CROSSCUTTING SOFTWARE ARTIFACTS FOR ACCESS CONTROL

APPROVED BY SUPERVISING COMMITTEE:

Jianwei Niu, Ph.D.

Kay Robbins, Ph.D.

Greg White, Ph.D.

Jeffery von Ronne, Ph.D.

Ram Krishnan, Ph.D.

Accepted: Dean, Graduate School
DEDICATION

For Dawn, Sarah, and Jake.
CROSSCUTTING SOFTWARE ARTIFACTS FOR ACCESS CONTROL

by

MARK ROBINSON, M. SC.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
May 2013
ACKNOWLEDGEMENTS

First of all, I am thankful to Dr. Jianwei Niu, my Ph.D. advisor, who provided endless guidance, support, and enthusiasm. Over the past few years, you truly helped me to believe in myself as a researcher and kept me focused on the finish line. I would also like to thank my previous undergraduate and graduate advisors, Dr. Kay Robbins, Dr. Gerald Pitts, and Barb Latimer, for their kindness and assistance in getting me to this point.

Secondly, I thank my parents for their love and support throughout the years. You provided me with so much during my youth and I did not always seem appreciative. All that is good in my life I owe to both of you.

I am so thankful to my beloved wife, Dawn. Your understanding and motivation kept me going during the rough spots. And thank you so much for running interference for me these past few months with the kids. My sanity remains intact.

Last, but not least, I thank God for... pretty much everything. Through You all things are possible.

May 2013
Today’s techniques for software artifact access control bear an increased maintenance cost due to their coarse granularity and limited expressive power. As a result, the implemented access control policies can be incorrect or incomplete, creating security vulnerabilities. Furthermore, the additional maintenance of access-restricted artifacts may negatively influence developers’ decisions for collaboration opportunities. This thesis proposes a crosscutting concern-based approach to a software artifact access control model that can reliably enforce access control, reduce maintenance, and increase the types of policies that can be expressed. We implemented our approach as a front-end integrated development environment, SaJE, and a back-end access control monitor, GitBAC. We evaluated our implementation in two laboratory studies and a human subject experiment, measuring reliability, maintenance, and usability. The results from our evaluation indicate that crosscutting concerns are an effective means of implementing software artifact access control, offering improvements over conventional techniques.
# TABLE OF CONTENTS

Acknowledgements ........................................................................................................ iv

Abstract .......................................................................................................................... v

List of Tables .................................................................................................................. xi

List of Figures .................................................................................................................. xii

Chapter 1: Introduction ................................................................................................. 1

1.1 Background ............................................................................................................... 4

1.1.1 Insider Threat ...................................................................................................... 4

1.1.2 Access Control .................................................................................................. 5

1.1.3 Collaboration as a Business Model .................................................................. 6

1.1.4 Crosscutting Concerns .................................................................................... 6

1.2 Related Work .......................................................................................................... 7

1.2.1 Conventional Techniques for Software Artifact Access Control ................. 8

1.2.2 Crosscutting Techniques ................................................................................ 8

1.2.3 Access Control Models ................................................................................... 9

1.3 Motivation ............................................................................................................... 9

1.3.1 Increased Maintenance Overhead .................................................................. 9

1.3.2 Limited Policy Implementation ....................................................................... 10

1.4 Approach .............................................................................................................. 10

1.4.1 Thesis .............................................................................................................. 11

1.4.2 Methodology .................................................................................................. 11

1.5 Evaluation ............................................................................................................ 11

1.5.1 Reliability Study ......................................................................................... 12
## Chapter 3: Approach

3.1 An Example: FooSoft ................................................................. 35
3.2 Using Crosscutting Concerns for Access Control .......................... 36
3.3 Design Goals ................................................................. 37
  3.3.1 Confidentiality ......................................................... 38
  3.3.2 Integrity/Reliability .................................................... 39
  3.3.3 Non-repudiation/Accountability ................................ 39
  3.3.4 Transparency .......................................................... 39
  3.3.5 Usability ................................................................. 39
  3.3.6 Flexibility ................................................................. 40
3.4 Overview of our Concern Model .............................................. 40
  3.4.1 Fragment Granularity .................................................. 42
  3.4.2 Fragment Annotation ................................................... 42
  3.4.3 Fragment Maintenance Heuristics ................................ 42
  3.4.4 The Access Control Monitor ....................................... 44
  3.4.5 Flexible Access Control Policies ................................ 45
  3.4.6 Audit Log ................................................................. 48
3.5 Enforcing Access Control ...................................................... 48
  3.5.1 Protecting Target Code ............................................. 49

## Chapter 4: Implementation ...................................................... 51

4.1 Front-end: SaJE ............................................................... 53
  4.1.1 Fragment Specification ............................................... 54
  4.1.2 Assigning Access Permissions to a Fragment ................... 55
  4.1.3 Access-aware Execution .......................................... 57
4.2 Back-end: GitBAC ............................................................... 61
  4.2.1 Overview of the GitBAC System ................................ 61
4.2.2 Access Control ................................................................. 64
4.2.3 Importing an Access-controlled Project using GitBAC ................. 69
4.2.4 Pushing Updates to a Project through GitBAC .......................... 69
4.2.5 Transforming Artifacts ....................................................... 70

Chapter 5: Evaluation .................................................................. 73
5.1 Study 1: Fragment Correctness .................................................. 74
  5.1.1 Method ......................................................................... 74
  5.1.2 Results ........................................................................ 77
5.2 Study 2: Maintenance Effect ...................................................... 78
  5.2.1 Method ......................................................................... 78
  5.2.2 Results ........................................................................ 80
5.3 Study 3: Usability .................................................................... 82
  5.3.1 Description of Subjects ....................................................... 83
  5.3.2 Method ......................................................................... 83
  5.3.3 Access control training ...................................................... 83
  5.3.4 Experimental Tasks .......................................................... 85
  5.3.5 Exit Questionnaire ............................................................ 87
  5.3.6 Access Control Model ....................................................... 88
  5.3.7 Metrics ......................................................................... 89
  5.3.8 Results ........................................................................ 90
  5.3.9 Discussion ..................................................................... 96
5.4 Evaluation Summary ............................................................... 98

Chapter 6: Conclusion ................................................................. 99
6.1 Future Work .......................................................................... 100
  6.1.1 Mainstream IDE Support .................................................. 100
  6.1.2 Automatic Mock Generation ............................................. 100
LIST OF TABLES

Table 3.1  Software artifact access control SAAC₁ configuration using ABACₐ . . . . 47
Table 4.1  GitBAC access control group attributes. . . . . . . . . . . . . . . . . . . . 65
Table 5.1  Study 1 character assignment. . . . . . . . . . . . . . . . . . . . . . . . . . . 75
Table 5.2  Study 1 access control policy for GitBAC test artifact a₁. . . . . . . . . . . 75
Table 5.3  Study 2 results. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
Table 5.4  Software artifact access control SAAC₀ configuration using ABACₐ . . . . 88
Table F.1  Software artifact access control SAACcw configuration using ABACₐ . . . 129
LIST OF FIGURES

Figure 3.1  Restricting access to the FooSoft codebase using a conventional technique.  . 34
Figure 3.2  Further decomposition of the FooSoft codebase.  . . . . . . . . . . . . . . . 36
Figure 3.3  The FooSoft access control policy as a crosscutting concern.  . . . . . . . . . 37
Figure 3.4  Concern Model Overview.  . . . . . . . . . . . . . . . . . . . . . . . . . . 41
Figure 3.5  Condensed FooSoft access control policy.  . . . . . . . . . . . . . . . . . . 46
Figure 3.6  Enforcing an Access Control Policy.  . . . . . . . . . . . . . . . . . . . . 49
Figure 4.1  Conventional and crosscutting access control techniques.  . . . . . . . . . . 52
Figure 4.2  The SaJE IDE.  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
Figure 4.3  Visualization of fragments in the SaJE source code editor.  . . . . . . . . . 55
Figure 4.4  Creating a fragment in SaJE.  . . . . . . . . . . . . . . . . . . . . . . . . 56
Figure 4.5  Assigning an access group to a new fragment.  . . . . . . . . . . . . . . . . 56
Figure 4.6  A new fragment with Public access in SaJE.  . . . . . . . . . . . . . . . . . 57
Figure 4.7  Access-aware execution menu options in SaJE.  . . . . . . . . . . . . . . . 58
Figure 4.8  Source code composition that cannot be compiled without access awareness. 59
Figure 4.9  Compilable source code with mutually exclusive access-restricted fragments. 60
Figure 4.10  Output of the example program executing as an Employee.  . . . . . . . . 60
Figure 4.11  GitBAC overview.  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
Figure 4.12  GitBAC import sequence diagram.  . . . . . . . . . . . . . . . . . . . . . 69
Figure 5.1  Message for removed GUI feature.  . . . . . . . . . . . . . . . . . . . . . 79
Figure 5.2  Study 3 Task 1 access control groups.  . . . . . . . . . . . . . . . . . . . . 86
Figure 5.3  Study 3 Task 3 access control groups.  . . . . . . . . . . . . . . . . . . . . 87
Figure 5.4  Study 3 average experience levels by technique.  . . . . . . . . . . . . . . . 91
Figure 5.5  Study 3 average task completion time by technique.  . . . . . . . . . . . . 92
Figure 5.6  Study 3 average number of change operations by technique.  . . . . . . . . 93
Figure 5.7  Study 3 average number of failures by technique. .............................. 94
Figure 5.8  Study 3 average technique usability scores. ................................. 95
Figure 5.9  Study 3 Conventional time and operations correlation. .................... 97
Figure 5.10 Study 3 Concern time and operations correlation. .......................... 98

Figure E.1 Sample project in the SAJE IDE. .................................................. 119
Figure E.2 Running a source file in SAJE. ..................................................... 119
Figure E.3 The program output pane in SAJE. ............................................... 120
Figure E.4 Creating a new source file in SAJE. ............................................. 120
Figure E.5 Naming a new source file in SAJE. ............................................. 120
Figure E.6 Viewing the new source file in the SAJE editor. ............................ 120
Figure E.7 Copying a directory in SAJE. ...................................................... 121
Figure E.8 Pasting a directory in SAJE. ....................................................... 121
Figure E.9 Renaming a directory in SAJE. .................................................... 121
Figure E.10 File-level decomposition to control access of a restricted artifact. .... 122
Figure E.11 Applying directory-level access control in SAJE. ......................... 122
Figure E.12 Selecting the access group. ....................................................... 123
Figure E.13 Public and private restricted directories in SAJE. ......................... 123
Figure E.14 Output after executing the public variant of AppMain in SAJE. ....... 124
Figure E.15 Restricting access to a code fragment. ....................................... 124
Figure E.16 Selecting the access group for the code fragment. ....................... 125
Figure E.17 Public and private restricted fragments in the SAJE editor. .......... 125
Figure E.18 Executing the public fragment of AppMain in SAJE. .................... 125
Figure E.19 Output after executing the public fragment of AppMain in SAJE. .... 126

