parsing times, but since the parsing is done in parallel the overhead does not lead to a linear increase in time.

Typically the performance overhead is quite small, and 16% slower in the worst case. Since RSSOwl is I/O-bound and typically is waiting for user input, even if the slowdown applied uniformly across all test feeds it would still most likely be negligible overhead. But most of the timings are very close (and in one case, better) where the difference is less than 10 ms. Note that because of the standard deviation
it would be wrong to conclude the aspectized version was faster in any case.

While I chose not to evaluate the memory consumption directly, there is some memory usage by the security framework. As the security manager must retain all of the specific permissions that have been granted along with potential permissions and aliases, there is some increased memory usage by the framework. To decide if the memory usage of my approach is reasonable, I considered a back-of-the-envelope scenario involving Web pages. Consider a user who visits 1,000 Web pages, each containing 100 hyperlinks with URIs containing hostnames that have a character length of 255 characters (the maximum allowed by DNS [30]) and an overall URI length of 300 characters, and link text that is 100 characters long. Creating an alias between the link text and the URI as if the user hovered on every link would require storing each URI and alias text once. If the user also visited every link, that would lead to each URI being stored along with the permission to connect to the URI. Working with Java’s internal string encoding of UTF-16 [4], 80 MB would be used to store the characters mapping link text to the URI and 60 MB is used to store the URIs: 140 MB total of transient memory which is not carried over to the next execution of the application. It could be possible to allow the strings to be garbage collected once a page is no longer visible to the user, but I have not considered this in my implementation because of the added complexity.

2.5.4 Summary

Use of the framework leads to measurable benefits for security when viewed from a sound security perspective. Viewing a user prompting security policy as the base security policy to improve upon — as it provides complete security coverage — shows that both applications see an improvement. RSSOwl has 96 out of 103 user prompts implicitly handled by the security framework; an over 93% reduction in prompts. For Lobo their was a 100% reduction of the 58 prompts for permissions.

The performance impact is also minimal. With a worst-case of a 16% performance impact, using the security framework would not be a severe detriment to an application’s performance. Memory usage would also not be a problem for a desktop application as a worst-case scenario of 1,000 Web pages with 100 links each leads to roughly a 140 MB increase in memory usage.
Chapter 3

Automatic Object Persistence in JavaScript

In order to motivate automatic object persistence in JavaScript, consider a scenario where a developer would like to deploy an offline Web application to support a Web-based e-mail client (initially mentioned in Section 1.2.1). Without offline support, a user would not be able to access their inbox while disconnected from the Internet. Furthermore, a user would not be able to compose new e-mail messages while disconnected. This is because modern Web applications make use of JavaScript APIs for the management of data such as individual e-mails; JavaScript does not inherently contain a storage resource. This has traditionally caused developers to rely upon making remote calls to a server on the Internet using XHR (an RPC-like mechanism) which imposes latency upon the application to fetch or store data.

Persistence of data for offline use by JavaScript is now being widely deployed in all major browsers because of recent advancements in HTML5 [49]. Some support is provided for both simple persistent hash tables [48] and also for SQL-style databases [47]. However, none of the existing storage mechanisms directly support either structured object data or language-integrated automatic persistence. This means that developers still must manage several difficult problems related to storage by themselves.

First, developers must determine how and when to save data from heap-allocated
JavaScript objects to local browser storage. This must be done carefully to balance several tradeoffs. If changes are saved too frequently, the overhead of each save action could degrade performance. Also, since JavaScript is single threaded, writing a single large batch of object changes will prevent the application from servicing any other event-handlers, e.g., those which respond to user actions or timers which control activities such as page animations. For example, on the one hand, if every character entered by a user while composing a new e-mail triggered an expensive saving of that e-mail then the act of saving could cause visible delays in the user experience. However, on the other hand, leaving changes unsaved for an extended period increases the chances of data loss; no one wants that e-mail draft to be lost because the browser crashed.

To deal with this problem, I have investigated the use of persistence by reachability [50] for JavaScript. In this approach, the object heap is divided into transient and persistent sub-graphs. Which sub-graph an object belongs to is based on whether the object can be reached by traversing the object sub-graph from a designated “root” object. If the object can be reached from the root object then it is considered persistent, otherwise it is considered a transient object. My runtime persistence library1 consists of several time-scheduled event handlers which perform several persistence-related tasks periodically. In this way, a developer could use JavaScript data objects to build a persistent data structure representing a user’s e-mail folder. When a new e-mail object is added to the data structure, my library would ensure the new e-mail is automatically saved to local storage in such a way as to not degrade application response, even when dealing with thousands of objects.

Second, a developer must determine how and when to synchronize offline data stored in a browser with a server copy of the data. This is difficult because copies of data may become inconsistent if they are made available to more than one browser session. For example, an e-mail draft might be copied to two different browsers that are used by the same user. If the user edits one of the copies offline, it may become inconsistent.

To deal with this problem, I have investigated and developed an application

1http://bcannon-www2010.googlecode.com
library for automated persistence at the granularity of individual objects. This allows the browser to communicate updates of data for individual objects which map directly to the object model created by the application developer. Conflicts between object versions are detected automatically which triggers a callback\(^2\) which can resolve the conflict. The developer simply needs to implement the callback to apply any semantically relevant resolution strategy. For example, an e-mail draft composed while offline would automatically be synchronized to a server when an Internet connection is re-established. If that same e-mail draft was in conflict with other edits made while the browser was offline the user would be notified of the conflict in order to resolve it.

Lastly, all of this done in a transparent fashion in regards to the developer not needing to explicitly request persistence of specific objects. As demonstrated by Figure 1.2, the current storage resources provided by browsers do not provide a mechanism which directly maps an object that a developer naturally works with to a format which can be persisted by the storage resource. By implementing object persistence, this alleviates the need for the developer to develop a mapping from data objects to the format of the storage resource, e.g., mapping an object representation to fit into a SQL database.

While previous work on object-oriented databases and distributed object languages had similar motivation, this work has dealt with different technical issues that are unique to the browser-embedded JavaScript programming environment. I have provided specific mechanisms to work around JavaScript’s lack of threading, primitive support for language extensibility, and made sure that support does not require the use of any features that are not part of browser standards.

To prove the created library is useful, an evaluation has been performed (Section 3.6). Micro-benchmarks have been run to show that the persistence library can handle persisting over 1,000 objects without perceptible slowdown. Benchmarking has also been done for a real-world scenario of querying a live Web service and persisting the returned results. To measure the usability of the persistence library, I ported a Web application that already stored data locally in the browser and synchronized to a remote server over to the persistence library to see if it would lead

\(^2\)A function which is executed when a specified event occurs.
Figure 3.1: Architecture diagram for automating the persistence of JavaScript objects

to a simplification of application logic. Lastly, I used the persistence library with a JavaScript GUI toolkit to verify that the library could be integrated with other libraries seamlessly.

3.1 Overview

To solve the problem of persisting objects in the browser easily I have implemented and evaluated an automated object persistence library for JavaScript. The architecture of the library is shown in Figure 3.1. The approach taken involves detecting mutations of persistent objects, serializing the objects to local storage, and then synchronizing the serialized form of the persistent objects to a server on the internet which includes pulling updates performed by other browsers to the persistent objects.

Two approaches to detecting mutations of persistent objects are used (Section 3.2). First, a maintenance task which executes on a set time interval detects *dynamic property additions* by scanning all of the persistent objects in use during the current execution of the browser for changes to the objects (Section 3.2.2). Second, *Accessors* are attached to persistent objects which detect mutations at the
time of assignment (Section 3.2.1). These two separate approaches for detecting mutations are required to work around shortcomings in JavaScript.

The detection of a mutation leads to two actions occurring locally in the browser (Section 3.3). The first is that the mutated object is serialized into a format called JavaScript Object Notation (JSON) which is subsequently put into local storage, which is a key/value store provided by the browser and exposed through JavaScript. Storing mutated objects makes sure that if the browser crashes the state of the persistent object is stored for later retrieval. The second action after detecting a mutation is to add any new objects to the live object set, which is the set of all persistent objects that have been read from or written to during the current application execution.

Periodically all mutations of objects are pushed to a server on the Internet (Section 3.4). The browser sends the new serialized form of any persistent objects that have mutated since the last time the browser and server communicated. The server responds with any updates to objects that the browser is lacking. In order to not overwhelm the browser with thousands of persistent objects when it has never run the Web application before, a “cold” synchronization is performed to gradually pull all persistent objects from the server to the browser. Once the browser matches the state on the server, “hot” synchronization is used so that only the persistent objects changed since the last complete synchronization with the server are sent, saving bandwidth and processing time.

Normal reading and writing of persistent objects is allowed while offline so that the Web application can continue to work as normal. Upon reconnection with the server any conflicts created while offline are detected by the server and sent to the browser so the Web application has an opportunity to resolve the conflict.

Before persisting objects both locally and remotely, one must know exactly what objects need to be persisted. This involves detecting when a persistent object has been mutated.

### 3.2 Detecting Mutations

In any application, data naturally falls into two classes: transient and persistent. Transient data is applicable only for the lifetime of an execution session, whereas
persistent data must survive across individual application execution sessions. In object-oriented languages, object persistence is a technique for automatically binding object instances to secondary storage, making them persistent. This allows developers to work with objects as they normally do — by getting/setting properties and calling methods on objects — while having persistence of certain objects managed for them.

Even when using an object persistence library, developers still must apply their knowledge of application semantics to classify objects as transient or persistent. *Persistence by reachability* has often been advocated as providing the simplest solution for classification [50]. In this approach, the object heap is divided into transient and persistent sub-graphs. A persistent “root” object is made available to developers and any object that can be reached transitively from this root through the object sub-graph is considered to be persistent [50]. My library automatically makes a special root object available, `persist.root`, that is attached to the global `window` object in JavaScript. To support this notion of persistence, the persistence library needs to have some way of monitoring the object heap to know when objects are made (un)reachable.

### 3.2.1 Detecting New Objects by Reachability in JavaScript

If I allowed myself to modify a JavaScript interpreter directly, the problem of detecting if an object is reachable from the root object would be straightforward. But as the goal is to develop a library that does not require browser modification, that option is unavailable to me. Fortunately there is some support in JavaScript to provide a subset of what is required to detect reachability. In JavaScript, special functions called *accessors* (see Figure 3.1) can be used which are called transparently when a specific object property is read or written. One can *bind* individual accessors to a property for assignment/writing and a separate accessor for accessing/reading. When the property is written to, my bound write accessor records that the object was mutated in a log and saves the new data for the object. My read accessor is then able to load the stored data as needed. By masking the function call behind standard property manipulation syntax the object’s interface remains consistent and seemingly unchanged to the developer while its behaviour changes.
to that of a function call. It also allows for the immediate detection of changes to a persistent object so that the possible loss of data is minimized to only the amount of time it takes the write accessor to process the mutation (for certain cases of dynamic properties this is not true, as discussed below).

Accessors allow for the detection of some mutations implicitly from the fact that an assignment has taken place. Still there are two more related problems to consider when using accessors. First, saving the new state of the application when a mutation is detected could take some time. Since JavaScript is not multi-threaded, this process will block any other application functions from executing. This is not desired as having property assignments suddenly take a significant amount of time would significantly degrade the performance of any data-intensive application. This problem is considered in Section 3.3.

Second, JavaScript is a dynamic programming language, which means one must consider not only changes to existing object properties but also the dynamic addition of properties. While detecting mutations on properties which already have the write accessor set is instantaneous, the case for when a property is written to which lacks a write accessor — i.e., a new property — must be handled.