Figure G.1 Demo codebase restricted using a SAAC_0 policy. ......................... 131
Figure G.2 Policy menu for demo. ............................................................. 131
Figure G.3 Extended run menu for demo. .................................................... 131
Figure G.4  Demo SAAC\textsubscript{0} policy. ........................................ 132
Figure G.5  Demo SAAC\textsubscript{1} policy. ........................................ 133
Figure G.6  Bob logged in with a non-local IP address. ......................... 133
Figure G.7  Bob logged in with a local IP address. ............................. 133
Figure G.8  Demo DAC policy. ......................................................... 134
Figure G.9  DAC Admin view of the demo codebase. ......................... 134
Figure G.10 Bob’s view of the DAC-restricted codebase. ............... 134
Figure G.11 Sue’s initial view of the DAC-restricted codebase. ....... 135
Figure G.12 Bob granting Sue access to his fragments. ................. 135
Figure G.13 Sue’s revised view of the DAC-restricted codebase. ....... 135
Chapter 1: INTRODUCTION

There are times when software developers must share only a portion of their software artifacts with other developers (e.g., contract work, collaboration with the public, code review, testing). These situations require that the developers define and implement a means to control access to the subset of their software’s features that they wish to make available to others. A software artifact is a general term describing the objects that are produced during the software development process, e.g., source code files or design documents. It is common for software features to crosscut, or share files with other features. A single crosscut is referred to as a concern, and can represent a single feature in its entirety, a task, or other non-modular concepts. Because a file system’s finest level of granularity for access control is a whole file, if a developer wishes to restrict access to a shared file, he/she must duplicate the file and remove the restricted feature code, or fragment, from the duplicate. The coarse granularity of file systems for software artifact access control creates an additional maintenance burden as any change to a duplicated file must be replicated among all duplicates in order to remain consistent throughout the source codebase.

An example can better illustrate this scenario. FooSoft is a fictitious software development company with a closed source software product (i.e., one must pay for the source code). In order to increase sales and product development, FooSoft decides to offer a free, or open, variant of their product’s source code and charge for the upgraded variant (their original product). They determine a subset of their product’s features to remove from the open source code and create a simple access control policy using two access groups (public and private) to document which source code files should be excluded from the public, or free, users. Because FooSoft believes its access control needs are simple, they elect to copy the source code files containing the code for the removed features into a separate directory and restrict access to that directory. All remaining code will be shared between public and private developers. However, not all of the restricted files pertain solely to the removed features, i.e., the removed features crosscut some of the files, sharing the files with other features or functions. So that public users can still build and execute the open source...
product, all crosscut files must be duplicated, stripped of restricted content, and distributed to the public users. Whenever non-restricted changes are made to either the public or private duplicates, FooSoft must identify and manually merge the changes into the other distribution. FooSoft handles the additional work necessary to maintain consistency in its public and private distributions by increasing the job responsibilities of their own internal developers.

Later FooSoft hires a contractor to work on a new module for their software product. Fearing theft of intellectual property, FooSoft decides to restrict access to their proprietary closed source algorithms for private developers who work outside of the FooSoft office. Developers who work outside of the office will only be allowed to use a testing version of the algorithms, while developers who work inside the office will have access to the original algorithms. FooSoft’s file system uses a static, built-in access control model with no ability to add new access control variables, such as user location. FooSoft must either leave the location-based change to their security policy unimplemented, or layer an additional access control system on top of their file system (e.g., a VPN).

An access control model is a guideline for implementing, or expressing access control policies. A model provides a set of characteristics for describing who can access what, and in what capacity. The most commonly used access control models today (DAC, MAC, and RBAC) are well-defined, but limited in their ability to accommodate new characteristics for the users and objects that an access control policy describes. This, in turn, limits the expressive capabilities of the system incorporating the model (e.g., a file system) and developers are left to ignore that aspect of their security policy, or supplement their access control system with another access control system. While adding another access control system may address the immediate needs of the security policy, users may be required to possess multiple sets of credentials and administrators will have to maintain the access control systems separately. Lastly, if the supplemental system uses a static access control model, then there are no guarantees that the patchwork of access control systems can accommodate further changes to the security policy in the future.

The additional software artifact maintenance and policy expression limitations of today’s ac-
cess control techniques may seem insignificant, but the additional work imposed by conventional techniques increases project overhead and the likelihood of inconsistent code and security mistakes. The specter of additional work to support access control-mandated modularization may also discourage satisfaction of the security policy, if satisfaction is even possible given the access control technique and model used. And as access control lies in the realm of computer security, a single security incident from an incomplete or incorrect access control policy can be a catastrophic one. Moreover, the increased work to maintain access control of crosscutting features may prevent those features from selection for collaboration or revenue-generating opportunities.

To address these problems, this thesis proposes a new access control approach using a crosscutting technique to implement access control policies of software artifacts. Our method involves the use of a concern model that allows one to assign access restrictions to source code fragments (i.e., sections of source code smaller than the containing files), automatically and reliably maintains fragments through a set of update heuristics, and enforces a flexible access control policy on source code transparently between developers and their source code repository. We have implemented a prototype for our technique in two parts: a simple Java integrated development environment (SaJE) and an intermediary server process (GitBAC) that sits between SaJE and the source code repository. SaJE provides traditional Java editor features, as well as functionality and visualization for restricting access to files, directories, and lines of source code. GitBAC is an access control monitor that reliably enforces a customizable access control policy between the users and the repository.

We evaluated our technique’s effectiveness with two case studies and a human subject experiment. Our first study evaluated the correctness of fragment maintenance over a large number of source code updates. The second study evaluated the effect of a conventional access control technique on artifact maintenance by mining the change history of three open source projects. Our third study evaluated the efficacy and usability of crosscutting concerns, versus a conventional file-based technique, in maintaining an access control policy on a codebase as changes occur. The results from these studies show that our concern-based technique is a reliable means for software artifact access control, that our technique outperforms conventional file-based techniques by re-
ducing maintenance time, operations, and failures, and is generally preferred to a conventional technique.

## 1.1 Background

Developing software as a team has several advantages over solitary development: increased productivity, broader creative input, and faster speed to-market. In fact, software engineering teams are the only way to tackle software projects more complex than garage programs, i.e., Brooks’ programming products and systems [12]. But not all members of a development effort should necessarily have complete access to all software artifacts, especially those containing sensitive company or client information, such as proprietary algorithms and database credentials. To protect its critical software artifacts, a company must define an access control policy, describing which team members require access to which artifacts with as much precision and detail as possible.

### 1.1.1 Insider Threat

The Carnegie Mellon Computer Emergency Response Team (CERT) defines a malicious insider (or simply “insider”) as a current or former worker (employee, contractor, or partner) who misuses his/her authorized access to a company’s information system resources in a negative manner [15]. In 2010, Sergey Aleynikov, working as a programmer for Goldman Sachs, transferred thousands of Goldman Sachs’ proprietary software files with the intent to deliver the code to one of their competitors [50]. Société Générale had a proprietary trading algorithm stolen by one of their programmers, Samarth Agrawal, in 2011. The company calculated the damage from the theft at $9.58 million dollars [18]. Bo Zhang, a Chinese contract programmer, was convicted in 2012 of stealing proprietary source code from the Federal Reserve Bank of New York [11]. While these insiders undoubtedly required access to some part of the source code to perform their programming duties, tighter access control strategies for the software artifacts could have reduced the impact of these incidents.

A 2008 CERT insider threat study estimates that 27% of the electronic intrusion incidents
reported during the summer of 2006 originated from insiders and 15% were of unknown origin [40]. The study recommends that organizations should carefully tune fine-grained access control so that users only have access to the data necessary to accomplish their duties. A 2009 study from the CERT cites that 26% of their intellectual property (IP) theft cases (8% of their total cases) involve insider theft of proprietary software or source code [54]. In 2011, Shaw and Stock examined over 550 insider theft cases from multiple sources, finding that 34% of the cases involving theft of trade secrets pertained to theft of source code and proprietary software [71]. Ponemon’s 2012 study on insider fraud surveyed 743 IT and security professionals (most having a supervisory or executive position), showing that 33% of the respondents felt that the risk of insider theft had worsened, an increase from 23% in their 2011 study [46].

1.1.2 Access Control

Access control embodies the concepts of least privilege and separation of duties, dictating who has access to which resources and in what capacity, in accordance with a well-defined security policy. Thus, access control is responsible for both restricting the access of authorized people and processes, and preventing access by unauthorized people and processes, to protected resources. Generally, there are three components to access control: the subject, the object, and the access right [75]. The subject is the entity (e.g., user, process, sets of entities) who wishes to perform a specific action (e.g., read a file, execute a program) on a particular resource (the object). The access right grants or denies the specified action. In the absence of a rule that precisely applies to a desired action, the access control policy can define default behavior (e.g., deny all actions by default except those listed).

Three of the most commonly used models for access control of computer objects (e.g., files and directories) are Discretionary Access Control (DAC), Mandatory Access Control (MAC), and Role-Based Access Control (RBAC). DAC allows users and groups to determine who can access which object. MAC determines access by matching a user or program’s access level with the security classification of the requested object. Sandhu, et al. describe RBAC models that incorporate
hierarchical relationships and configurable constraints between roles and users [69]. Access control models provide the guidelines, constraints, and characteristics by which access control policies are expressed. A policy is a specific description of which subjects are allowed (or not allowed) to access which resources and is based on a model. Access control policies are enforced by an access control monitor. A monitor is the sole interface between subjects and objects, and grants or rejects a subject’s access request to an object by consulting a configured policy. A monitor may be a standalone system (e.g., a VPN server), or built into other systems (e.g., file systems).

1.1.3 Collaboration as a Business Model

Software artifact access control is required in order to utilize certain business models, such as Freemium open source software. Open source software extends the membership of the development team to the general public. But in a Freemium project, a company removes selected for-pay premium features from a freely available public variant. The goals of this business model are to accelerate development, promote adoption, and increase sales by enticing the free users to purchase the premium product in order to gain access to the restricted features. For example, Tasktop is a closed source application lifecycle management product that offers its core codebase as an open source product, known as Mylyn. Anyone can use and contribute to Mylyn’s development, but must pay for access to Tasktop’s restricted features, such as instant desktop file search, enhanced time tracking, and export capabilities [77].

1.1.4 Crosscutting Concerns

Developers typically modularize their source code based on a dominant decomposition (e.g., by functionality) described within the design of the software they are building [76]. However, the selected modularization does not always suit all of the software’s design properties (i.e., the properties do not compose in the same manner). Properties that do not cleanly fit the dominant decomposition are referred to as crosscutting properties. While crosscutting properties may represent important design goals, they can be scattered across a program’s modules and tangled with other
non-concern code. A common example of crosscutting is a feature that possesses hooks into an application’s graphical user interface (GUI), e.g., a drop-down menu item to activate a particular feature. Developers tend to centralize GUI-related code for all of a software program’s features that have GUI-based interaction (i.e., developers have a tendency to include the GUI as part of the software’s dominant decomposition). Research into modularization behavior shows that crosscutting is normal in software development although each crosscut encompasses only a small number of classes and a small amount of code per class (6% on average) [48]. Access control policies may also be viewed as crosscutting concerns as access-restricted code usually involves features or smaller artifact fragments, such as credentials (e.g., database login and password).

1.2 Related Work

Current techniques for software artifact access control have drawbacks when the source code that must be restricted crosscuts a codebase. Some techniques are well-suited to handling crosscuts of source code but lack mechanisms for access control enforcement among developers. Additionally, the access control models that current techniques utilize lack the flexibility and expressiveness necessary to accommodate a wide range of security policies. As such, the access control specifications of such security policies are either incorrectly or incompletely implemented [66].

Our systematic literature review found numerous articles regarding insider threat, crosscutting concerns and techniques, and access control models. These are discussed at length in this introduction and the Background chapter. We found no previous work that approaches or implements access control of software artifacts as a crosscutting concern. We primarily used the ACM Portal and IEEE Xplore databases, but also searched product websites, special interest publications (e.g., the Carnegie Mellon CERT reports), and reputable news sources for insider arrest information (e.g., the Wall Street Journal). We favored scholarly articles from top Software Engineering conferences (i.e., ICSE, ASE, and FSE), relevant security conferences (e.g., SACMAT), ACM and IEEE journals, and subject matter experts, e.g., Gail Murphy, Martin Robillard, and the CERT. We excluded papers that were more than 15 years old except for papers that are recognized as seminal.
contributions. Keywords used in our searches were crosscutting, concern, access control, access control model, insider threat, insider theft, aspect oriented programming, feature, traceability, version control, source control management, source code management, configuration management.

1.2.1 Conventional Techniques for Software Artifact Access Control

Current techniques for controlling access to software artifacts involve physically separating a project’s source code into modules (e.g., separate projects or test-only libraries), where each module belongs to a specific level of access control (or access control hierarchy) [66]. Physical separation of artifacts requires additional management to maintain consistency of changes between duplicated artifacts and additional communication when interfaces change. While physical separation techniques can provide adequate developer security for some software projects, they suffer from inflexibility when an access control policy changes or when a policy has to be retroactively applied to a project already in progress. Because these techniques have a file-level granularity, artifacts must be duplicated or refactored when the restricted content is smaller than an entire artifact.

1.2.2 Crosscutting Techniques

Many different techniques have been researched that involve feature crosscutting of source code, although they refer to crosscutting, and the concept represented by a crosscut, differently: concerns [28, 37, 52, 54, 64, 65], collaborations [80], feature-oriented programming [3, 5], aspect-oriented programming [38], and mixin layers [74]. We use the terms “crosscutting” and “concern” as the majority of the software engineering articles we found favor those terms.

Developers have implemented access control in software using aspects although their techniques only deal with access control in the application domain (i.e., restricting access of the users of the software to certain application functions or data) [62]. Other crosscutting techniques cannot be completely automated as they try to maintain the semantic aspect of the concern and have trouble automatically determining when new code has semantic relevance to the concern [55, 63].
1.2.3 Access Control Models

As mentioned previously, the three most common access control models are DAC, MAC, and RBAC. While these models are straightforward in their implementation, they are inflexible in their definitions of “user” and “group”. Bishop introduces the Attribute-Based Group Access Control (ABGAC) model as a flexible superset of RBAC, allowing finer-grained specification of groups, jobs, tasks, and the degree of threat presented by each included entity (user, process, etc.) [7]. Bishop states that access control models that do not support a level of granularity that well-suits the implemented security policy create gaps that can prevent an accurate specification of least privilege, increasing a system’s vulnerability to insider threat. A further generalization of ABGAC is the Attribute-Based Access Control (ABAC) model. ABAC can formally describe many of these models, such as DAC, MAC, and RBAC, and can also accommodate attributes that are difficult or impossible to represent in other models [31].

1.3 Motivation

Access control plays an integral role in protecting software artifacts from insider threat during the development process and can provide new opportunities for community collaboration and business revenue. But conventional techniques for implementing access control can increase maintenance overhead and stymie complete and correct access control policy implementation.

1.3.1 Increased Maintenance Overhead

If an access control policy perfectly fits the dominant decomposition of a project’s modularization strategy (e.g., restricting access to a plug-in’s source code) then current methods typically suffice without much additional maintenance. But, when the restricted source code is spread over multiple artifacts and mixed with non-restricted code (i.e., scattered and tangled), the coarse granularity of conventional techniques requires that artifacts be duplicated or refactored in order to preserve the confidentiality of the restricted material. Artifact duplication is simple to implement but necessi-
tates additional, ongoing maintenance of all non-restricted artifact content for consistency between the access-controlled implementations. Refactoring an artifact requires more initial work to reduce the amount of duplicated non-restricted content but may still require ongoing maintenance as some duplicated content is unavoidable (e.g., class and method declarations).

1.3.2 Limited Policy Implementation

Conventional access control techniques are based on access control models that are built into the artifact storage mechanism (e.g., database management systems, operating systems). These static models may be suitable for policies that fit cleanly to them, but they are limited in the range of policies they can implemented. This can make an access control policy impossible to implement completely, which can lead to security incidents (i.e., too little access control), missed opportunities for collaboration (i.e., too much access control), and smaller profits (e.g., unappealing feature selection for Freemium open source projects).

The motivation for our research is to improve the state of the art of software artifact access control by reducing maintenance caused by access control and increasing the expressivity of the access control mechanism. Such improvements may then lead to increased developer productivity, fewer security incidents, and more opportunities for collaboration and profitability.

1.4 Approach

Controlling access to software artifacts is an important design goal that may not cleanly fit the functional decomposition of a software product’s codebase, especially further along in the software’s life cycle and as a security policy evolves. Therefore, we consider the treatment of access control of software source code as a crosscutting property, or concern, similar to other crosscutting techniques for non-access control concerns.
1.4.1 Thesis

We propose that a precise and practical concern model and supporting tools can provide some of the benefits of modularity for non-modular design goals of software source code, such as information hiding. This concern model can be used as an access control monitor between developers and an artifact repository, instead of using conventional modularization techniques. Additional benefits are reduced artifact maintenance through increased code sharing, enhanced access control policy expressiveness, and collaborative development opportunities.

1.4.2 Methodology

We have designed a concern model, allowing the specification of access-restricted fragments within source code, and a flexible software artifact access control (SAAC) model using ABACα [31]. SAAC-based policies are enforced as the artifacts pass between developers and a source code repository. Our concern model includes a set of automatic heuristics to ensure that fragment contents are reliably maintained and a transformation mapping mechanism that automatically extracts and reintegrates restricted fragments as artifacts pass through the access control monitor. We have implemented our approach in two distinct parts: a simple integrated development environment for visualization of and interaction with access-restricted source code fragments, and an access control monitor that transparently enforces a SAAC policy between developers and the repository.

1.5 Evaluation

To determine if our technique and implementation achieve our research objectives, we conducted two case studies and one human subject experiment. The results from these evaluations show that crosscutting concerns can provide a reliable alternative for software artifact access control, reduce access control maintenance time, tasks, and errors compared to a conventional technique, and are preferred over a conventional technique.
1.5.1 Reliability Study

Our first study investigated the use of crosscutting concerns to correctly enforce access control. We use precision and recall as the basis of our metric for determination of correct access control enforcement [4]. Perfect precision and recall are necessary components for access control as imperfect results indicate a potential security breach. We measured the precision and recall of a test artifact’s fragments over a series of 1,500 random insert and delete operations, applying an access control policy and maintenance heuristics after each update. We conclude from this study that our concern-based access control monitor can reliably enforce a configured access control policy.

1.5.2 Maintenance Study

In our second study, we examined the use of a conventional, file-based modularization technique for access control of software artifacts to determine how the technique affects software maintenance tasks. We arbitrarily selected a subset of features from three open source products, jEdit 4.3, JHotDraw 7, and ArgoUML 0.35 from which public access would be removed in a pseudo Freemium distribution. After manually determining fragment boundaries for the features, we mined each product’s SVN change history, identifying which lines of changed source code were in access-restricted feature artifacts but did not pertain a restricted feature. With conventional access control techniques, these changes would have to repeated in other artifact duplicates to maintain consistency. Lastly, we applied the same access control policy using our concern-based technique, confirming that no artifact duplications were required. We conclude from this study that conventional techniques increase maintenance tasks when an access control policy crosscuts the codebase.

1.5.3 Usability Experiment

Our third study involves human subjects and the usability of crosscutting concerns as a means of access control of source code. This experiment was performed on 14 human subjects, randomly assigning each to use either a conventional access control technique (the experimental control)
or our technique. The subjects were asked to perform a series of four tasks involving access control of a toy software project and minor code changes to the source code files. We recorded the number of failed attempts each subject made, the amount of time, and the number of artifact change operations required to successfully complete each task. An exit questionnaire captured usability feedback from each subject, as well as his/her software development, security, and access control background. We conclude from this experiment that conventional access control techniques increase maintenance more than our concern-based technique. Additionally, users tended to prefer our concern-based technique over the conventional one.

1.6 Contribution

The main contributions from this research are:

1. We introduce the concept of crosscutting concerns as a means for enforcing access control of software artifacts. The design of our concern model allows access control policies for source code to be expressed as crosscutting concerns. In order for crosscutting concerns to be reliable for access control, we have defined a set of heuristics that automatically maintain fragment boundaries. We have also designed an automatic method for concern fragment extraction and weaving, similar to the techniques used in aspect-oriented programming. Papers that involve the design of our technique are:

(a) Access control fragment visualization was modeled after our work with the Business Process Execution Language (BPEL) [68].

(b) GitBAC’s design was influenced by our previous sequence diagram research, published in ICSOFT 2012 [72].