The easiest solution, if it were possible, for detecting new properties on persistent objects would be to have the equivalent of a generic write accessor for the whole object. Then any property change, for new or pre-existing properties alike, would be detected instantly. For some languages, such as Python and its `setattr` method, this is actually possible. Unfortunately JavaScript lacks an equivalent object-level hook for a write accessor, so a custom approach must be used.

### 3.2.2 Detecting Dynamic Property Additions

When a persistent object is loaded, it is added to the “live” object set (see Figure 3.1). The set represents all persistent objects which have been loaded during this execution, which is a subset of all persistent objects. Since the object set represents all objects that have been loaded during the current execution, it also represents the set of persistent objects that may have mutated. Having this subset of persistent objects allows operations pertaining to object mutation to restrict their
function detectNewProperties(object) {
    var foundNew = false;
    for (property in object) {
        if (isNewProperty(object, property)) {
            foundNew = true;
            if (object[property] instanceof Object) {
                var newObj = object[property]; 
                registerObject(newObj);
                persistObject(newObj);
                log(newObj, "created");
                detectNewProperties(newObj);
            }
            addAccessor(object, property);
        }
    }
    if (foundNew) {
        persistObject(object);
        log(object, "updated");
    }
}

Figure 3.2: Pseudocode for detecting new properties and persisting any new parts of the object graph.

operations to only pertinent objects, saving valuable processing time.

Minimizing the amount of work performed is critical during stabilization [50]. When any persistent object has a property that is not being persisted, the live object set is considered unstable. Stabilization persists previously unknown data, making the live object set stable again.

During stabilization a scheduled dynamic property addition task (see Figure 3.1) is used which iterates over the live object set to search for new properties (shown in Figure 3.2). The task is simply a JavaScript function running at a developer-specified interval set by calling window.setTimeout on the function. As JavaScript is single-threaded, the concurrency model guarantees that once the stabilization task is finished that the entire live object set is stable.

The stabilization task must be done efficiently so that it does not block the main application from executing its own event handlers because of JavaScript’s concurrency model. To prevent blocking the main thread too long, the stabilization task can be paused after a certain amount of time has elapsed to prevent the locking up of the user interface. By setting a small enough interval between iterations over the live object set to stabilize it, the window where data mutation could be lost
by a browser crash can be minimized while not impacting application responsiveness. The maximum interval time the stabilization task should take at any one time is influenced by overall performance of the task, so this issue is explored in the evaluation in Section 3.6.

The live object set is periodically iterated over and each object is processed by the function detectNewProperties shown in Figure 3.2. This function iterates over each property in an object to see if it already has accessors bound to it (lines 3-15), initially assuming that there are no new properties (reflected through the foundNew variable). For each property on the object being processed, a check is performed to see if the property is new (line 4). If the property is new according to the library, the object being processed is marked as mutated by assigning true to foundNew (line 5). If the new property points to an object itself (line 6) it is considered a new object that must be persisted as it was reached through another persistent object. The new object must be registered with the library (line 8), persisted for safe keeping (line 9), logged as new for eventual sending to a server on the internet (line 10), and then itself checked for other new objects (line 11).

3.3 Persisting Data Locally

Stabilization from searching for new properties along with mutations detected by accessors provides the knowledge of what needs to be persistent locally in the browser. The next step for implementing object persistence is to figure out how to store the object data in the browser. To do so requires both a persistent data store in the Web browser as well as a serialization format for JavaScript objects.

For storing objects, the persistent key/value store called localStorage (referred to as “local storage” in Figure 3.1) that is being standardized as the W3C Web Storage standard is used [48]. A Globally Unique Identifier (GUID) is assigned to each persistent object so that any persistent object can be looked up in localStorage by its GUID [14, 15, 60]. Having a globally unique way to identify an object is also useful when having to synchronize objects between the browser and the server. JavaScript has no built-in notion of GUIDs like some languages (e.g., Java’s System.identityHashCode), and so one was created using the string representation of two calls to Math.random. This gives a high
enough entropy (roughly $10^{30}$ possible values) to not have GUID clashes within the same application.

Saving an object to storage also requires that any objects referenced by the object are saved. This is currently not handled by any existing JavaScript standard. So, when an object is saved to storage, a traversal of the object is performed to flatten the transitive graph of object references into a set of objects. The traversal transforms each object property that holds an internal JavaScript reference to a GUID.

For serialization the standard JavaScript Object Notation (JSON) is relied upon to represent each individual object after the graph has been flattened. This promotes interoperability with any components on the synchronization server, discussed in Section 3.4. Note that while there is already standard browser support for serializing JavaScript objects using a JSON API, this API does not support cycles in an object graph; only trees are supported. Additionally, a serialized JSON object tree is represented by a single string, which does not allow for indexing, retrieval, or updates at the granularity of individual objects. Using GUID references isolates objects in the object graph. This is why when an object is serialized all references to other persistent objects are replaced with their GUIDs instead of serializing the other persistent objects in-place.

A side-effect of introducing the concept of GUIDs as references is it allows for the lazy loading of persistent objects from local storage. When a property on a persistent object is accessed, its read accessor is triggered. If the live object set contains the object referenced it is returned from the set immediately. However, if the object has not been loaded yet, and thus is not in the live object set, the GUID is used to load the object from local storage and has its properties bound to accessors before caching the object in the live object set and returning the object.

The consequence of this strategy is that if a persistent object is never used during an execution of the browser it is not loaded from storage, saving the cost of loading it from disk and deserializing it. This allows the accessors to act as a read barrier to make sure that an object gets loaded from whatever storage location is the fastest (memory, the browser, or the server). It also speeds up start-up by avoiding having to wait for all persistent objects to be loaded from local storage before work can begin as only the root persistent object must be loaded to allow
loading other objects to begin.

### 3.3.1 Garbage Collection

Ideally when a persistent object is deleted or made to be no longer reachable from the root persistent object, it would be best to receive immediate notification of property deletions. In some languages, it is possible to use a deletion accessor or tie into the garbage collector to discover that an object has been deleted [50]. Unfortunately JavaScript provides no mechanisms for notification when an object is deleted.

In order to know when a persistent object is no longer reachable from the root persistent object a persistent garbage collector task at the application level has been implemented (see Figure 3.1). This is reasonable as this garbage collector is used to remove dead objects from secondary storage, not from main memory, so it does not need to execute frequently and therefore incurs minor overhead. While also clearing out dead objects from local storage, the garbage collector simultaneously removes dead objects from the live object set to prevent including them during stabilization.

A mark-and-sweep algorithm is used to implement the garbage collector as shown in Figure 3.3. Initially the live object set is cloned (line 1) so that it can be set back to this cloned value after the traversal as the garbage collection step eventually loads all objects into memory (this is discussed in detail below). A set of all persistent objects, referenced by `GUID`, is also created to represent all objects as initially “dead” (line 2). A breadth-first search (lines 7-11) starting at the root persistent object, `persist.root` (line 6), is used to find all live objects, removing every found object from the dead object set (line 13).

The traversal of the persistent object graph does lead to all persistent objects temporarily being loaded. Obviously this is unfortunate as it does raise the memory pressure on the application. But once the traversal is done the objects loaded explicitly for the garbage collection can be removed from memory safely and the live object set is set back to its initial value before the garbage collection traversal began.

The persistent objects found to be dead are then cleared from local storage.
Figure 3.3: Pseudocode for the garbage collection step, using set notation (lines 14-16). As the dead object set contained the GUIDs of the dead objects clearing them from local storage is quick. The dead objects are also removed from the live object set, if applicable (line 16).

3.4 Persisting Data Remotely

When designing a distributed system that allows for offline use, two approaches can be used: optimistic or pessimistic use of the data while offline [57]. In an optimistic updating scheme you allow for offline mutation of data, with the hope that when it comes time to synchronize the data you will be able to do so without conflicts. A pessimistic scheme is the reverse of optimistic updating: assuming that the reconnection and eventual data resolution will lead to conflicts with changes made while offline, you do not allow a user to make any mutations while offline. The reading of data while offline is allowed.

To support the user in doing work offline, you need to use an optimistic approach. For those instances where the server detects a conflict during an update (discussed in Section 3.4.2), the tradition of letting the application handle how to
This allows developers to create Web applications where data is shared between users but are still able to use automated persistence; either the developer structures the objects such that only a single user edits any one object at a time or must develop their own approach to resolving conflicts.

Supporting persistent updates at the level of individual objects is difficult in a browser because low-level access to the JavaScript interpreter is not available. Other object persistence systems integrate into a compiler or interpreter, giving them access to the underlying object representation at a primitive level [16, 29, 50, 77, 91]. In my case I have to perform similar operations at the application level. However as described below, this is necessary to support efficient pushing and pulling of fine-grained updates for synchronization with a server.

Communicating updates to a server requires two parts. First, one must know what to send (Section 3.4.1). A log is used that is actively maintained as persistent objects are changed. Once it is known what must be sent to the server, that information needs to be communicated (Section 3.4.2). The case of a browser that has never communicated with the server before or has its data heavily skewed from the server’s master copy is explicitly handled; I call this “cold” synchronization. A different update strategy can also be used that is tailored to the case when the browser is very close to matching the state the server is storing, known as “hot” synchronization.

### 3.4.1 Browser-Side Logging

For each mutated object, its GUID, what operation triggered the mutation (creation, update, or deletion of an object), the JSON stored in local storage for the object, and the object’s group labels (a form of index, discussed in Section 3.4.2) are logged. Each log entry is stored individually in local storage so that the log can be reconstructed if the browser is closed before the log is sent to the server. The server is updated in batches — which requires a log — instead of after each individual mutation to prevent having many mutations in a short period of time (e.g., mutations in a loop) triggering a large number of network connections being created.

The triggering of adding entries to the log is handled by either write accessors
or the stabilization task. As mentioned in Section 3.2, write accessors are able to detect mutations immediately, which allows the window of exposure to a data loss to be the time it takes for the write accessor to execute. For the stabilization task, the window of vulnerability is the time interval between executions and how long it takes to execute the stabilization task. But the log can be viewed as an optimization to speed up synchronizing with a server. The log does not store the serialized form of the object as that is inherently done already through the persistence of an object and would lead to duplication of work. Thus it is more important to get an object’s serialized form into local storage so that the object’s new state is saved rather than record the fact it mutated.

3.4.2 Synchronization With A Server

While the client-side log records what needs to be pushed to the server, it does not tell what needs to be pulled from the server. In the case of updating the local copy of the persistent object graph I consider two situations. One is where the local copy is “cold”; either the browser is loading a Web application for the first time or has been disconnected for quite some time. The other situation is when the local copy of the object graph is “hot”; the difference between what the browser has and what the server has is minimal.

Consider an e-mail Web application that synchronizes with a server. Cold synchronization comes into effect when a new browser is used to load the Web application when the user already has an extensive backlog of e-mails stored on the server. The user wants e-mail that is in their inbox to be made available first so as to not have to wait to start interacting with the Web application. But if they do decide to access older e-mails they will want those to be made available as well with as little waiting as possible. This also applies if the user has been working at their desktop for weeks and then ends up needing to take their laptop on a business trip. The data in the browser of the laptop could be very stale, almost to point of having the state of the persistent objects being completely invalid.

The common case, though, for a Web application like an e-mail client will be that of hot synchronization. Personal experience suggests that users typically use the same browsers regularly, minimizing any skew between the data stored in the
browser and that kept on the server. This means that there is less data to send and worrying about priority of objects just adds computational overhead.