2. Our end-to-end implementation for the specification and use of crosscutting concerns for source code access control was constructed in two parts. The front-end of our implementation is an integrated development environment that allows for visualization and interaction
with access restricted fragments in the source code editor. The back-end of our implementation is a transparent access control monitor that enforces access control concerns between developers and a Git repository. As part of our front-end and back-end systems, we have created our own Software Artifact Access Control model using ABACα. Papers regarding the implementation of our research are:

(a) A paper detailing the design and implementation of GitBAC was published in ASE 2011 [67].

(b) A publication regarding the end-to-end implementation of our technique will be submitted to ASE 2013.

3. We conducted a series of extensive tests, demonstrating that our fragment heuristics and weaving techniques can effectively enforce access control policies, preserving the integrity of the restricted fragments and preventing security leaks. We performed a second evaluation using a three open source projects, showing that conventional access control techniques increase project maintenance. Lastly, we evaluated our technique against a conventional technique in an experiment involving 14 human subjects. Our concern-based technique reduced maintenance overhead and was preferred by the subjects over the conventional technique. These experimental results will be submitted for publication to ICSM 2013.

1.7 Outline

Chapter 2 provides a background for our research, including insider threat, access control of software artifacts, crosscutting techniques, and related research. In chapter 3, we describe our approach to modeling access control as a crosscutting concern and our design for a secure and usable concern-based system. We detail the implementation of our concern-based access control system in Chapter 4. Chapter 5 evaluates our implementation for concern-based access control and discusses the experimental results. We summarize our research and its contributions in Chapter 6, and propose future directions for our work.
Chapter 2: BACKGROUND

Our research draws from different areas of software engineering and computer security. Insider threat is the result of insufficient access control of software artifacts and is a significant concern to industry. Access control is the computer security mechanism that can mitigate or eliminate insider threat. There are several models upon which access control may be implemented and a few different techniques that industry uses for implementation. We conclude our background with an overview of crosscutting concerns and a literature review of additional topics relevant to access control of software artifacts and crosscutting concerns.

2.1 Insider Threat

Bishop and Gates define an insider as someone who either violates a security policy using legitimate access (e.g., leaking information to an outsider) or violates an access control policy (an expression of the security policy) by obtaining unauthorized access to a resource [8]. Our research addresses the second defining operation of insider (i.e., the obtaining of unauthorized access to a resource), protecting software development artifacts through the specification and control of access to them.

Attacks on artifacts are broadly categorized as either external incidents (e.g., through network intrusion or theft of a computer) or internal incidents (e.g., a current or ex-employee within the organization violates the security policy with respect to the software assets). Internal incidents are commonly referred to as insider attacks. The goal of our research is to provide a means by which an organization may flexibly and precisely control access to its software artifacts, making it more resistant to insider incidents that target software artifacts and with less maintenance overhead than conventional access control methods.

We do not address the threat of outsiders obtaining unauthorized access to a resource as this type of threat is usually handled in parts of the security policy that do not directly involve access control (e.g., network and computer intrusion detection and prevention systems, password policies,
etc.). We also do not address threats that fall within Bishop’s first operational definition (e.g., collusion) except with regards to providing access controls to enforce a well-defined separation of duties, thus reducing the impact of the insider attack.

Insider attacks do not only target physical technological resources and data. Attacks can also target the development of software itself. For a company that relies on a software product developed internally, the damage to software artifacts and data caused by an insider can be catastrophic.

The process of software engineering is divided into several distinct phases, typically consisting of requirements gathering, architecture and specification, design, implementation, testing, and maintenance. Some projects may coalesce phases together or further sub-divide them. Several different types of engineering artifacts are produced throughout the process, each phase tending to focus on a subset of them. Software artifacts throughout the entire software engineering process are susceptible to insider attack and can lead to theft, fraud, and/or sabotage. For example, Cappelli states in a podcast that insider attack through source code artifacts can occur during implementation, deployment, and/or maintenance [14]. In this remainder of this section, we consider source code artifacts and target code artifacts individually, and examine all other artifacts collectively. In each section we also discuss the specific attacks that can occur on each type of artifact.

2.1.1 Source Code Artifacts

The insider threat to source code artifacts is the most obvious threat in the software development process. Source code provides exact algorithm details, is human-readable, and facilitates efficient reproduction of the original software’s functionality. Because of the huge potential for damage from source code artifacts falling into hostile hands, they are typically well-protected from outsider threats, forbidding outsider access entirely. This all-or-nothing approach to access control can work very well against outsider threats but offers no protection against someone who has some amount of inside access.

Even though source code artifacts are an obvious target for insider attack during software development, companies should not feel they are immune because attacks are rare or they only target
large firms. Moore, et al. cite that 26% of the CERT’s cases (8% of all of their cases) involve insider theft of the intellectual property (IP) of proprietary software or source code [54]. They describe two different models of insiders who target IP theft of software for business advantage: the entitled independent (e.g., a disgruntled employee acting alone) and the ambitious leader (i.e., an outsider who recruits insiders to commit the theft). While 74% of the insiders stole information within the area of their job responsibility, their responsibility may have been loosely or too largely defined.

Source code artifacts are also the target of insider attack for malicious code alteration (i.e., modifying the program code to execute instructions that are harmful to the program and/or its data). A contractor at Fannie Mae inserted a logic bomb (malicious code or programs that deliver their payload when a specified condition occurs) into one of their software programs on the day that he was fired. The malicious code was detected before any damage was done but the FBI claim that the bomb “would have wiped out all of Fannie Mae’s servers had it not been discovered in time” [82]. This example illustrates the importance of prompt removal of access privileges to source code artifacts when an employee is, or going to be, terminated. Source code artifacts can also be targeted for fraud. For example, source code can possess authentication information to access a protected data store (e.g., a password for a connection to a remote credit card database). An insider can mine the source code for authentication credentials to the credit card database and then steal and sell the card data.

Another potential area of attack on source code artifacts is a version control repository. Version control systems are widely used today for team-based software development projects. They centralize and distribute software artifacts among team members, track changes to them over time, and resolve change conflicts in shared artifacts. While many version control systems provide some specification for access control of the artifacts, the granularity of specification may not allow for the precision dictated by the security policy.

When a company’s risk analysis determines that a software development project needs to be protected from the threat of insider attack, an access control strategy is developed for the project.
Source code is usually the first, and often only, artifact considered for access control. Even if an initial access control policy perfectly describes the security policy’s intent, the access control strategy must continue to be enforced. This requires a continuous effort to monitor, evaluate, modify, and update the access control strategy based on the current needs of the project. In reality, the budgetary pressures of time, money, and personnel to meet deadlines can erode an access control strategy. Additionally, some software companies may not perform adequate risk analyses or correctly specify, implement, and enforce an access control strategy. As with many security-related mechanisms, the efficacy of an access control strategy may be difficult to ascertain until an incident occurs.

Enforcement of an access control policy for software artifacts should permeate the software development life cycle (although the impact of exploit might change from one phase to another). While all artifacts are vulnerable to attack at any point during the life cycle process, source code tends to be an especially large target. Cappelli states that the majority of the cases of source code artifact attack that she examined occurred during maintenance of the software project, primarily due to relaxation of security and quality control procedures that were in place in earlier phases of development [14].

2.1.2 Target Code Artifacts

Target code is another type of artifact with the potential for a large amount of damage, if attacked. Target code is a byproduct of the transformation from source code artifacts to an executable program. Target code can be produced in many different forms. It can be a collection of machine-level instructions to be executed directly by a processor, or an intermediary to be translated by another program that will then feed instructions to a processor (e.g., Java bytecode). While target code is not as readable as the source code that produced it, target code can nonetheless convey useful and potentially damaging information to someone who knows how to interpret it.

Aside from software piracy (which is not within the scope of this dissertation), the most damaging threat to target code is reverse engineering. Reverse engineering is the process of transforming
target code back into source code. While the accuracy of the transformation is not always 100%, it may be enough to suit the perpetrator’s purpose (e.g., an accurate-enough transformation to infer a sensitive algorithm or decipher a password to a database). At first glance, the issue of reverse engineering might seem to be restricted to outsider incidents. However, even if a software company implements perfectly secure access control to a project’s source code, insiders with access to the entire target code could indirectly access the entire source codebase using reverse engineering techniques.

Main and van Oorschot describe the Insider Threat Model as one of three basic threat models against deployed software, stressing the importance of hardening a target codebase against reverse engineering attacks using techniques such as code obfuscation, transformations, and just-in-time decryption” [51]. Obfuscation and code transformation techniques try to disguise information in a program that may assist inference of the program’s semantics while maintaining program correctness (e.g., changing all variable names to random variable names, control flow transformation, dead code insertion). Just-in-time decryption decrypts an encrypted executable program as it is needed during runtime, thus requiring reverse engineering attempts to first crack the encrypted executable or intercept the decrypted program as it is running. However, these techniques are of limited use to an insider who has full access to a complete and unprotected base of target code during development.

2.1.3 Least Privilege and Separation of Duties

Source and target code artifacts can both be protected from insider threat by restricting access to the artifacts to only those developers who “need” access to them. This follows the principle of Least Privilege. Least Privilege states that an individual perform his/her duties with the least amount of privilege necessary [75]. Least privilege can thwart attacks on source and target code by preventing the insider from having access to restricted artifacts in the first place. For example, a reverse engineering attack reveals no restricted information if the target code that is attacked was generated from entirely access-appropriate source code.
CERT’s Common Sense Guide to Prevention and Detection of Insider Threats details two practices (of the sixteen recommended) that can help prevent and minimize the impact of insider attack during software development [13]. Practice 8 stresses the importance of separation of duties and least privilege. Separation of duties spreads the responsibility of critical functionality across as many people as possible, limiting the damage that a sole insider may inflict. Least privilege states that authorized insiders have access only to the resources they require to successfully perform their duties. As an insider attack can occur at any point during software development, separation of duties and least privilege should extend throughout the entire software development life cycle, from requirements to maintenance. Thus, specifying and enforcing least privilege with regards to source and target code can stymie insider attack.

2.1.4 Other Engineering Artifacts

While source code and target code artifacts are the two largest targets of insider attack, all of the artifacts in a software engineering process are a potential candidate for attack. For example, a specifications document may reveal a runtime vulnerability that can be sold to outsiders interested in an exploit. Certain test cases may be created that require the entire target codebase to run, thus indirectly exposing the software to reverse engineering. While the attacks on the examples are not as simple as attacking the source or target code directly, they may nonetheless yield results with similar damaging impact. Additionally, these other artifacts may be generally overlooked as threat candidates because of their seemingly innocuous place in development. But as with source and target code artifacts, controlling access to all artifacts using the principles of separation of duties and least privilege will help harden the entire software engineering process against the possibility of insider attack.