All of this needs to be balanced for the unique characteristics of working over the Internet. As it is not a private Wide Area Network (WAN), the Internet has greater varying levels of bandwidth and latency between a browser and a server as the environment is under less control of the user or service provider. There is also no guarantee of a stable connection (e.g., a data connection on a mobile phone), which leads to needing to plan for an Internet connection being absent for an unknown length of time.

These uncertainties lead to the need to cache aggressively in the browser by caching all objects. While previous research involving object persistence has considered offline access, specifically the Thor object persistence database system [61, 62], it has been in a WAN environment where performance characteristics of the network are more controlled. This led to caching being motivated for performance reasons instead of for offline access to data [41]. In Thor’s case, there is the concept of hoarding, as introduced by the Coda file system [57]. Thor allows a user to specify queries that would be run to decide what objects to cache locally (Coda uses hoarding to define the priority of objects to cache when storage space is limited). That way, when faced with an unexpected disconnection from the servers, critical objects happen to be still available. But this is driven more for performance reasons; if an object changes on the server, it notifies all clients to evict the object from their cache without sending the new object. A simple form of hoarding is supported by my library, using a concept of grouping (discussed with cold synchronization), but I do not need a complex version as all objects are automatically synchronized instead of only ones specified by a query or objects that just happen to have been requested by the browser.

Working within the browser also prevents taking the approach by Thor because of how browsers communicate with the server. Since the server cannot instantiate communication with the browser, reliance on notification of stale objects is not possible; XHR allows only for pull-based communication from the server as initiated by the browser. Doing this also allows for a stateless, connectionless server design which enhances scalability by eliminating bookkeeping requirements on the server for each browser that has data stored.
“Hot” Synchronization

If the churn rate on data mutation is low then it is not hard for the browser to stay in sync with the server and is in a position to use “hot” synchronization. When the browser is synchronized to a specific point in time with the server then the browser is a “snapshot” of the server [18].

When it comes time to sync with the server (see Figure 3.1), the browser sends the log of mutated objects (Section 3.4.1), the new serialized form of each persistent object referenced in the log, and the snapshot timestamp of the last synchronization as determined by the server (the hotSyncBrowser function in Figure 3.4, lines 11-20). The browser leaves the connection open, waiting for a response from the server.

On the server’s end (hotSyncServer in Figure 3.4; lines 35-49), each mutated object is to be stored if no conflicts exist. For each object, the server stores at least an object’s GUID, JSON representation, timestamp of the last mutation, and what groups the object belongs to (line 43, calling updateObject on lines 1-8).

As the server processes the log sent by the browser (lines 39-44), each object is checked to see if there is a conflict with the proposed update from the browser (line 40). For newly created objects, a conflict occurs if the object already exists. For updated objects, a conflict occurs if the timestamp on the object is newer than the snapshot timestamp the browser sent with the log. A deletion of an object is in conflict if the object has mutated since the last synchronization [9]. By committing the update immediately, updates follow “first-committer-wins” semantics like snapshot isolation in databases [18].

Preferring the version of an object in the server over the browser provides master copy semantics [11, 12]. This simplifies semantics for synchronizing as it allows for conflict detection to be consistent with a single point. It also allows for using the server’s clock as the snapshot timestamp provider.

As a return value sent to the browser (line 20), the server sends all objects (and their respective details) whose timestamp is newer than the browser’s snapshot timestamp (line 46). An array of all objects’ GUIDs who were found in conflict are also sent along, including the master copy version of the objects as found on the server (line 48).

81
function updateObject ( guid , json ) {
    localStorage.setItem ( guid , json ) ;

    if ( guid in liveObjectSet )
        updateObjectInPlace ( guid , json ) ;

    notifyAboutUpdate ( guid ) ;
}

function hotSyncBrowser () {
    var lastSync = getLastSyncWithServer () ;
    var log = getLog () ;
    var changes = [] ;

    for ( guid , action in log ) {
        var json = localStorage.getItem ( guid ) ;
        changes.push ( [ guid , json , action ] ) ;
    }

    var server_update = hotSyncServer ( changes , lastSync ) ;

    var updatedObjects = server_update[0] ;
    var newTimestamp = server_update[1] ;
    var conflicts = server_update[2] ;

    for ( guid , json in updatedObjects )
        updateObject ( guid , json ) ;

    for ( guid in conflicts ) {
        var oldState = getOldState ( gui , changes ) ;
        notifyAboutConflict ( guid , liveObjectSet[guid] , oldState ) ;
    }
}

function hotSyncServer ( changes , browserLastSync ) {
    var conflicts = [] ;
    var timestamp = getCurrentTime () ;

    for ( guid , json , action in changes ) {
        if ( detectConflict ( guid , action ) )
            conflicts.push ( guid ) ;
        else
            updateObject ( guid , json , action , timestamp ) ;
    }

    var updatedObjects = getMutatedObjects ( browserLastSync ) ;

    return [ updatedObjects , timestamp , conflicts ] ;
}

Figure 3.4: Pseudo-code for hot synchronization between the browser and the server (note on syntax: [x, y] represents creating a list consisting of x and y; for(x, y in z) represents iterating through z which is a sequence of two-item lists)
Once the response is received by the browser for the log it sent over, the browser processes the data (lines 22-32). First, each object that the server sent over as mutated is processed. All updates are stored to localStorage immediately (line 2 in updateObject). For each updated object a callback is optionally called so the application is aware of any changes that have occurred (lines 6). If the object is in the live object set, it is updated in-place automatically by deleting all properties on the pre-existing object and then individually adding all properties as found in the version of the persistent object sent over from the server (line 4-5, details not shown). By not creating a new persistent object this allows pre-existing references held to the object to continue to work and use the new state.

The hollowing out of an object by deleting all of its properties and then rebuilding it from scratch is done for two reasons. One is that because mutations are not tracked at the property level, there is no way to know exactly which ones need an update. While this could theoretically be done, it would complicate the implementation along with adding more storage pressure. Two, trying to infer what has changed through value comparison is potentially costly. Since the size of the values being referred to by properties could be arbitrarily large, a comparison could take longer than simply replacing the property itself. Taking the simple approach of completely rebuilding the object keeps the semantics simple and does not try to make assumptions about how an application’s objects might influence performance.

Technically it should be possible to shift to the granularity of properties instead of objects so that only changed properties are updated, but there are costs. The server would need to have individual entries for each mutated property which greatly increases the amount of queries on the datastore used to both store and retrieve what state an object’s properties are in when a browser is synchronizing. When updating entries in the datastore when synchronizing, most likely a master entry per object would need to be updated along with the individual property entry. As for gathering updates to send to the browser, only the latest value would need to be stored and not every value that has been assigned to the property. But in order to gather all updated properties of an object would require pulling many more entries from the datastore rather than the single object entry. In other words, because bookkeeping becomes more complex, computational costs also increase.
Next, all conflicts are processed (lines 29-32). For each conflict, a callback is called which passes in the state of the object as found on the server and the conflicted state the browser originally sent to the server (line 31). This gives the Web application a chance to resolve conflicts in an application-specific way as the object will have been updated to the state as found on the server in-place [40].

Synchronizing with the server occurs at regular intervals through asynchronous XHR calls. By synchronizing regularly and on short intervals the local copy of the object graph can be kept in sync to the server with minimal skew between the two. But as the library supports offline use of objects, it is entirely possible that the snapshot stored by the browser becomes far out of sync with the server. This can lead to the skewing of persistent data between the browser and server that a developer may view as too much to be acceptable and thus want to prioritize the updating of the browser. This “cold” browser scenario is especially acute in a browser that has never accessed the Web application before.

“Cold” Synchronization

When the browser has been disconnected from the server over an extended period (which includes connecting to the server for the first time) there is the potential for a large amount of data to be sent to the browser from the server. This could be troublesome as the Web application might not be able to function until at least some critical subset of persistent objects is available for use. If the user is on a slow Internet connection that only compounds the problem.

To avoid the problem of forcing a Web application to have to wait for all persistent objects to be downloaded when a specific subset can suffice, the concept of optional object groups exists within the library. A group identifier is nothing more than a numerical value. A persistent object is assigned to a group with a simple function call: assignGroup(object, groupId). The group acts as an index to retrieve objects by a developer-specific priority. When the browser connects to a server to start a cold synchronization (Figure 3.5), the browser begins requesting groups (line 6), starting at the number 1. The server then packages up all the objects belonging to the specified group and sends them over together along with the newest timestamp amongst the objects sent and specifically what group was
function coldSyncBrowser() {
    var startedRequests = null;
    var group = 1;
    while (group) {
        var serverResponse = coldSyncServer(group);
        var returnedGroup = serverResponse[0];
        var objectsInGroup = serverResponse[1];
        for (guid, json in objectsInGroup) {
            updateObject(guid, json); // Shown in hot sync pseudocode.
        }
        if (returnedGroup) {
            group = returnedGroup + 1;
        }
    }
}

function coldSyncServer(groupId) {
    var gteObjects = objectsGreaterThanOrEqualToGroup(groupId);
    sortByGroup(gteObjects);
    var desiredGroup = gteObjects[0].group;
    var objectsInGroup = filterByGroup(desiredGroup, gteObjects);
    return desiredGroup, objectsInGroup;
}

Figure 3.5: Pseudo-code for cold synchronization between the browser and the server

sent over (coldSyncServer, line 21). When a non-existent group is requested the server automatically sends the next available group greater than the requested group (line 24). When all groups have been sent over, the last “group” sent over is that of all objects never assigned to a group (handled implicitly by line 21).

What happens if the browser needs an object that is not in any of the groups that have been synchronized yet? In this case the read accessor acts as read barriers, catching a storage fault. When the read accessor notices that a GUID does not exist in local storage, a request for that specific object is made. The server replies by looking up the group the object belongs to and sending over the entire group. This acts as a form of data locality to help prevent future requests for individual objects. Once the objects have been received the read accessor is then able to return the proper object and stop blocking.

85
If needed the developer can request a specific group upfront through an asynchronous call. This can be viewed somewhat like cache warming; the developer can explicitly prepare for expected object access patterns. A developer might wish to do this if the user is performing actions which will necessitate access to specific objects soon, but not that the exact moment.

Once all groups — including the non-existent one — have been sent over, the browser is considered ready to begin performing hot synchronizations. The first hot synchronization is performed with the browser considered in a snapshot state, synchronized to the point of time when the first query gathering the first group together was beginning to execute. This is similar to Initial Version database dumping where updates are tracked that occur after the dump begins [80]. This prevents the browser from losing any updates to objects within the first group that occurred while the other groups were fetched. This does lead to redundancy from objects that were mutated in other groups after the first query in the first group started.

However if individual groups were updated based on the timestamp when the query gathering them began instead of all groups based on a single timestamp then cold synchronization could theoretically never halt. An individual group could consistently be updated immediately after the timestamp is recorded, leading to an infinite need to update that single group. But by forcibly stopping cold synchronization at a single timestamp it prevents this issue and allows for all groups to be updated as a whole. Otherwise staleness for individual groups would become greater than all persistent objects as a whole.

When using groups, the most important group is the one the root persistent object is assigned to. As that persistent object is needed immediately to allow any persistent object to be reached, it will automatically trigger a storage fault when the library starts. If the developer is careful to assign all critical objects to the same group as the root object then they will be pulled in as the first group and thus will minimize the time before the Web application is usable.

Before implementing groups, the idea of automating the grouping of objects was explored. Fetching persistent objects in breadth-first order was considered, but if the application traverses deep into the persistent object graph immediately, this would be the wrong approach. Using a depth-first approach has a similar flaw, though, in that if the Web application initially accesses persistent objects shallow
in the object graph then the persistent objects will not arrive fast enough still. Hav-
ing developers specify which approach could work, but that requires developers to
know their common access pattern of the object graph. It seemed more reasonable
to simply have developers semantically group objects when this cold synchroniza-
tion optimization is needed (if no groups are specified then all objects belong to
the non-existent group and thus get downloaded at once).

3.5 Correctness

When storing a user’s data in an automated fashion, it’s important to make sure that
the approach is correct. It is unacceptable if a flaw in the design causes a user to
lose critical data. The user should also be able to access their data as frequently as
possible; disconnection of the Web application from the Internet before performing
a complete synchronization with the server should be the only reason an object
cannot be accessed.

Correctness follows from 5 assertions, enforced by my library, as described in
the following paragraphs:

1. All persistent objects are eventually identified
2. All mutations of persistent objects are eventually detected
3. All mutations are eventually recorded by the server
4. No recorded mutation will overwrite unexamined mutations on the server
5. No conflicting update detected by the server will lead to a browser using the
   conflicted version of a persistent object

(1) Correctness requires that all persistent objects are identified (how long this
takes, along with other time-related aspects, is part of the focus of Section 3.6.1).
The objects to persist are based on reachability from the root persistence object.
Conceptually a depth-first search from the root object would discover all objects
that are to be persisted. In truth the live object set keeps track of persistent object
as they are accessed to alleviate the need to perform the depth-first search. The set
provides a way to always known what objects need to be persisted.
(2) Mutations must be detected in order to ensure correctness. Assignment to a property previously known to the persistence library is detected by the write accessor instantly. Checking all properties on all objects in the live object set for new properties (at a frequency defined by the developer) handles discovering properties not known to the persistence library.

(3, 4) Updating the server occurs when a log of changes is sent by a browser. Correctness dictates that updates be recorded as long as they will not lead to the loss of data. This would occur only when overwriting other unexamined updates by the browser sending the update. As each object is timestamped on the server with the time when the mutation was recorded by the server, the timestamp can be compared against the last synchronization timestamp from the updating browser. If the synchronization timestamp is newer than the mutation timestamp of an object then there is no data change that the updating browser is unaware of, thus the object can be updated on the server. Otherwise conflict detection is needed based on what kind of mutation it is.

The timestamp also allows for correct updating of the browser sending the update of mutations made by other browsers. If any object has a mutation timestamp newer than the last synchronization timestamp, the updating browser needs to update its cached version of that object. Since the timestamps are all controlled by the server there is no chance of clock skew causing a browser to miss an object update because the updating browser’s had different clock values.

(5) To be correct the browser cannot update its local copy of an object when it is discovered it conflicts with the server’s version of the object. If there is a conflict, the application is asked to resolve the conflict, preventing an improper heuristic causing a bad merging of data for an object.

3.6 Evaluation

3.6.1 Benchmarks

To demonstrate the feasibility of the persistence library it has been tested on several micro-benchmarks and a use-case based on eBay Web service data. The focus is to determine the scalability of application-level persistence under a variety of object
configurations. I focus only on measurements of the JavaScript client-side portion of the library since this is the contribution of the research. A server-side object repository in Python for use with persistence library prototype is implemented, but its performance characteristics are not a salient part of this work as no novel contribution is dependent on server performance.

All reported benchmark numbers were taken using the Dromaeo JavaScript performance testing suite [68] to execute and measure the benchmarks. Dromaeo executes each benchmark as many times as possible until the total elapsed time surpasses one second (which can include a single execution taking longer than a second), leading to a unit of “operations/second”. This allows for extremely fast benchmarks to be accurately measured as the accuracy of the clock in browsers can be as coarse as $15\text{ ms}$ executing for at least one second makes a $15\text{ ms}$ skew lead to at worst-case $1.5\%$ mis-report of results. Restricting total benchmark time to one second also prevents any single benchmark from delaying the entire measurement process. Each benchmark is measured five times and the average is reported. All numbers are rounded to the nearest whole number, including rounding down to zero; since the targeted benchmark speed is $150\text{ ms/operation} — 7\text{ operations/second} —$ then any benchmark taking longer than 2 seconds — $0.5\text{ operations/second}$ — leads to a rounding down to 0 operations/second which is clearly unacceptable. This specific target rate is desired as a person typically cannot notice GUI delay if it is under $150\text{ ms}$ [25]. Not meeting this performance goal would mean the library has introduced overhead that will be visibly detrimental to the Web application’s performance. This is especially important not to cause as the library introduces periodic operations such as stabilization and garbage collection which may be triggered when the user is in the middle of a GUI operation.

The machine used to take the benchmark measurements is an Apple MacBook laptop running a 2.2 GHz Intel Core 2 Duo processor with 4 GB of RAM. The operating system is OS X 10.6.3. Numbers are reported from tests on Apple Safari 5.0.4 [4], Google Chrome 4.0.249.30 [5], Mozilla Firefox 3.6.3 [6], and Opera 10.53 [7].

3http://ejohn.org/blog/accuracy-of-javascript-time/
4http://www.apple.com/safari/
5http://www.google.com/chrome/
7http://www.opera.com/browser/
Figure 3.6: Benchmarking the persistence of 200 new objects

At the moment Internet Explorer 8 supports the Web storage API but it does not yet support ECMAScript 5 accessors for JavaScript objects (all the other browsers support (at least) Mozilla’s `defineSetter` and `defineGetter` which operates like ECMAScript 5’s `defineProperty`). Thus, the evaluation reflects what is possible working within proposed emerging, but not widely deployed, standards. Clearly, offline Web applications are a new area and browser vendors are still working towards finalizing their implementations.

Micro Benchmarks

There are three micro-benchmarks in total. All three use a JavaScript object with a fixed number of properties consisting of other JavaScript objects, each of which has 20 properties made up of the integers 0 through 19, as the test data (the exact number of bytes used per object is unknown as each JavaScript interpreter uses their own representation for data and do not provide an API to find out the bytes used by an object). The number of objects involved is either 200, 1000, or 5000 objects per benchmark run. The first benchmark measures the cost of persisting new objects. The second benchmark measures the cost of stabilization. The third benchmark measures the cost of persisting new objects and then garbage collecting them.
Figure 3.7: Benchmarking the persistence of 1000 objects

Figure 3.8: Benchmarking the persistence of 5000 objects
The first benchmark measures the cost of persisting objects. The control scenario takes the described object graph, serializes it to JSON, and then stores it in local storage. The library scenario takes the object graph and assigns it to the root object which has a write accessor set on the property that the object graph is assigned to. This benchmark is meant to illustrate the cost of persisting an entirely new object graph which has become reachable instantaneously, by constructing the graph as transient objects and then attaching the root of the graph to the persistent root. This allows the control to store the equivalent data (only because there are no cycles in the object graph), but without the benefit of automation.

At 200 objects (Figure 3.6), all browsers are able to perform both the control and library scenarios at the target 7 operations/second or better; the library scenario performance ranges from reaching the target to being over 16x faster. At 1000 objects (Figure 3.7), though, all browsers but Safari can perform the control scenario at no worse than 26 operations/second while for the library scenario only Safari can operate fast enough at 18 operations/second (although Firefox is very close at 6 operations/second). At 5000 objects (Figure 3.8), only Firefox and Opera can perform the control benchmark above 7 operations/second while no browser comes close to that target under the library benchmark (Safari comes the closest at 2 operations/second; the performance difference between the control and library case is explained below).

This suggests that processing 200 new objects at once for persistence works across all modern browsers tested. With Safari’s 18 operations/second at 1000 objects, it seems reasonable to think that the other browsers will be able to perform to process 1000 new objects while reaching the 7 operations/second goal in the near future if browser speed improvements continue at their current rate. Unfortunately processing 5000 objects at once does not seem reachable in the immediate future.

Fortunately, it does not seem unreasonable to think that Web applications will amortize the cost of persisting objects by adding new objects over time instead of at once as this benchmark does. Most Web applications are (currently) not data-intensive. While this might stem from performance issues, it does mean that at least for the time being the likelihood of a Web application creating 5000 objects transiently and then suddenly want to persist them is very small. Even looking at current Web applications that could potentially have large data sets, e.g., Web-
based e-mail, typically one does not suddenly get 5000 new e-mails requiring persistence. The type of data-intensive workloads requiring 5000 objects to be created quickly would most likely be processor-heavy as well and thus not be appropriate for being written in JavaScript to run in a browser.

It should be pointed out that across this benchmark that Safari 5 actually performed worse in the control scenario than in the library one. The exact same benchmark was used for all browsers, so this was not caused by benchmark error or code variance. When the same benchmark is run against Safari 4, the control numbers are much like Opera’s while the library numbers are roughly equivalent to Safari 5’s; Safari 5 performs better at 200 and 1000 objects while performing marginally worse at 5000 objects. The loss of performance stems for Safari 5 taking longer to serialize large objects to JSON. As Safari 5.0.0 is such a new browser this can be viewed as a temporary performance regression unless the reason behind the performance loss is for semantic correctness.

The second benchmark measured the overhead of stabilization for 200, 1000, and 5000 objects (Figure 3.9, Figure 3.10, and Figure 3.11, respectively). This benchmark is meant to show what the general cost of using the persistence library

---

The benchmarks were run on 2010-06-07, the day Safari 5.0.0 was released.
Figure 3.10: Benchmarking the stabilization of 1000 objects.

Figure 3.11: Benchmarking the stabilization of 5000 objects.
is in the best case; no persistent objects are mutated but simply checked to make sure their persistent state is accurate.

The numbers show that all browsers can check the state of 5000 objects at 8 operations/second or better. In the best case, assuming a roughly linear degradation, all browsers except Firefox should be able to handle 10,000 objects. This suggests that unless a Web application is extremely data-intensive that the persistence library will not be a performance bottleneck when it is not having to actually process new objects.

The final micro benchmark measured the overhead from the garbage collection step. For each measurement the data set is created, persisted, local state is reset (i.e., the live object set is emptied), and then a garbage collection process is run. This measures not only the cost of persisting the data set, but also the overhead of a garbage collection pass which collects nothing but has to load all persistent objects from local storage; short of a stabilization step being executed, this benchmark covers the entire life-cycle of a persistent object from persistence to deletion. This is the worst-case scenario for garbage collection as it requires loading all objects from local storage as well as traversing the entire object graph.

As can be seen just by examining the results for 200 new objects (Figure 3.12),

**Figure 3.12:** Benchmarking the garbage collection overhead of not collecting 200 objects.
**Figure 3.13:** Benchmarking the garbage collection overhead of not collecting 1000 objects.

**Figure 3.14:** Benchmarking the garbage collection overhead of not collecting 5000 objects.
garbage collection in this scenario is very expensive. While Safari is the only browser to operate above 7 operations/second — 15 operations/second — the added overhead of garbage collection slows the browser down to around 13% of the performance if the 200 new objects were simply persisted (Figure 3.6). Opera has the smallest performance penalty, but it still performs at 30% what it could without garbage collection.

Examining the 1000 and 5000 object benchmark numbers do not portray a better situation (Figure 3.13 and Figure 3.14). No browser is able to operate at 7 operations/second at either data set size. At 5000 object the performance is so poor that the numbers round down to zero.