2.2 Access Control

Access control embodies the concepts of least privilege and separation of duties, dictating who has access to which resources and in what capacity, in accordance with a well-defined security
policy. Thus, access control is responsible for both restricting the access of authorized people and processes, and preventing access by unauthorized people and processes, to protected resources.

Generally, there are three components to access control: the subject, the object, and the access right [75]. The subject is the entity (e.g., user, process, sets of entities) who wishes to perform a specific action (e.g., read a file, execute a program) on a particular resource (the object). The access right grants or denies the specified action. In the absence of a rule that precisely applies to a desired action, the access control policy can define default behavior (e.g., deny all actions by default except those listed).

2.2.1 Application to Insider Threat

We have shown that least privilege can reduce the threat of insider attack on software development artifacts. Access is limited to only those resources (and modes of access) that insiders require for performing their duties. More then one-half of the CERT’s studied cases involved an insider acting alone. Separation of duties can further reduce the insider threat by limiting the amount of responsibility any one insider possesses. Insiders must then resort to collusion in order to achieve their malicious goals. The CERT reports that fewer than half of the cases it studied involve insider collusion [13]. The percentage is higher for crimes for financial gain than other types of insider IT crimes.

Both of these principles of computer security involve the restriction, or controlling, of access that individuals have to software development artifacts. Therefore, access control techniques can reduce the threat of insider attack in software development. The CERT supports this claim, stating that access control is a critical component of the mitigation of the risk of insider attack [13].

2.2.2 Access Control Models

Three of the most commonly used models for access control for computer objects (e.g., files and directories) are Discretionary Access Control (DAC), Mandatory Access Control (MAC), and Role-Based Access Control (RBAC). DAC allows users and groups to determine who can access which
object. DAC is flexible but can be difficult to manage with large numbers of artifacts, users, and groups. MAC determines access by matching a user or program’s access level with the security classification of the requested object. MAC is simple to apply but lacks flexibility. Sandhu, et al. describe RBAC models that incorporate hierarchical relationships and configurable constraints between roles and users [69]. RBAC promotes flexible maintenance and self-administration of user/role assignments and access control permissions. ABAC, ABGAC, and g-SIS are three more recent access control models that present additional control mechanisms for managing user and group access to software artifacts. ABAC, or Attribute-Based Access Control, is a very flexible and descriptive access control approach, but does not have a clear and widely accepted definition. Jin, et al. formalize a variant model of ABAC, called ABAC$\alpha$, that possesses enough definition to describe the DAC, MAC, RBAC$_0$ (flat), and RBAC$_1$ (hierarchical) models [31]. The Attribute-Based Group Access Control (ABGAC) model is a superset of RBAC, allowing finer-grained specification of jobs, tasks, and the degree of threat presented by each included entity (user, process, etc.) [7]. Krishnan, et al. formally describe the temporal nature of users sharing information in isolated groups using a Group-Centric Secure Information Sharing (g-SIS) model and first-order linear temporal logic [42]. They demonstrate g-SIS on a small system of groups and objects but can generalize their approach to larger systems.

2.2.3 Techniques for Software Artifact Access Control

There are three common, currently used techniques for controlling access during the software development process within industry:

1. A separate collection of source artifacts (e.g., project) for each level of access, integrating together through target code. Developers with lower levels of access do not see restricted source artifacts, but may have access to restricted target code for complete or partial integration testing. This technique is simple to implement and can control access to entire source artifacts. However, overlapping artifacts between access levels can create duplications and update inconsistencies. Also, access to target code is not controlled, allowing opportunities
for reverse engineering.

2. Fake integration points. Restricted artifacts are represented with non-sensitive interface stubs. These stubs are not the real artifacts, but merely mock-ups. Developers with lower access privilege may build and test using the stubs but never have access to sensitive source nor target code. Integration testing and deployment of the complete product is handled by a separate team with the highest level of access. This access control technique is the most effective, protecting both source and target code, but requires the greatest amount of expense, planning, and maintenance.

3. Environment virtualization. This technique is typically used in conjunction with one of the other two techniques. If virtualization is paired with separated source artifacts, this technique can restrict access to the target code and build environment. When virtualization is paired with the technique of fake integration points, it can additionally restrict access to non-source development artifacts, data, and business processes. A developer’s environment is configured to match his/her access level. But, if developers share network access with other users and resources and are not restricted to least privilege, an opportunity for insider breach is still possible.

2.2.4 Implementing Access Control

Applying an access control policy to a store of artifacts can be accomplished in a variety of ways. The levels at which these techniques operate range from a low operating system level to a high application-level. All of these techniques, with the exception of branch-level access control, facilitate the physical modularization of artifacts to accomplish the goals of an access control policy. The finest granularity of the techniques (other than branch-level access control) is limited to whole artifacts, or files. The granularity of branch-level access control depends on the version control system and the type of artifact involved, but is typically finer than the whole artifact. Not all techniques for implementation are well suited for all access control models. For example, most file
system permissions are based on the Discretionary Access Control model, but may have significant difficulty implementing Attribute-Based Group Access Control models where some attributes may be too high-level for file systems to express correctly.

2.2.5 Access Control and Capability Lists

Access control rights are typically expressed in object-level lists or subject-level lists. Object-level lists have a single entry for each controlled object followed by a set of subjects and whether access is allowed or denied for a particular action. Object-level lists are commonly referred to as Access Control Lists, or ACLs. ACLs are efficient for determining if a subject may access a known object in a particular way. They are not efficient for determining the total set of access rights for a given subject. Capability Lists (CLs) are subject-level lists that state all access rights a subject has on a list of objects. CLs are more efficient for determining a subject’s total set of access rights but are less efficient for determining which subjects have access to a given object. ACLs tend to be more commonly used for access control right specification than CLs.

2.2.6 File Systems

The most common artifact store is a computer’s file system, i.e., the subsystem in an operating system responsible for organizing and managing files. A file system can provide local and/or network access to its files (and directories) and commonly controls access using ACLs for the basic actions of read, write, and execute. File systems can be used to implement access control policies that physically modularize a software project’s artifacts. The finest granularity for access control for a file system is an entire file.

2.2.7 Higher-Level Artifact Systems

Database servers and web servers are software programs that manage networked access to one or more data stores (e.g., table rows in a database, a set of web pages). Access to artifacts may only be achieved through the particular server software and access rights are usually controlled
using ACLs. The finest granularity for access control for a web server is an entire file. Database servers can have finer granularity than an entire file (e.g., rows of text from a source code artifact stored as rows in a database table) but they are not typically used as an artifact store for software development (although our future work explores the use of databases to replace file system-based artifact stores).

2.2.8 Version Control Systems

Version Control Systems (also referred to as Revision Control Systems), or VCSs, are a primary component of Source Control Management systems (SCMs). VCSs store and manage changes to software artifacts for a software development project. They are commonly used in team-based software development as VCSs can attribute single changes to an individual team member and help resolve conflicts when changes disagree. VCSs also serve as a backup for the software artifacts and a repository through which the development team may access any/all of the stored software versions.

Many of today’s popular VCSs and SCMs (e.g., SVN, Microsoft’s Team Foundation Server, Borland’s StarTeam) include some form of access control, so that project administrators may designate which software artifacts a team member may access and for which operations [17] [53] [70]. VCSs use ACLs for controlling artifact access and typically implement Discretionary Access Control models or simple Role-Based Access Control models.

2.2.9 Version Control System Branches

Branches are a logical modularization strategy intended to isolate a set of changes specific to the development of a feature set or version. Thus, branches have their own inherent modularization strategy. Branches allow fine-grained specification of artifact fragments (i.e., finer than an entire artifact) and can be used to control access to the changes within a branch (e.g., SVN and the Gerrit variant of the Git version control system [26]). However, developers work only within a single branch at a time, whether it be an access-restricted group of changes or a new feature/version (the
intended use of branches). As with physical modularization strategies, branches that need to share changes must have the changes manually pushed/pulled between them. Sensitive source code that is leaked into a less restricted artifact must have the source code removed and all indications of the leak manually removed from the VCS change history. Additionally, a restricted branch cannot be merged back to a less restricted branch/trunk as this would violate the access control policy.

### 2.2.10 Mocks and Stubs

To prevent reverse engineering of target code, access must be restricted to sensitive source code artifacts and the target code that is produced. However, developers must be allowed to build and test their work with as little difficulty as possible. One technique is to deploy external libraries that are stripped of all sensitive source. They can be distributed in source format and/or as a pre-compiled library. While this technique satisfies both access control of sensitive source and target code and allows developers to run and test their software, drawbacks include artifact redundancy, code-sharing obstacles, and the possibility of obsolete distributions. Another technique that can satisfy both is to insert a non-sensitive representation of the restricted source code at the time the target code is built. The resulting target code would therefore contain a non-sensitive representation of the sensitive target code. If an insider were to perfectly reverse engineer the target code, the result would not contain any sensitive source code artifacts.

A form of this second technique for protection of target code against reverse engineering attempts involves the use of mocks and stubs. Mocks and stubs are incomplete representations of source code constructs that are created primarily for testing purposes. They allow testing to focus on a part of the software without requiring complete implementation, making it easier to detect certain kinds of bugs. Fowler discusses the subtleties between mocks and stubs, stating that mocks use behavior verification and stubs can use either behavior or state verification for test result determination [24]. He also differentiates between “mockist” and classic approaches to test-driven development.

In addition to testing, mocks and stubs can be used to dynamically (i.e., at build time) insert
incomplete or non-sensitive representations of source code just before it is built based on the executing user’s privileges. This would both allow the compiled product to run successfully for testing and thwart reverse engineering attempts. Additionally, the dynamic insertion of mocks and stubs for sensitive code replacement could increase the likelihood that access-restricted interfaces will stay up-to-date.

2.3 Crosscutting Concerns

Dijkstra introduces the concept of a concern, stating that by focusing on a single concern at a time, one can produce more efficient and consistent work [22]. While Dijkstra leaves the definition of concern open, we view a concern as a logical and dynamic modularization of artifacts, able to overlay any other physical modularization strategy. In practice concerns can represent many different concepts that occur during software development, such as use cases, design properties, or tasks.

2.3.1 Modularization and Crosscutting

Parnas describes software design modularization techniques and stresses the importance of making good module creation decisions to avoid unnecessary maintenance when software design changes [59]. A module represents a subset, or type, of software functionality (e.g., billing reports), or a design pattern (e.g., the model/view/controller design pattern). However, physical modules that are completely self-sufficient (i.e., containing all code necessary to support the module’s purpose) can increase code maintenance as some of the module’s code may have been duplicated from other modules, particularly as the module’s size increases [6]. Modules that do not duplicate code, but instead reuse and share code with other modules, are referred to as “crosscutting”. In essence, the code necessary to perform the module’s purpose no longer resides within a single module.