Luckily garbage collection should be a rare occurrence. As it should only be run after the browser has been executing for some time, at least some portion of the object graph should already be live. The triggering of garbage collection could also be made to be a user-chosen action so that the user is fully aware that the application will be unresponsive for a short time. For example, except for Chrome the other three browsers operate at or better than 1 operation/second for 1000 objects. It would seem to be reasonable for a user to wait for a second to pass while a rare operation occurs. Consider when the browser clears its browser cache, which itself is a rare operation but can take several seconds to complete and causes a large amount of disk activity and something users are accustomed to. For these reasons, the overhead shown in these measurements is not quite as gloomy as it might seem.

**eBay Benchmark**

The micro-benchmarks were scaled specifically to demonstrate the limits of the approach being presented by this thesis. For a quantitative evaluation of the library using real-world data, data from a live Web service was desired. Since most Web services have a maximum query result limit — you are typically expected to paginate the results to get results beyond the maximum single query limit — it was also desirable to test against a realistic maximum load on the library rather than trying to find its theoretical limit.

EBay’s online Web service was chosen to base the real-world quantitative benchmarking on. The service was queried using the search term “iPod” and asked
to deliver 100 auctions (the limit for a single request) in JSON format. The resulting object tree is made up of 592 objects with an average 4.7 properties/object. The JSON was stored as a string literal in the test code. The control experiment converted the JSON to its object representation, serialized the created objects back to JSON, saved the new deserialized JSON representation to local storage, and then converted the objects to XML. The conversion is used so as to exercise accessing every property on every object. For benchmarking the persistence library the JSON test data was converted to its object representation, serialized the created objects back to JSON, attached the objects to `persist.root`, stabilized the object graph, and then converted the stabilized objects directly off of `persist.root` to XML.

The benchmark measures the overhead of the persistence library compared to a very simplified, similar approach to storing an object tree. By fully stabilizing the object graph for the library the benchmark not only measures the cost of persisting the data but also the cost of adding accessors for all properties. Also by converting the object to XML the benchmark makes sure to visit every object and property, thoroughly exercising the accessors.

The results of the benchmark in Figure 3.15 shows that Safari (but not any other browser) is able to operate quickly enough at 20 operations/second for the

---

**Figure 3.15:** Benchmarking the use of the eBay Web service for 100 auction results which are stored to local storage and converted to XML.
user to not notice any performance degradation. In terms of overall performance loss, Chrome comes in at the worst by performing at 2% the rate that the control scenario is able to. It should be kept in mind, though, that the library scenario adds several beneficial features that the control scenario lacks, e.g., instant detection of mutations, synchronization to a server, etc. Safari’s performance also gives hope that the other browsers will eventually match its performance.

If you compare the eBay benchmark results to the persistence micro-benchmark, the closest numbers are for persisting 1000 objects (Figure 3.7). While the eBay benchmark uses 59% as many objects of the 1000 persistent object micro-benchmark, it is only 11% faster for Safari. The reason for the difference is the overhead of taking the JSON to create the objects to persist, the cost of calling every accessor in the object graph, and the cost of creating the XML representation.

3.6.2 Experience

In addition to quantitative measurements, I wanted to gain some experience using my approach to better understand its use in an application. For this purpose, an existing Web application was ported to use the persistence provided by the library, as well as comparing it to an existing JavaScript toolkit that provides some support for persistence.

Moi Web Application

To initially make sure that the library not only worked, but to understand how it could be used effectively, I ported a Web application of my own design, called Moi, to the persistence library. The application was designed to allow me to report to my supervisor what I had done during the prior week using a simple Web page form — which saved its input after every keystroke — to enter data and an Atom feed to consume the data. The application supported offline use so that updates could occur at any time.

The original Moi application (shown in Appendix A) consisted of objects made up of the date at the start of a work week and the summary of what had occurred that week. When a user changed/added an entry it was saved locally, added it to a log of changed entries, and then submitted it to a Web service to store it (if there
was a network connection).

When moved over to the persistence library this simplified Moi greatly (shown in Appendix B), dropping the LOC from 216 to 58 – a nearly 74% reduction. The objects were attached to persist.root with property names consisting of the date of the Monday representing the week of the object. The persistence library then handled the rest. That eliminated all of the code which had been written to manage the storage of the data. It also allowed me to focus on the usage and structure of the data instead of its storage since the persistence library took that over that responsibility. In the case that a large number of entries were necessary to be stored on a browser, I used object groups to divide entries by their year. This would allow the persistence library to automatically chunk 52 entries in each batch. In the case of where data was updated while offline, I registered a callback for when new data was received. This allowed me to update the current view if needed. I also provided a simple view to handle conflicts by popping up a window with the data that was originally pushed to the server, so the user could view the updated data and manually decide how to resolve the conflicts.

**Dojo Toolkit**

Probably the closest existing JavaScript library that provides data persistence and offline support is the Dojo Toolkit\(^9\). Within Dojo there is the JsonRestStore for reading and writing data to a server using a common HTTP Representational State Transfer (REST) API [34]. Dojo additionally provides the use of the OfflineRest library with JsonRestStore to provide offline access to the data they manipulate through JsonRestStore.

JsonRestStore allows a developer to work with objects directly, like the persistence library presented by this thesis, and not through a low-level data manipulation API. The library does require, though, developers register objects for storage. To save any mutations a developer must mark an object as “dirty” when they have mutated and then explicitly call a global save function:

```javascript
// Marking an object as 'dirty' and saving it.
jsonStore.changing(item);
```

\(^9\)http://www.dojotoolkit.org/
item.attr = newValue;
jsonStore.save();

The explicit registering of objects for saving and the need to mark mutated objects as dirty along with an explicit save call prevents JsonRestStore from being as automatic as the approach presented by this thesis. JsonRestStore also does not process mutations to sub-objects; every object that changes must be marked as dirty to have changes be picked up.

A possibly bigger difference between my library and JsonRestStore is that communication with the server is one-way from the browser to the server. There is no support to pull new objects from the server nor check to make sure data unseen by the browser is not overwritten on the server. JsonRestStore takes the view that the browser is the master copy of data, as opposed to my library which makes sure all browsers used by a user are consistent.

Dojo provides a simple e-mail application, Dojo Mail, to demonstrate their GUI toolkit. The application was used for gaining further experience porting applications to use my library. Dojo Mail contains a tree representing various e-mail folders. That tree is rendered and data exposed through an object implementing the model interface. The interface itself exposes data in a tree structure and allows for notifying the tree widget when the data it is rendering has changed.

By implementing the model interface that Dojo’s requires for use with model, I created a persistent version of the model. This shows that it is possible to take the persistence library and use it with pre-existing JavaScript libraries transparently.

It should be noted, though, that blindly using the persistence library with Dojo’s JsonRestStore fails. Certain metadata set by Dojo on every persistent object is expected to be transient and not persistent; since my library persists all properties found on an object, this causes errors for Dojo. This should become less of an issue once ECMAScript 5 [28] comes into wider use.

The latest version of JavaScript provides a function that is able to specify whether a property is enumerable or not. If JavaScript libraries used that functionality to “hide” metadata properties from being discovered when an object’s properties were enumerated then my library would not see them and thus have them persisted.
Chapter 4

Related Work

As with any substantial work, my research builds upon the work of others. Since my research divides itself cleanly between desktop and Web application work, that division has been followed in the related work section for easier referencing.

4.1 Having User Actions Control Resource Access

My research resulted in an approach to security where the POLP is enforced by monitoring the I/O of an application to deduce the intentions of the user in regards to desired resource usage. This required researching code access policies to see how well a statically defined security policy could work. I learned about the Java security model and system in order to know its limits. I looked into research regarding how users grant explicit authorization through their actions. Since my security code was to be codified independent of the desktop application being secured, privilege separation was studied. Object capabilities was researched to control resources by controlling whether the application is ever given an object representing the resource instead of checking permissions based on identity. I looked into information flow analysis to see what considerations I should have in terms of how data can travel through an application. I researched what AOP security work preceded me. And finally, I looked into secure GUI research.
**Code Access Policy**

Security policies for controlling access to resources can be defined upfront, codified in a code access policy. Some work has focused on trying to construct a suitably expressive policy file format rich enough to express any potential security need [26]. A finite automaton can be defined statically which dynamically enforces security decisions [32, 84].

Compared to my work, neither approach above attempts to enforce the POLP. With my work I have done this by dynamically considering how the user is interacting with the application to make security decisions. Statically stating what the security policy for accessing resources should be inherently limits you to be either too restrictive or too relaxed in order to cover all possible cases. While one can try to predict a user’s behaviour, there is no guarantee the detail will be of a sufficient level to capture all possibilities.

**Java Security**

Java’s virtual machine, the JVM, provides its own security system [1]. The JVM uses a permission-based system where a user can specify security policies in a configuration file. Rules can be defined that control issues ranging from what classes are allowed to be instantiated to what files and server domains can be accessed [7]. Security is then enforced by a Java security manager whose performance can vary widely depending on what is being enforced [46].

For my work I am able to leverage the JVM’s security manager as a superclass to derive my own security manager. This allowed me to not have to implement static policy rules which are required by an application to begin execution, such as the ability to read its configuration file.

**Explicit Authorization**

Yee [102] introduced the *Principle of Explicit Authorization*: the safest way to ensure that resources are accessed according to a user’s wishes is to allow the user to mediate resource use.

As a proof-of-concept, a custom file chooser GUI widget is given by Yee. The use of the file chooser is natural for users as it provides a transparent way to monitor
the intentions of users resource access decisions. A user study conducted by De-Witt and Kuljis [27] showed issues with the transparency of the widget, something that my work does not attempt to address. Other work provides implementations of this file chooser widget for Windows, Linux, and Solaris [53, 85, 90].

However, their implementation only handles explicit authorization for one specific widget type instead of trying to solve arbitrary GUI widgets as my work does. This thesis describes a new mechanism which goes beyond authorizing access for just the resource that is directly accessed by the user to the resource dependencies of that directly-accessed resource.

**Privilege separation**

Software assurance requires trust in all code which may influence information security and privacy. Since real applications are built from many different modules, this makes it difficult to audit a complete application to know exactly which modules require what trust. Basically, there is a fundamental tension between software assurance and software complexity.

As noted by Provos et al. [78] in their work on privilege separation, it is useful to isolate code that is critical to security to specific modules so that those modules can run with specific security permissions to prevent privilege escalation. My approach helps with minimizing the amount of code that needs a security audit for desktop applications, because the resource access behaviour of the application is refactored into policies that can be audited. Work has been done by Brumley and Song [17] to try to automate the partitioning using a minimal amount of code annotations. In contrast, my work fully partitions security-related code to authority aspects which are fully separate from an application’s source code, although using AOP does put security-critical code physically farther away from other source code using it, potentially leading to code maintenance burdens.

**Object Capabilities**

Capabilities are tokens which represent permission to perform an action. Object capabilities are capabilities represented by holding a reference to an object that

---

1Gaining permissions to perform an action that it was not originally allowed to perform.
has permission to perform an action. For capabilities in general and object capabilities specifically, enforcing security works through the control of references to security-critical objects [65, 89 96]. Work with Java and capabilities has been done by Hawblitzel et al. [44] who introduced revocable capabilities. Mettler and Wagner [64] created the Joe-E language which turns a subset of Java into an object capabilities language. Their approaches allow for a very fine-grained level of authority control that can be dynamically determined. However, mainstream programming languages and execution platforms are not capabilities-based and thus do not always fit in with a developer’s skillset.

Object capabilities formed my thinking around the idea of controlling resource access by simply not letting the program have access to any resource except the ones it needs to have access to. By not relying on the security manager to manage every resource use but instead focusing on whether an application is even given a resource, security can be enforced at the edge of the application where resources flow into the application. Often called a membrane [65] in object capability terminology, this boundary between the code that handles the security — the powerbox [89] — and the application itself which has restricted privileges, was a key inspiration for my work.

Language-Based Information Flow Analysis

Language-based information flow analysis can be used to ensure that untrusted data does not flow into some sensitive operation, leading to a security breach. This analysis can be achieved both dynamically and statically.

Tainting is the technique of dynamically marking input into a program as untrusted and preventing all such input from being used in any sensitive way at runtime [42, 59, 69 101]. Some languages such as Perl and Ruby actually include tainting in the underlying language system, mainly to prevent strings from being used as malicious input [6, 94]. Others have used tainting to prevent executing input as pointers or other forms of malicious input at the memory level [92]. Part of a taint system is identifying which parts of the system are generating data from privileged sources. Work has been done to attempt to automatically detect when code requires privileged permissions compared to other code in the application by
examining Java bytecode to perform interprocedural analysis [76].

Static analysis can also be used to provide a conservative approximation of whether untrusted data may flow into a sensitive operation. Privilege separation can also be used to empower flow analysis by helping to focus the analysis on only the code that requires privileged permissions [88].

My approach can be viewed as an information flow policy which mandates that resource identifiers used in resource access operations have originated from a trusted source: GUI, disk, or network I/O. The implementation of my security mechanism is more similar to tainting-based systems than to static analysis since it is based on run-time enforcement. This is motivated by the need to distinguish between different elements which can only be determined at runtime, e.g., the content of an XML document.

Tainting also has different goals than my work. The common use of tainting is to make sure malicious input originating from outside the system is not used in any situation where input should be verified. This is very coarse-grained and applied to all input for all security-related methods. My work, on the other hand, is more fine-grained by focusing on resources, e.g., I do not worry about SQL injection attacks; I only care about resource access. This allows me to have a more focused approach than tainting by not worrying about all input into the application but only those that relate to resources. A benefit of this is that my approach is not as invasive to the application as tainting, although it does not directly address the problem tainting is designed for. Tainting also does not help enforce the POLP as tainting protects against using malicious input, not minimizing permissions granted to an application.

**AOP security**

Security is a well-known crosscutting concern which can be described and implemented using an aspect-oriented methodology [100]. As with any software methodology, the benefits and costs of using a specific methodology can depend on the precise requirements of a particular software development project. AOP’s flexibility has allowed its use to introduce security into an application, like Walker and Viggers [97] who used their declarative event patterns to add user authentication
for FTP. For me this showed that using AOP for adding security to an application is a feasible approach.

Some work has been done in applying security by adding it externally to the program being secured without using AOP. Work has been done by multiple people to develop a binary rewriting approach to add security into a program in order to enforce a code access policy, all before research of using AOP and AspectJ as the mechanism to handle binary rewriting [31, 72, 98]. Once again, this showed me the feasibility of AOP for security.

Leveraging existing security libraries and weaving them into an application was explored by Huang et al. [51]. Other research explores using AspectJ to implement an entire security kernel for Java [99]. In work by Gao et al. [36], security policies based on different types of role-based access control are implemented using AOP. Tying together several security systems at once into a compositional framework is even possible. Charfi and Mezini [23] integrate security policies into business workflows using AOP.

A Domain-Specific Language (DSL) was introduced by Mourad et al. [67] to help harden existing applications. Their goal is to provide a DSL that non-security experts can use to secure an open source application. This varies from my approach as they trust the developer as a trusted participant in the hardening process instead of as a possible adversary.

AOP has also been harnessed for intrusion detection [71, 103]. Haley et al. [43] enumerate key assets and potential attacks to discover the crosscutting concerns involved in security issues. Others have created access control systems specifically for use with AOP [79].

My framework differs from all of these approaches by supporting the POLP by restricting resource access for applications. None of the previous work in AOSD has described how to support this particular security requirement. Previous work has focused on adding in an access-control solution to applications which typically requires defining upfront what permissions are to be granted, compared to following the POLP which only grants exactly what permissions are needed.

The approach I designed relies on monitoring the use of I/O channels which crosscut the application source code and connect an application to its environment. Specifically, I considered special requirements for I/O channels such as graphi-
cal user interaction and structured data formats such as XML. Since it is common for I/O channels to be implemented as frameworks or middleware libraries, I leveraged AspectJ’s abstract aspect mechanism to separate high-level policy from framework-specific deployment. Differing from previous work which has explored abstract aspects, I needed to consider the decomposition of aspects based on separation of concerns of the developers in conjunction with the trust associated with different developers. In my approach you have a policy expert, framework expert, and application developer all contributing to the security solution, but at varying levels of required trust on the part of the user.

**Secure GUIs**

Work has been done to secure the window managers and servers on operating systems. Feske and Helmuth [33] worked on a secure overlay for X11 called Nitpicker, while Shapiro et al. [86] created a new window manager for the EROS operating system. While both help to prevent GUI spoofing\(^2\) by other applications, the goals of secure GUIs does not encompass capturing GUI I/O to make permission decisions.

### 4.2 Automatic Object Persistence in JavaScript

My research into introducing automatic object persistence into the Web browser led me to researching various areas in computer science. I looked at network file systems and databases as both dealt with using data both locally and remotely. Updating, synchronization, and replication were examined individually to see how best to handle each. I looked at how persistent programming languages have tackled automated object persistence. Distributed object systems and languages was researched as they have dealt with how to shuttle objects across a network quickly and reliably. Persistent object systems were researched for obvious reasons. Web browsers were examined to know exactly what is and will be available to Web developers to work with.

\(^2\)Providing a look-alike GUI so as to fool the user into entering sensitive information. Commonly seen in online phishing where a website poses as a person’s bank.
Networked File Systems and Databases

A modification of the AFS file system by Huston and Honeyman [52] allowed for disconnected use using an optimistic update policy much like I use. Callbacks were also used to notify the user if conflicts arose from disconnected mutations. The Coda filesystem also used optimistic updating [57, 82]. Coda viewed local data as second-class replicas of the master data, in line with my master-copy view. The Bayou system influenced me by making it clear that metadata should never contain individual server-specific details as that prevents scaling and thus all metadata should be globally applicable and usable [93].

The Ficus file system emphasized “no lost update” semantics to retain data consistency [45]. The authors also pointed out that users act as their own write token as a user typically works on only one computer at a time instead of multiple ones. This minimizes actual conflicts in data from multiple updates while working offline as the user causes updates to only come from one source at a time by physically only using one computer at a time. While strictly not a networked file system, Balasubramaniam and Pierce [9] wrote about synchronizing files where they pointed out optimistic updating is a must for disconnected operation as not being able to write data while offline is too much of a hinderance to a user. While all of these researchers contributed to how I handle offline mutations, their work focused on file systems and not object persistence.

Gray et al. [40] noted that conflicts from offline mutations should be handled by the application as opposed to applying a generic algorithmic conflict resolution solution. Each application has their own unique requirements such as which update should take precedence, what conflicts can be automatically resolved, etc.

Updating, Synchronization, and Replication

When working in a transactional system where multiple clients can update the server there must be a way to guarantee that communication is properly controlled so that transactions do not interfere with each other. Traditionally a two-phase commit protocol is used [37], although other commit protocols can be used in a distributed setting [24]. My work avoids the need for locks by not supporting

\[3\text{It is never acceptable for the data contained in an update to not eventually be saved}\]
transactional commits between multiple objects, instead updating objects in isolation.

To control synchronizing caches, the goal is to minimize the divergence between the cache and the source [70]. File systems have been dealing with this issue for quite a while thanks to the concept of dirty blocks in memory that must be committed to disk [66]. This relates to my work as the server can be viewed as the source of data and the browser’s snapshot of the data acting as a cache.

Replicating data requires knowing when to trigger the replication of the data. Having a measurement of staleness helps as shown by the Mariposa distributed database system [87]; sending updates to a server is my way of preventing staleness of any data on the server compared to what the browser has. One must take into consideration the logical, distribution, and transmission requirements when communicating [75]. Clock skew must be avoided, so using a master concept of a timestamp as I do eliminates this problem [13].

From a Web services perspective, this ties into Brewer’s conjecture: of consistency, availability, and partition tolerance you can only have two [38]. Since one almost always wants partition tolerance (viewing the system as a single whole), you must choose between consistency or availability. Since consistency can eventually be reached, it can be weakened to allow for availability to be chosen [95]. My work chooses partition tolerance and availability. This allows the data returned by the server to not necessarily being the most up-to-date. Consistency is eventually reached once all updates are sent to the server and conflicts have been handled.

**Persistent Programming Languages**

Brown et al. [16] showed that persistence can be treated as orthogonal to the language definition as persistence can be made transparent to the language runtime. For the orthogonality to work, though, it requires some rather low-level access by the language. Eliot and Moss [29] implemented the Mneme object store which used pointers to represent edges in the object graph like in this research. Hosking and Chen [50] created an orthogonal persistence system that tied into the garbage collection step of a programming language, finding it acceptable to persist only when garbage collection is run and thus with a slight delay. My work differs from
these papers by adding object persistence to a pre-existing programming language without interpreter/compiler modification.

**Distributed and Persistent Object Systems**

Research on object-oriented, distributed programming languages solves many issues with how to transport objects over a network to remote systems. The Emerald and Modula-2 programming languages were some of the first languages to provide location transparency while not hiding details that may be needed for performance reasons [14, 60]. I borrowed ideas on how to identify objects from these works even though they focus on object mobility for execution while I have focused on persistence.

The work of Printezis et al. [77] on the PJama persistent object store showed me that using a read barrier to hide persistence mechanisms was a reasonable approach. This allows me to use accessors to make reads of an object’s property seem normal while in actuality they can stop execution while the object is retrieved from local storage.

Carey et al. [21] created SHORE which was an object-oriented database. Their novelty focused on server-side gains such as creating a peer-to-peer server architecture, which did not directly relate to my work.

The Thor object-oriented database was a persistent object system that supported disconnected operation in an optimistic fashion [41, 61, 62]. It was designed for desktop applications operating in a WAN environment where server redundancy was key for guaranteed data availability. Thor used page memory locality to decide how to pull over related objects. One could also specify a query which warmed the object cache in an operation called “hoarding” which Coda pioneered. This allowed a user to specify key objects to make sure were always available instead of just recently requested objects, similar to my concept of object groups except without the tiering of importance like I support.

In comparison to my work, the design space I worked within was the Internet where I had to operate within the confines of the browser and not with the full power of C++, with its operator overloading and ability to directly manipulate memory which would have simplified my design and improved performance. I
also didn’t have access to the underlying OS as Thor did, which would have let me have more control over what local storage mechanism I could have used. This required me to do eager caching of all objects in the browser to avoid latency issues, keeping the dichotomy between working online compared to offline a transparent operation under a single system to keep transparency high [83]. Having to work within JavaScript’s restrictions prevented me from having lower level hooks into objects where I could more readily be notified of mutations instead of developing my stabilization step.

**Web Browsers**

The Gears plug-in introduced the idea of local storage in Web browsers [39], which helped lead to the W3C Web Storage API [48].