A large body of research has focused on crosscutting modules and features, showing that crosscutting is a common occurrence. VanHilst and Notkin state that crosscutting increases as a product’s components are increasingly reused, software functionality changes, and the classes
evolve [80]. Lopez-Herrejon and Apel show through four case studies that crosscutting is typical in software development although each crosscut usually involves only a small number of classes and a small amount of code per class (6% on average) [48]. Feature-oriented programming and collaboration-based design acknowledge this fact, representing a feature as a collaboration of roles (or collection or code fragments) [3, 5]. Smaragdakis and Batory define refinements as a unit of functionality that involves parts of multiple classes spread over a codebase, and introduce a technique for expressing refinements called mixin layers [74]. These approaches are analogous to crosscutting concerns, i.e., multiple fragments of source code files that encapsulate some function or feature. The research argues that software incorporating these techniques improves in scalability, code reuse, and evolution [74, 80].

2.3.2 Access Control as a Crosscutting Concern

Prior to our research, access control has been studied as a crosscutting concern at the software application level or to manage runtime resource utilization and execution of classes, but not at the source code level. Ramachandran, et al. implemented access control in software using aspects, only handling access control in the application domain (i.e., restricting access of the users of the software to certain application functions or data) [62]. Toledo, et al. use aspects to provide a logical modularization for the Java Access Control architecture, demonstrating reductions in maintenance and enhanced policy expressiveness [79].

The goal of software artifact (i.e., source code) access control is to prevent unauthorized entities from performing certain, specified operations on restricted code. The restricted code can be as small as a single line of code (e.g., a database connection statement with a confidential login and password) or as large as an entire module (e.g., a directory and all contained directories and files). Developers typically restrict access to confidential credentials stored in the source code, proprietary or regulated algorithms, and features that are withheld to entice users of the source code to pay an additional premium for access. Tasktop is a closed source application lifecycle management product that offers a popular, open source Freemium variant known as Mylyn. Mylyn
users who pay for Tasktop gain access to the withheld features of instant desktop file search, enhanced time tracking, and export capabilities [77]. Mik Kersten, one of the founders of Tasktop, states that, while Tasktop’s closed source (i.e., restricted) features are modularized, there is still some duplication due to crosscutting between modules and core classes and such duplication is undesirable and increases maintenance [36].

2.3.3 Aspect-oriented Programming

Kiczales, et al. introduce aspect-oriented programming (AOP), a technique that allows programs to be written using the original functional decomposition and rewrites the program, via an aspect language and weaving tool, so that it better suits the crosscutting properties [38]. Thus, programmers are able to represent the intended design properties in their programs and retain the simplicity and integrity of the original functional decomposition. Examples of common crosscutting properties in software design are error checking and handling, synchronization, performance optimizations, monitoring and logging, and debugging support [1]. We use an access control policy instead of an aspect language to “rewrite” the originally decomposed source code into a view of the codebase that satisfies a developer’s access rights.

2.4 Related Work

A fair amount of research has been conducted in the area of access control models and tools. Joshi, et al. discuss the significance of insider threat to software and the importance of access control as a protection mechanism [32]. They survey seven access control models, comparing them with respect to web-based applications and workflow management systems. They conclude that RBAC is the best candidate for suiting most enterprise access control web application requirements. However, they stress that RBAC will need to be extended in order to fully satisfy security policies. Pramanik, et al. feel that a least privilege-based access control mechanism alone will not prevent insider attacks [60]. They implement a document access control model that enhances least privilege, considering the context in which the access request is made (e.g., currently opened
documents and programs, prior requests). Their access control system also monitors information flow between documents and can react if an information flow satisfies a specified insider threat rule. Their technique is similar to profile-based intrusion detection systems and seems to offer additional strength to least privilege access control policies. However, these improvements also bring additional complexity to policy management specification, enforcement, and maintenance. Our proposed approach could instead allow the flow of more restricted information into a less restricted artifact, but would place the information in a more restricted fragment within the artifact. Thus, lower access requests would only receive the information in the artifact that satisfies their access privilege (i.e., any information in the more restricted fragment would not be visible).

Blackwell introduces a three-layer security architecture to protect against insider threat along the physical, logical, and social fronts [9]. Data and software access controls are a necessary component of his logical layer, as boundary protections are powerless against insiders. Our proposed platform is designed to fit within such a layered system’s security policy.

Kruchten introduces the “4+1” software architectural model using four distinct views of a software’s architecture, each one presenting crosscutting architectural concerns directed at a different type of stakeholder [43]. The four views are logical (object relationships), process (concurrency and synchronicity), physical (hardware mappings), and development (static organization). Scenarios comprise the “+1” part of the model’s name. The scenarios clarify each relevant view and stimulate evolution of the overall architecture. While the concerns described in Kruchten’s paper are higher-level than those we normally consider, the “4+1” model shows the usefulness of distinct concern-driven views of a shared source of artifacts (i.e., architectural elements, forms, rationales, and constraints).

Recent advances in visual interface design suggest that developer efficiency increases when the developer’s programming environment is built around concern-oriented development. Bragdon, et al. introduce Code Bubbles, an experimental development interface that allows the user to focus on a collection of bubbles, or fragments, that pertain to the user’s current task [10]. DeLine and Rowan describe the CodeCanvas interface which allows a user to maintain an overall view of a
development task as a canvas [20]. The developer can “tear off” artifact parts that better pertain to the task.

Computer supported cooperative work incorporates separation of concerns, as well as the necessary work of reintegrating partitioned work back into the whole. Dekel and Herbsleb address the usefulness of providing community-generated knowledge about software artifacts to interested software engineers [19]. They implement their technique specifically for pushing API directive links into the calling source code. Their technique is useful for less experienced developers as a means of quickly adjusting to an unfamiliar codebase. Our technique can potentially include fragments from API documents, developer wikis, ad hoc support links, diagrams, etc. All of this material can be associated with a concern.

Lopez and van der Hoek survey and categorize many intensional and extensional concern-oriented approaches [47]. They suggest that future techniques incorporate more than one of the categories of approaches and support artifact types other than source code. Our access control concern model is an extensional technique, but our platform can provide a foundation for many of the other approaches they review. An Eclipse navigation-based concern management tool, known as Mylyn (previously called Mylar) extends the Eclipse views and editors to visualize the concern and supports several existing issue tracking databases, allowing support for multiple concurrent concerns [37]. Robillard and Murphy implement FEAT, a tool that uses structural dependencies in code and queries for locating and gathering concern-related fragments [64]. Tarr, et al. identify the maintenance difficulties of multiple, overlapping concerns and describe the use of hyperslices to promote separation of concerns and ease of maintenance [76]. Harrison, et al. describe the Concern Manipulation Environment (CME) as a tool that can construct and maintain arbitrary concerns via forward-engineering enumeration (extensional) and reverse-engineering queries (intensional) [28]. They promote artifact neutrality and representation of concern relationships and constraints. Majid and Robillard describe NaCIN, an Eclipse tool that monitors and associates a developer’s navigation/event activity and code structural dependencies with high-level concerns (i.e., developer-assigned tasks) [52]. They use Java methods as the basis of a fragment and store
the concern model as XML. Robillard and Weigand-Warr present ConcernMapper, an Eclipse tool that assembles concerns based on an intensional technique using Eclipse code searches [65]. They support Java methods and member variables as the fragment elements. Nistor and van der Hoek introduce a tool for dynamic concern-driven programming [55]. Intensionally specified artifact fragments for development tasks (i.e., concerns) are dynamically updated as developers work with them using semi-automated heuristics and manual specification/correction. Our platform stores both artifacts and the concern model in the same database but can potentially support these techniques (both intensional and extensional) for concern management.

Other techniques utilize concerns through mining code, dependency graphs, or change set information from a source code version control system. Ratanotayanon, et al. describe a method for updating concern fragments using the diff utility: a conflict resolution tool native to version control systems [63]. Adams, et al. implement and evaluate a history-based concern mining approach called COMMIT [2]. Their results indicate the usefulness in mining program elements at fine levels of granularity and the granularity depends on the type artifact being mined (C versus Java versus design documentation, etc.). Kozaczynski, et al. describe an approach to automate transformations in source code using syntactic, semantic, and abstract concept-based knowledge via recognition rules and concept pattern descriptions [41]. Tip examines a multitude of program slicing techniques, supporting a variety of language features [78]. Program slicing filters a codebase to the program elements that are directly and indirectly responsible for a specified point (or line) and variable pair in the target program, using dependency traversal. Codebase filtering can be considered a limited type of concern mining where the slicing criterion is the concern and the filtered code represents the related fragments. Program slicing can also supplement other concerns by facilitating capabilities like determining the reach of dependent code modifications in fragments. These techniques can be adapted to use our platform, exploiting only its access control strategy or also utilizing our concern model as their own.

Databases have been used to store and analyze software source code. Linton describes a set of tools to import source code into a relational database and facilitates visual exploration of the code
database [45]. Kotik and Markosian propose a method that transforms source code between the file system and an object-oriented database and allows software analysis, querying, and test case generation [39]. We differ from these techniques in that we intend to use a relational database as an artifact store during the development process, as well as analysis.

To address the overhead of maintaining multiple program alternatives, Nita and Notkin propose a method for developers to specify code-level mappings between code alternatives instead of requiring redundant updating [56]. Shonle, et al. demonstrate the use of Arcum to automatically transform a codebase containing problematic and inconsistent crosscutting design idioms [73]. These techniques, similar to other crosscutting-oriented techniques, improve code reuse and consistency but do not possess the means to restrict access to crosscuts.
Chapter 3: APPROACH

As we have discussed, the accepted method for controlling access to software development artifacts involves physically modularizing the source code into separate modules, most likely on top of some other physical modularization strategy (e.g., files grouped into directories by function). Developers naturally modularize a project’s software artifacts, grouping them into categories based on an envisioned design-level commonality (e.g., functional similarity). In short, the essential modularization of a software project’s artifacts fits fundamental design properties of the software. But, physically modularizing a codebase for access control overlays a second modularization strategy (the access control modularization) on top of the first (the design modularization), decomposing the original modularization strategy into smaller micro-modules. Coupled with an artifact granularity of no finer than an entire file, this creates redundant artifacts spread across multiple access-restricted modules. This in turn increases the amount of effort required to maintain the codebase in order to preserve consistency between the redundant artifacts. It also decreases the flexibility of the access control policy in general (adding a new access group requires further decomposition of the modular structure).

Figure 3.1: Restricting access to the FooSoft codebase using a conventional technique.
3.1 An Example: FooSoft

To better illustrate this problem, let us return to the example scenario from the introduction. FooSoft is a fictitious software development company that has developed a simple Java integrated development environment, available for download at a modest fee. FooSoft decides to open source a Freemium variant of their IDE as a free download. They have two goals with this marketing decision:

1. To enhance and debug their IDE through public collaboration
2. To increase sales of their premium IDE variant by removing certain features from the free version in order to entice them to upgrade.

FooSoft decides to remove the “Find Text” feature from the free variant of their IDE. Their IDE codebase contains 9 source directories and 78 source code files. The “Find Text” feature is spread across 5 files in 4 directories. Out of those 5 files, only 1 of them is completely devoted to the “Find Text” feature. Thus, FooSoft’s access control policy states that all developers may access the Public code, but only internal, or Private, developers may access code pertaining to the “Find Text” feature. Using a conventional file-based access control technique, the one file that only contains the feature code may be placed in a restricted directory. The remaining 4 files must be duplicated, stripped of the restricted code, and placed in a publicly visible directory. Figure 3.1 shows the modular decomposition of the FooSoft IDE after this duplication.

If any unrestricted code in a duplicated artifact is modified, the changes must also be duplicated in all other access-restricted artifacts, increasing the amount of work necessary to maintain artifact consistency. Furthermore, if an access control policy changes (e.g., a new access entity is created or a new feature is restricted), further decomposition may occur, requiring additional duplications and work to maintain consistency. Figure 3.2 shows FooSoft’s access control policy change when they decide that no employees working remotely may have access to a proprietary algorithm in one of the project’s source code files. This scenario is similar to the non-export regulations of cryptographic algorithms when code development extends outside of the United States.
The reason the conventional file-level access control technique requires artifact duplication is
because FooSoft’s access control policy crosscuts the IDE codebase, i.e., it does not completely
overlay a single module of the software’s original modularization strategy. In situations where an
access control policy fits perfectly into an existing codebase’s modularization strategy, crosscutting
may not occur, and thusly neither may artifact duplication. These situations typically occur where
access is restricted to a separate plug-in or add-on, and the module is small enough so as not to use
any code outside of the module.

3.2 Using Crosscutting Concerns for Access Control

While crosscutting properties may represent important design goals, they can “tangle” the imple-
mentation of software, further decomposing source code modularization and increasing the dif-
ficulty of development and maintenance. This is the case in our FooSoft example. However, if
we acknowledge the crosscutting nature of FooSoft’s access control policy and use a crosscutting
concern-based technique for access control with fragments smaller than whole files, we can avoid
modular decomposition and artifact duplication. Figure 3.3 shows FooSoft’s access control policy.
expressed using a crosscutting concern-based approach. Note that the color representing Public access is the same color of the code that all developers may access. If the Public access group had any code that was exclusive to its access group, then there would be a separate fragment color for the Public access group, in addition to the color for code that all users can access.

### 3.3 Design Goals

To reduce the maintenance attributable solely to crosscutting access control policies, we propose the use of a concern model to apply access control policies to software artifacts. While there is substantial previous work into concern-based annotation of source code, none of the work addresses the use of concerns for access control within the source code itself. As access control is within the realm of computer security, our concern model must address the following security principles with respect to software artifacts:

1. Confidentiality

2. Integrity/Reliability

3. Non-repudiation/Accountability
If our concern-based technique cannot satisfy these security goals, then our approach is unsatisfactory for access control. Please note that while availability is a primary security principle, we feel that availability can be addressed on a lower level than our concern model, e.g., server clusters, cloud deployment. In addition to supporting the security principles, we want our concern model to be useful to developers and not get in their way. We also want our approach for access control to lead to new opportunities for sharing code and expressing interesting and complex access control policies. Thus, we want our concern model to address the following additional design goals:

1. Transparency
2. Usability
3. Flexibility

3.3.1 Confidentiality

An access control system must preserve confidentiality, i.e., the prevention of unauthorized disclosure of protected information [75]. Our concern model will allow users to access all source code fragments that they are allowed to access and will prevent them from accessing any source code fragment for which they are not allowed. Any access control technique that does not provide confidentiality of source code fragments cannot be used for access control. Additionally, we do not want to provide unauthorized developers with any cues about source code to which they do not have access. Such cues may cause unnecessary attention and provoke attacks. In short, if developers do not need access to it, then they do not need to know of its existence. Lastly, our approach does not monitor the semantics of the restricted material, meaning that we do not track source code that is copied by an authorized developer from a fragment and pasted elsewhere (i.e., declassified). This scenario involves security leak detection and we consider that as future work.
3.3.2 Integrity/Reliability

Access control systems must be reliable, preserving the integrity of the access control policy and the objects it protects. This means that our concern model must ensure that access-restricted fragments cannot be modified without appropriate permissions, intentionally or accidentally. We also regard integrity as ensuring that fragment boundaries remain intact as fragment contents grow and shrink, as well as the content surrounding the fragments. Our approach should also serve as the sole point of entry for the artifact store, to prevent developers from circumventing access control.

3.3.3 Non-repudiation/Accountability

Non-repudiation prevents a developer from denying his/her participation in a security incident. Our concern model must provide for tracking of developer actions to assist in testing and forensics should a security breach occur. Typical approaches to accountability are the use of logging and audit trails.

3.3.4 Transparency

Almost as important as satisfying the security goals is that the concern model must be transparent to the developer. Interference with developer workflow can cost a company time and money, as well as increase user frustration and dissatisfaction with our technique. Additionally, we cannot require developers to manually maintain or inspect fragment boundaries for consistency. Our concern model must fit invisibly into existing developer workflow and automatically and reliably maintain fragment consistency without manual intervention.

3.3.5 Usability

Usability of our technique is an important design goal. By “usable”, we mean that our approach is helpful to developers in implementing and maintaining access control of software artifacts. Our definition of usability draws from the ISO standard definition of usability (i.e., facilitate task com-
pletion, efficiency in task completion, and user satisfaction) [23]. Our concern model should also be easy to use and intuitive. Lack of usability could reduce or eliminate any improvements our approach might otherwise bring to developers. Simplicity of fragment specification and visualization are important components to usability, as well as not introducing any impediments in the software development process.

3.3.6 Flexibility

Some important limitations of conventional techniques for implementing access control are that the model for applying access control is static and built into the artifact storage system. For example, if developers use a standard network server file system to implement access control of software artifacts, then they are most likely limited to using the file system’s Discretionary Access Control model for policy implementation. While this satisfies some policies, there may be additional policy properties that are impossible to implement using DAC, e.g., restricting artifact operations based on a developer’s location. Our concern model is abstracted away from the software artifact storage system so not only are we free to utilize virtually any access control model of our choosing, but we also have many more characteristics by which access entities may be described. Thus, our concern model allows for more flexible expression of access control policies than traditional techniques.

3.4 Overview of our Concern Model

Our approach to software artifact access control is to model access control as a crosscutting concern in the codebase. Our mechanism for access control, or the access control monitor, sits between developers and their source code repositories (see Figure 3.4). When a developer requests access to the artifact store, the access control concern is applied with the requestor’s credentials. This results in a permitted set of artifact fragments that are accessible by the requestor for the specified operation. Because we treat access control as a crosscutting concern, an access control policy can be applied as a logical modularization strategy on top of any other physical modularization strategy used by the software developers. Our approach eliminates the need for redundant artifacts
Figure 3.4: Concern Model Overview.

To support access control, it also simplifies the maintenance and application of an access control policy as it is a purely logical modularization technique.

To accomplish our design goals for strict access control of software artifact fragments, our concern model must possess the following characteristics:

1. A definition of fragment granularity
2. Annotation of access-restricted fragments in software artifacts
3. Automatic and reliable maintenance of fragment boundaries via a set of fragment update heuristics
4. A single point of communication between developers and software artifacts that enforces configured access control policies
5. Flexible access control policy implementation
6. An audit history of all developer fragment access

3.4.1 Fragment Granularity

The types of software artifacts that comprise a software engineering effort are numerous, including requirements notes, specifications documents, design documents, test cases, and source code. Our research only addresses access control of text-based source code artifacts. While text-based concern fragments can be as fine-grained as a single character, we limit the granularity of our technique to a single line of code (i.e., a string ending with a new line character or end-of-file). This provides us with a much finer granularity than conventional techniques, simplifies implementation, and allows us a reasonable level of detail for access control assignment. This characteristic address the design goals of usability and flexibility.

3.4.2 Fragment Annotation

For developers who possess read access to restricted fragments, we feel it is important to provide simple, visual cues representing fragment placement within source code. Similarly, if a developer does not have read access to a fragment, we want no cues that restricted code is missing from his/her view of the source code. If a developer has the ability to create fragments (e.g., a project lead), access-restricted fragment markup should occur directly in the source code editor to increase ease-of-use. Lastly, a composition of fragments should not hamper successful building of target code or program execution at a designated level of access. Therefore, our approach requires that an authorized access level be selected for execution of the program (e.g., build the program at an “Employee” level of access). Our approach to fragment annotation within software artifacts addresses our design goals of confidentiality, usability, and transparency.

3.4.3 Fragment Maintenance Heuristics

Our heuristics for automatic fragment boundary maintenance for a single change operation depend upon where the modification occurs in the artifact relative to a particular fragment and whether
characters are added or removed. Each operation may only be an add operation or a remove operation. Moving code from one location in an artifact to another location is two operations: first a remove then an add. We identify nine heuristics to provide automatic fragment maintenance:

1. Add in front: characters are added completely before the fragment. The fragment body remains the same but the fragment starting position moves further down in the artifact based on the number of characters added.

2. Delete in front: characters are removed completely before the fragment. The fragment body remains the same but the fragment starting position moves further up in the artifact based on the number of characters removed.

3. Add behind: characters are added completely after the fragment. The fragment body and position in the artifact remain the same.

4. Delete behind: characters are removed completely after the fragment. The fragment body and position in the artifact remain the same.

5. Add inside: characters are added completely within the fragment. The fragment body increases and the ending position moves further down in the artifact based on the number of characters added.

6. Delete inside: characters are removed completely within the fragment. The fragment body decreases and the ending position moves further up in the artifact based on the number of characters removed.

7. Delete straddle start: characters are removed on both sides of the fragment’s starting boundary. The fragment body decreases based on the number of characters removed within the fragment. The fragment starting position moves further up based on the number of characters removed outside of the fragment. The fragment ending position moves further up based on the total number of characters removed.
8. Delete straddle end: characters are removed on both sides of the fragment’s ending boundary. The fragment body decreases and the ending position moves further up in the artifact based on the number of characters removed within the fragment. The fragment starting position is unaffected.