Transparently making offline data access in the browser has been attempted in other ways than my approach. Mehta et al. [63] developed a plug-in which cached AtomPub requests, acting as a proxy for a specific Web service data format. A more general proxy for caching Web accesses for later access has also been done [22]. Sync Kit automates the caching of website templates and data using HTML5’s SQL database storage, allowing page rendering to happen locally in the browser [10].
Chapter 5

Conclusions

This thesis has addressed the disparity between desktop and Web applications in terms of resource access and usage. Desktop applications have traditionally not secured access to resources with as much rigour as Web applications have thanks to the sandboxing of a Web browser. On the other hand, Web applications have not had an intuitive and powerful way to persist data as desktop applications do thanks to their file system access. Both of these shortcomings have been addressed by this thesis.

I have developed an approach to allow desktop application developers to apply the POLP for resource access while being an untrusted participant in the security mechanism. By monitoring I/O through authority aspects, it is possible to detect a user’s intention for accessing resources through an extension of the Principle of Explicit Authorization. This allows the user to have complete security coverage when used in conjunction with explicit user prompting.

By initially only providing the authority to access what is absolutely necessary for the application to start through a code access policy, no ambient authority is given to the desktop application. Input from configuration files then extend the authority granted by XPath rules specified in a security policy file such that the application can reach its initial state with the minimal amount of authority. Once started, the desktop application gains more authority as the user interacts with the application, expressing his intentionality through input. Not only is GUI interactions used to infer intent, but so is network and disk input.
All of this is possible through the use of authority aspects which weave into the desktop application, capturing all of this input to infer the intent of the user. Authority aspects are created by framework experts which weave into libraries and frameworks so that they can be re-used by other application developers. The desktop application developer himself can provide authority aspects to handle any GUI widgets which do not capture a user’s intentionality in an obvious way. All of this, along with the XPath and JVM code policy files, can then be examined by a security expert before weaving the code into the application to secure it. This provides a multi-layered architecture where stakeholders of varying levels of trust can contribute to the security of the application.

I performed a case study on RSSOwl and Lobo, and successfully applied my framework with only moderate effort. It caused both applications to not access any resources without the user performing an action that inferred that the resource usage was intended. Out of all the GUI menu options in RSSOwl, only two instances required explicit prompting of the user; Lobo required no prompting. The performance cost was also found to be negligible, especially for what was gained by the user.

As for improving the situation for resource usage in Web application, I have created an automated object persistence library for Web browsers while relying exclusively on mechanisms which are described in emerging or proposed standards. I was able to work around shortcomings of the JavaScript programming language (and any other dynamic programming language with the same meta-programming restrictions) in order to detect mutations of persistent objects as quickly as possible; either on a set interval for new properties on JavaScript objects or instantly for mutations of known properties. All detected mutations are logged and stored to local storage to allow for synchronizing the mutations to a server on the Internet.

For communication between a browser and a server, I developed a communication protocol designed specifically for use over the Internet. It had the requirement that it work with only the browser initiating a connection. It also was designed to work with a potentially latency-heavy Internet connection. Finally, it works in the single-threaded environment of a Web browser.

The library not only works with a possibly slow, restricted Internet connection, but with no Internet connection at all. By utilizing offline storage the persistence
library can persist objects while disconnected from the Internet. When a connection is re-established all changes are sent to a server where they are then persisted and potential mutation conflicts are detected.

My evaluation of the work shows that thousands of objects can be persisted with acceptable performance overhead on select browsers. Support for the persistence library exists in four of the five major browsers thanks to my reliance on standards and not proprietary browser features or plug-ins, with the fifth most likely being supported once it implements all of the standards depended upon.

5.1 Unsolved Problems Between Desktop and Web Applications

While my work has helped solve two major issues between desktop and Web applications, there are still other issues to resolve before all technical differences vanish. For desktop applications, rapid deployment continues to be an issue. Web applications can be refreshed and updated locally simply by visiting a website. Desktop applications continue to lack a universal solution to this problem that is as seamless as for Web applications. While some applications have custom solutions, operating systems still do not provide a service on the behalf of application developers. There is also still an installation delay which Web applications lack thanks to their inherent design of serving self-contained, fully functioning components of an application on-demand. If an update feature were provided by operating systems and desktop applications could be structured to work in a more component-based approach to serving the application then this discrepancy could potentially be overcome.

Desktop applications also continue to have portability problems. A Web application works wherever there is a Web browser. While there can be some discrepancies from browser to browser, overall it is still a better experience compared to desktop applications. There are some GUI toolkits which try to allow for cross-platform development, but they require the toolkit to be ported which typically is not done as widely as Web browsers.

Web applications are not perfect, though. They lack access to all services and APIs provided by the operating system compared to desktop applications. For
instance, clipboard access is not provided typically by Web browsers. For desktop applications, though, access to the clipboard is not an issue. Only by continually trying to provide proper APIs to Web applications could this be overcome.

There is also an issue of not having a native look-and-feel for Web applications. Since desktop applications use native GUI widgets provided by the operating system they “fit” into the environment more naturally. Web applications, on the other hand, tend to take a more generic approach as they could be running on any operating system. They also do not have access to native GUI widgets so they can only at best simulate native widgets. HTML can slowly begin to gain the concept of GUI widgets more and more to allow for interactions which fit in better with typical GUI widgets (as some try to do using CSS).

In summary, technological differences still exist between desktop and Web applications. While this thesis helps eliminate issues for both approaches, there are still open issues to be explored to eliminate various differences between desktop and Web applications even more.

5.2 Future Work

There are a couple of directions to take the security work forward. One is to perform an extensive survey of popular Java desktop applications to see what their security needs are and if they are handled by authority aspects. This could lead to learning that there is a certain need not being met or simply that the approach works beyond the evaluation done in this thesis.

Providing custom wrappers for GUI widgets that require using the Status Bar to report what permissions an action will grant could help with usability. Having the application developer wrap GUI widgets which lack any inherent intentionality of what a user’s action represents is not necessarily having the best developer do the wrapping. The framework expert should create the wrappers as they have the skills to do the best wrapper compared to the application developer. For instance, having the application developer need to create authority aspects to handle situations like RSSOwl’s feed listing is suboptimal; it would be best if all GUI widget work was left to a framework expert. For that to work it would require expanding the API of the widgets to let the developer specify what authority would be need when
a certain action is performed. This could then be exposed either with a hover box or the Status Bar to provide the ability to communicate the consequences of interacting with the GUI widget.

It would also be interesting to see what would be required to securely allow other data formats other than XML and HTML to be plugged into the framework for processing for resources. For instance, it is not inconceivable that a Web service call would return resources in a JSON format. While the XPath policy rules will work on any structured tree format, the question is can a third party develop code to securely support parsing e.g., JSON for URIs.

As for the automated object persistence work, I see a few ways to try to improve performance. An alternative approach to the mark-and-sweep solution to detect dead persistent objects would be using a reference-counting approach. I chose not to use this approach for two reasons. One is that reference counting has known issues regarding cycles. While there might be some way to store references in such a way as to solve this issue by relying on the fact that only references from objects that are still reachable from the root object are important, that has not been looked into or thought about deeply.

Another issue with using reference counting would be synchronizing the reference count metadata with a server maintaining copies of all objects. In the situation where a property on a parent object was deleted, but did not lower the reference count of the child object to zero, then updates on both the parent and child would need to be sent to the server. But in the case of the mark-and-sweep approach, only the deletion of the property requires updating the server. It might be possible to not worry about synchronizing reference counts as that should be consistent based on the state of the object graph, allowing the reference count update to be passive and simply a side-effect of updating the server about mutations.

To minimize or remove the stabilization step, a function could possibly be provided which processes an object to set new accessors eagerly. The function could be called in the object constructor under the assumption that the object will not gain new properties in the future. If the developer agreed to that contract then stabilization would not be needed for objects processed this way. Objects that get persisted through assignment to a property could be flagged as “dynamic” such that stabilization applies to them. This would lead to only a subset of persistent
objects needing to be processed by the stabilization step. If enough objects were eagerly processed and considered “static” then this could significantly cut down on how many objects needed to be processed during stabilization. The question then becomes whether Web applications could follow this contract or not.

An object-wide accessor protocol could be added to JavaScript objects by altering the JavaScript interpreter of the browser. In doing so the need for a stabilization step would be entirely eliminated. Measurements would be needed, though, to show that overall JavaScript performance does not degrade and that any overhead from having to check for an object-wide accessor for all property accesses was a net gain in performance compared to having a stabilization step.

Altering the interpreter could also allow for introducing the concept of a deletion accessor. This would eliminate the garbage collection step which is one of the most costly operations my library does.

Adding support of multi-object transactions could be of interest. While introducing the concept would not technologically be difficult – bundling object updates as a single unit and not making a single commit if any object has a conflict seem doable – but may have performance implications. It would also require evaluating if the added complexity was warranted based on real-world application needs.

Tracking and updating mutations at the property level instead of at the granularity of objects could be of interest. While data sharing in a collaborative situation is doable if one does not have multiple users mutate the same objects, some application designs may not allow for this. If multiple users are updating the same object simultaneously then conflicts will be detected by the server as conflicts are done for the whole object. If a shift was made to detecting conflicts at the granularity of properties then this may help alleviate that problem.

Research is never finished. Thus there is room for this research to continue on to be improved and expanded upon beyond this thesis. These ideas seem the most beneficial research topics to pursue to further the work.
Bibliography


124


126


Appendices
Appendix A

Manual Persistence Version of Moi

```javascript
/*
 Sample source code for Moi application without automated object persistence applied (Section 3.6.2).
 */
/**
 Storage module.
 */
var Moi = {};
/** REST API URLs */
var REST_urls = {
    get: "/_api/get", set: "/_api/set", del: "/_api/del"
};
/**
 Function to simply making XMLHttpRequest calls.
 */
var xhr = function (action, args, callback) {
    var error_callback = function (xhr_ob, textStatus, errorThrown) {
        var http_status = xhr_ob.status;
        var data = JSON.parse(xhr_ob.responseText);
        callback(http_status, data);
    };
    var success_callback = function (data, textStatus) {
        callback(200, data);
    };
    var options = {
        async: true,
```
```javascript
    cache: false,
data: {json_data: JSON.stringify(args),
dataType: "json",
error: error_callback,
success: success_callback,
type: "POST",
url: REST_urls[action]};

$.ajax(options);

/**
 * Store a weekly entry, both locally and remotely.
 */
Moi.set = function() {
    if (arguments.length < 3 || arguments.length > 4) {
        throw new Error("Moi.set expects between 3 and 4 arguments");
    }
    var monday = arguments[0];
    var summary = arguments[1];
    var oldValue = null;
    var callback = null;

    if (arguments.length == 3) {
        callback = arguments[2];
    } else {
        oldValue = arguments[2];
        callback = arguments[3];
    }

    var localStorage = function(http_status, data) {
        if (http_status == 200) {
            realStorage.setItem(monday, summary);
        }
        callback(http_status, data);
    };

    if (!oldValue) {
        oldValue = realStorage.getItem(monday);
    }

    xhr("set", {monday: monday, summary: summary, old_value: oldValue},
    localStorage);
};
```
Asynchronously load a weekly entry.

```javascript
Moi.get = function(monday, callback) {
    var localstore = function(http_status, data) {
        if (http_status == 200) {
            realStorage.setItem(monday, data.summary);
        }
        callback(http_status, data);
    };
    xhr("get", {monday: monday}, localstore);
    return realStorage.getItem(monday);
};
```

Delete a weekly entry.

```javascript
Moi.del = function() {
    if (arguments.length < 2 || arguments.length > 3) {
        throw new Error("Moi.del expects between 2 and 3 arguments");
    }
    var monday = arguments[0];
    var oldValue = null;
    var callback = null;
    if (arguments.length == 3) {
        oldValue = arguments[1];
        callback = arguments[2];
    } else {
        callback = arguments[1];
    }
    var localstore = function(http_status, data) {
        if (http_status == 200) {
            realStorage.removeItem(monday);
        }
        callback(http_status, data);
    };
    if (!oldValue) {
        oldValue = realStorage.getItem(monday);
    }
};
```
xhr("del", {monday: monday, old_value: old_value}, localstore);

UI module.

var MoiUI = {};

Convert a Date object to a string.

MoiUI.dateToString = function(date) {
  return date.toString("yyyy-MM-dd");
};

Calculate the preceding Monday (if not already Monday).

MoiUI.snippetStartDate = function(date) {
  var monday = date.clone();
  if (monday.getDay() != 1) {
    monday.moveToDayOfWeek(1, -1);
  }
  return monday;
};

Return the preceding Monday for today (unless it is Monday already).

MoiUI.currentSnippetDate = function() {
  return MoiUI.snippetStartDate(Date.today());
};

Calculate the following Sunday from a Monday.

MoiUI.snippetEndDate = function(date) {
  return MoiUI.snippetStartDate(date).addDays(6);
};

Update the UI to signal if the browser is working offline or not.

var update_status_indicator = function() {
  var border_color;
  if (navigator.onLine) {

border_color = "green";
}
else {
    border_color = "yellow";
}

$("#summary-form").css("border-color", border_color);

/**
 * Display a pop-up window when there is an update conflict from the server.
 */
var handle_set_conflict = function(event)
{
    $("#view-form").val(event.data);
    get_event();
    var popup = open("", "conflict-info", "directories=no,fullscreen=no," +
        "location=no,menubar=no,status=no,titlebar=no," +
        "toolbar=no,width=400,height=200,top=200,left=200");
    var conflict_value = localStorage.getItem("conflict:" + event.data);
    popup.document.write("<p>&lt;pre&gt;<pre&gt; + conflict_value + "</pre>&lt;/pre&gt;");
};

/**
 * Handle the conflict resolution decided by the user.
 */
var handle_deletion_conflict = function(event)
{
    $("#view-form").val(event.data);
    get_event();
    var popup = open("", "conflict-info", "directories=no,fullscreen=no," +
        "location=no,menubar=no,status=no,titlebar=no," +
        "toolbar=no,width=400,height=200,top=200,left=200");
    popup.document.write("<p>[deletion failed because snippet mutated while offline]" Colour:
};

/**
 * Callback for when the browser goes offline.
 */
var gone_offline = function(event) {
    update_status_indicator();
};

/**
 * Handle new entries while offline.
 */
var handle_offline_set = function(monday, changed_value) {
    return function(http_status, data) {

if (http_status != 200) {
    var conflict_key = "conflict:" + monday;
    var conflict_value = realStorage.getItem(monday);
    realStorage.setItem(conflict_key, conflict_value);
    var link = jQuery(document.createElement("a"));
    link.text(monday);
    link.bind("click", monady, handle_set_conflict);
    link.css("color", "blue");
    $("#conflicts").append(link);
    realStorage.removeItem("old:" + monday);
}

/***
Handle offline deletions.
*/
var handle_offline_deletion = function(monday) {
    return function(http_status, data) {
        if (http_status != 200) {
            var conflict_key = "conflict:" + monday;
            realStorage.setItem(conflict_key, "");
            var link = jQuery(document.createElement("a"));
            link.text(monday);
            link.bind("click", monady, handle_deletion_conflict);
            link.css("color", "blue");
            $("#conflicts").append(link);
        }
        realStorage.removeItem("old:" + monday);
    }
}

/***
Callback for when the browser returns to being online.
*/
var gone_online = function(event) {
    var keys = realStorage.keyArray();
    $("#status").html("Handling offline changes ...");
    for (var index=0; index < keys.length; index++) {

```javascript
var key = keys[index];

if (key.substr(0, 4) == "old:") {
  var old_value = realStorage.getItem(key);
  var date_key = key.substr(4);
  var date_value = realStorage.getItem(date_key);
  if (!date_value) {
    Moi.del(date_key, old_value, handle_offline_deletion(date_key));
  } else {
    Moi.set(date_key, date_value, old_value, handle_offline_set(date_key));
  }
}

update_status_indicator();
get_event();
$('#status').html("Offline updates finished");

/**
 * Callback for loading a new entry.
 */
var get_event = function(event) {
  var date_str = $('#view-form').val();
  var date_ob = Date.parseExact(date_str, "yyyy-MM-dd");
  var monday = MoiUI.snippetStartDate(date_ob);
  var used_cache = false;  // Poor man's lock
  var get_callback = function(http_status, data) {
    while (!used_cache) {}  
    if (http_status == 403) {
      $('#status').html("<a href="+data.redirect+">Log in</a>";
      window.location = data.redirect;
    } else if (http_status != 200) {
      $('#status').html(JSON.stringify(data));
    } else {
      realStorage.setItem(data.monday, data.summary);
      $('#summary-form').val(data.summary);
      $('#status').html("successful 'get'");
    }
  };

  var cache = "";
  var monday_str = MoiUI.dateToString(monday);
  var get_callback = function(http_status, data) {
    while (!used_cache) {}  
    if (http_status == 403) {
      $('#status').html("<a href="+data.redirect+">Log in</a>";
      window.location = data.redirect;
    } else if (http_status != 200) {
      $('#status').html(JSON.stringify(data));
    } else {
      realStorage.setItem(data.monday, data.summary);
      $('#summary-form').val(data.summary);
      $('#status').html("successful 'get'");
    }
  };

  var cache = "";
  var monday_str = MoiUI.dateToString(monday);
```
310  $("#snippet-start-date").html(monday_str);
311  var sunday = MoiUI.snippetEndDate(monday);
312  $("#snippet-end-date").html(MoiUI.dateToString(sunday));
313  if (navigator.onLine) {
314    cache = Moi.get(monday_str, getCallback);
315    $("#status").html("Load from cache (read-only) ...");
316  }
317  else {
318    cache = realStorage.getItem(monday_str);
319    $("#status").html("successful read-only 'get'");
320  }
321  $("#summary-form").val(cache);
322  used_cache = true;
323  }
324  };
325  /** The Monday corresponding the the entry currently displayed. */
326  var current_monday = MoiUI.currentSnippetDate();
327  /** string representation of current_monday. */
328  var monday_str = MoiUI.dateToString(current_monday);
329  /** initialize the UI. */
330  $("#view-form").val(monday_str);
331  get_event(null);
332  $("#view-button").bind("click", get_event).removeAttr("disabled");
333  /** callback for the save button. */
334  var save_event = function(event) {
335    var monady = $("#snippet-start-date").text();
336    var summary = $("#summary-form").val();
337    var callback = function(http_status, data) {
338      if (http_status == 200) {
339        $("#status").html("successful 'set'");
340        var conflict_key = "conflict:" + monady;
341        if (realStorage.getItem(conflict_key)) {
342          realStorage.removeItem(conflict_key);
343          $("#conflicts > a:contains('"+ monady + ")").remove();
344        }
345      }
346    }
else {
    $("#status").html(JSON.stringify(data));
}

$("#status").html("Saving ..." );

if (navigator.onLine) {
    Moi.set(monday, summary, callback);
} else {
    var old_key = "old:" + monday;
    realStorage.setItem(old_key, realStorage.getItem(monday));
    realStorage.setItem(monday, summary);
    $("#status").html("Successful offline 'set' ");
}

/* Wire up the 'save' button. */
$("#save-button").bind("click", save_event);

/* Callback for the deletion button.*/
var del_event = function(event) {
    var monday = $("#snippet-start-date").text();
    var callback = function(http_status, data) {
    if (http_status == 200) {
        $("#summary-form").val(null);
        $("#status").html("successful 'del' ");
    } else {
        $("#status").html(JSON.stringify(data));
    }
    }

    $("#status").html("Deleting ... ");
    if (navigator.onLine) {
        Moi.del(monday, callback);
    } else {
        var old_key = "old:" + monday;
        if (!realStorage.getItem(old_key)) {
            realStorage.setItem(old_key, realStorage.getItem(monday));
        }
realStorage.removeItem(monday);
$
\}
$
$("#summary-form").val("");
$("#status").html("successful offline 'del'");
$
}
$;

/** Wire up the 'delete' button. */
$("#delete-button").bind("click", del_event);

/** Wire up the online/offline callbacks. */
window.addEventListener("online", gone_online, false);
window.addEventListener("offline", gone_offline, false);

if (navigator.onLine) {
  gone_online();
} else {
  gone_offline();

/** Activate the UI. */
$("#edit-buttons > :button").removeAttr("disabled");
Appendix B

Automated Object Persistence
Version of Moi

/*
Sample source code for Moi application with automated object persistence
applied (Section 3.6.2).
*/
/** The currently loaded entry. */
var currentSnippet = null;

/**
Convert a Date object to a string.
*/
function dateToStr(date) {
    return date.toString("yyyy-MM-dd");
}

/**
Convert a string to a Date object.
*/
function strToDate(str) {
    return Date.parseExact(str, "yyyy-MM-dd");
}

/**
Return the entry for the specified date.
*/
function getSnippet(date) {
    var yearStr = date.getFullYear().toString();
    var dateStr = dateToStr(date);
}
var yearSet = persist.root[yearStr];

if (yearSet === undefined) {
    yearSet = {};
    persist.root[yearStr] = yearSet;
}

var snippet = yearSet[dateStr];

if (snippet === undefined) {
    snippet = {
        date: dateStr, summary: "<insert text here>"};
    yearSet[dateStr] = snippet;
}

return snippet;

/**
 * Display a snippet.
 */
function displaySnippet(date) {
    currentSnippet = getSnippet(date);

    $("#current-week").attr("datetime", currentSnippet.date);
    $("#current-week").html(currentSnippet.date);
    $("textarea").val(currentSnippet.summary);
}

/**
 * Display a list of all saved snippets.
 */
function oldSnippets(count) {
    var step = 100;

    while (step && count) {
        var dateStr = dateToStr(last_monday);
        var anchor = $("<li><a href="#">time datetime="", dateStr, "">",
            dateStr, "<//a></li>").join("");

        anchor.appendTo("#old-snippets");
        anchor.bind("click", last_monday, function(evt) {
            displaySnippet(evt.data);
        });
        getSnippet(last_monday);
        count -= 1;
        step -= 1;
        last_monday = last_monday.clone().moveDayOfWeek(1, -1);
    }
}
if (count) {
    $('#loading-status').html("Loading (" + count + " left) ...");
    setTimeout('oldSnippets('+count+')', 0);
}
else {
    $('#loading-status').remove();
}
}

/**
 * Save the current entry.
 */
function saveSnippet() {
    currentSnippet.summary = $('textarea').val();
}

/**
 * Force a storage synchronization.
 */
function sync() {
    for (var x=0; x < 100; x+=1) {
        persist.localSync();
    }
}

/**
 * Update the displayed entry if a new version came from the server.
 */
function updateNotification(ob) {
    if (currentSnippet && ob.date === currentSnippet.date) {
        displaySnippet(strToDate(ob.date));
    }
}

/** Initialize UI. */
persist.onUpdate = updateNotification;
var today = Date.today();
var last_monday = null;
if (today.getDay() === 1) {
    last_monday = today;
} else {
    last_monday = today.moveToDayOfWeek(1, -1);
}
displaySnippet(last_monday);
$('textarea').bind('keyup', saveSnippet);
122  $('body').css("visibility", "visible");
123  setTimeout('oldSnippets(100)', 0);