9. Delete straddle fragment: characters are removed before the fragment’s starting boundary and beyond the fragment’s ending boundary. The fragment body is empty, the fragment starting position moves further up in the artifact based on the number of characters removed before the starting position. The fragment’s ending position moves further up in the artifact based on the number of characters removed within the fragment and before the fragment starting position.

There are no heuristics for Add straddle start and Add straddle end as all characters are added at a single point of insertion, either inside or outside the fragment. Our automatic update heuristics address the design goals of confidentiality, integrity, transparency, and usability.

3.4.4 The Access Control Monitor

The access control monitor authenticates the access requestor and determines access privileges for the requested object(s) and operation by consulting a pre-loaded access control policy. The monitor then filters the data stream between requestor and artifact store based on the requestor’s access privileges for the data stream’s objects (i.e., the various artifacts and fragments that comprise the data stream). The access control monitor is the sole interface between the developers and the artifact store. This prevents developers from bypassing the access control monitor and requesting artifacts directly from the artifact store. However, developers do not need to be aware of the access control monitor, believing they are communicating directly with the artifact store. All communication between the developers, the access control monitor, its policies, and the artifact stores must be secure in order to prevent eavesdropping.

When a developer requests an artifact through the access control monitor, our concern model
consults the access control policy for that artifact using the requestor’s credentials. The monitor requests the artifact from the artifact store, transforming the artifact by removing any read-restricted fragments from the artifact according to the permissions in the policy. The read-restricted fragments are stored in a transformation database and the filtered artifact is sent to the requestor. No trace of a read-restricted artifact remains in the filtered artifact. When a developer sends an artifact to the access control monitor, the monitor uses the requestor’s permissions to determine if it must extract any changes from write-restricted fragments in the artifact. If the monitor must extract changes, the changes are stored in the transformation database to restore the next time the developer requests the artifact from the monitor. This process provides transparency to the requesting developer. The monitor then restores any read-restricted fragments that were removed when the artifact was sent to the requestor and sends the resulting transformed artifact to the artifact store. This process provides transparency to the artifact store and other developers. The design for our access control monitor addresses our design goals of confidentiality, integrity, and transparency.

3.4.5 Flexible Access Control Policies

Our concern model treats access control policies abstractly, allowing the use of interchangeable policies to suit different organizational strategies and evolving security policies. The underlying access control model must allow for permissions to be assigned for entire codebases, directories, files, or single lines of code within a file (i.e., a fragment). The access control model must also allow for flexible and dynamic descriptions of entities, as well as entity groups and hierarchies. Such an approach is necessary if we wish to satisfy a broad range of company security policies. Access control policies should also be easily interchangeable without requiring any change in the codebase file structure (e.g., duplicating directories or files). Flexible access control policy expression addresses our design goals of usability and flexibility.

We use ABAC\(\alpha\), proposed by Xin, et al., as our underlying access control specification language. With ABAC\(\alpha\), we have defined three software artifact access control models for our research [31]. The models are:
Figure 3.5: Condensed FooSoft access control policy.

1. SAAC$_0$: hierarchical groups, single membership, and object exclusivity

2. SAAC$_1$: similar to SAAC$_0$ plus location

3. SAAC$_{cw}$: same as SAAC$_1$ plus Chinese Wall

SAAC$_1$ is our ABAC$_\alpha$ configuration of a modified FooSoft access control policy shown in Figure 3.5. We condensed the FooSoft access control hierarchy in Figure 3.2 by collapsing the Remote Employee access group into the Private access group. In this way, Employees who are logged in locally will still have access to all Private objects except those that are marked as mutually exclusive (i.e., accessible only to those Employees logged in remotely). SAAC$_0$ is described in our human subject experiment in the Evaluation section of this document. SAAC$_{cw}$ is described in Appendix 2.

In Table 3.1, UA is the set of user attributes with a single atomic group attribute, $ugroup$, and a location attribute representing the user’s physical location (e.g., IP address). SA, the subject attribute set, has only a single atomic group attribute $sgroup$. The object attribute set, OA, has an atomic group attribute for reading, $rgroup$, for writing, $wgroup$, and a boolean flag for exclusivity. Our model defines five access group values (G): No access, Public, Private, Local Employee, and Admin. Each group attribute for users, subjects, and objects may take one, and only one value in
Table 3.1: Software artifact access control SAAC₁ configuration using ABACα.

<table>
<thead>
<tr>
<th>Basic sets and functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>UA = <code>{ugroup, location}</code>, SA = <code>{sgroup}</code>, OA = <code>{rgroup, wgroup, exclusive}</code></td>
</tr>
<tr>
<td>P = <code>{read, write}</code></td>
</tr>
<tr>
<td>Range(ugroup) = Range(sgroup) = Range(rgroup) = Range(wgroup) = G</td>
</tr>
<tr>
<td>Range(location) = I</td>
</tr>
<tr>
<td>G = <code>{No access, Public, Private, Local Employee, Admin}</code></td>
</tr>
<tr>
<td>Partial Order = No access &lt; Public, No access &lt; Private, Private &lt; Local Employee, Local Employee &lt; Admin, Public &lt; Admin`</td>
</tr>
<tr>
<td>I = <code>{all IP addresses}</code></td>
</tr>
<tr>
<td>I' ⊂ I</td>
</tr>
<tr>
<td>attType(ugroup) = attType(location) = attType(sgroup) = attType(rgroup)</td>
</tr>
<tr>
<td>= attType(wgroup) = attType(exclusive) = atomic</td>
</tr>
<tr>
<td>Thus, ugroup: U → G, location: U → I, sgroup: S → G, rgroup: O → G</td>
</tr>
<tr>
<td>, wgroup: O → G, exclusive: O → true/false</td>
</tr>
</tbody>
</table>

G. Our model supports two access modes (P), read and write. I is the set of all IP addresses (or simply an IP address validator) and I’ is the set of FooSoft’s local IP addresses.

SAAC₁ supports access inheritance, so that a subject has access to all objects with group attributes less than or equal to the subject’s UNLESS the object’s exclusive flag is true. The partial order of G shows the access hierarchy among the groups. The exclusive flag allows mutually exclusive access to objects.

The authorization policies in SAAC₁ show that subjects may access objects if the subject access group is greater than or equal to the object’s and the object is not exclusive. If the object is
exclusive, only a subject with the same access group value as the object may access it in the
intended manner. Subject constraints are shown in ConstrSub where users may create subjects for
any access group equal to or less than their access group value (other than the No Access group),
but only Employees who have a local IP address may create a Local Employee subject. Employees
logged in remotely may only create Private subjects. The object constraints show that subjects
may create, or modify the attributes of, objects with an access group value less than or equal to the
subject’s group. SAAC_1’s object constraints imply that subjects can declassify information at will,
increasing the threat of security leaks. We address this in the Conclusion as future work.

3.4.6 Audit Log

For the purposes of forensics and testing, the access control monitor maintains an audit log of
all client access attempts to artifacts, fragments, and the implemented access control policy. This
characteristic addresses our design goal of non-repudiation.

3.5 Enforcing Access Control

In order to better illustrate how our concern model approach fits within a software developer’s
workflow, let us return to the FooSoft example. Figure 3.6 shows a simplification of how our
design for a concern model enforces an access control policy between a developer and an artifact
store. In this figure, the developer has read, but not write access to Remote Developer fragments
(the pink fragments) and no access to Employee fragments (the turquoise fragments).

In Figure 3.6A, the developer requests artifact $a^s$ from the access control monitor. In Figure
3.6B, the access control monitor transforms $a^s$ into $a^e$ based on the requesting entity $e$ and the ac-
cess control policy $p$ loaded into the monitor. This transformation removes any fragments that the
requesting entity cannot see (i.e., the read operation). The transformed $a^e$ is sent to the requesting
entity in Figure 3.6C. Because our approach is transparent, the requesting entity may be completely
unaware of the presence of the access control monitor and the removed fragment, believing he/she
is communicating directly with the artifact store.
A reverse transformation is applied to artifacts that are transmitted from a developer to the access control monitor. Figure 3.6D shows a modified artifact $a'^e$ being sent from the developer to the access control monitor. In this example, the developer has modified a fragment for which he/she does not have write access. Our approach does not try to prevent a developer from modifying an artifact within the developer’s work environment. We believe that such enforcement would affect usability, decrease transparency, and create additional vulnerabilities. In Figure 3.6E, the access control monitor transforms the artifact by removing any changes to write-restricted fragments and re-inserting previously removed read-restricted fragments. Lastly, the transformed $a'^e$ is sent to the artifact store. If the developer again requests $a'^s$, the access control monitor re-inserts the disallowed modification (the salmon-colored section) previously made by the developer to aid transparency and usability by “remembering” the developer’s last changes saved by the monitor.

### 3.5.1 Protecting Target Code

Lastly, it is important to elaborate on how our approach protects access to sensitive target code, i.e., pre-compiled libraries of sensitive source code. Reverse engineering of target code poses an insider threat to software during the development process, allowing a malicious developer to steal sensitive
data or proprietary algorithms. One way to prevent the reverse engineering of target code is to ensure that developers only have access to the artifact fragments they need. However, developers with restricted access must still have enough code to build and test their work. Separate testing APIs can solve this problem, but create additional maintenance overhead, can change without notice, have limited granularity, and do not decompose easily should access privileges change. Our concern model provides protection of target code through the use of mocks and stubs, providing executable source placeholders for restricted target code and removing the need to distribute pre-compiled libraries that may be vulnerable to reverse engineering attacks.
Chapter 4: IMPLEMENTATION

When access control policies crosscut software artifacts, the coarse granularity of conventional techniques forces developers to protect sensitive code fragments by restricting access at the artifact level. These techniques are implemented through duplication of artifacts, typically one per access group or subject, and modifying the artifacts to suit each group’s access level. While this approach may satisfy the access control requirements of the security policy, the techniques create duplicates of non-restricted code that require additional maintenance for consistency among the duplicates when the non-restricted parts change. Figure 4.1(A) illustrates a scenario where a single codebase has private and public distributions, sharing all software artifacts between the two distributions except for separate implementations of Artifact 1. Current techniques manage distribution selection for projects that have more than one level of access using build path switches, factory arguments, conditions embedded in the source code, or pre-compiler instructions. All of these selection mechanisms are partial implementations of the access control policy, leaving the remainder of the access control policy to be implemented using storage mechanisms like file permissions or server access control lists.

Our access control technique presents a client request’s credentials to a configured access control policy associated with the artifact store attempting to be accessed by the developer. This results in a set of artifact fragments to which the requestor has the specified type of access. Because we treat access control as a crosscutting concern, it can be applied as a logical modularization strategy on top of any other physical modularization strategy used by the software developers. This eliminates the need for duplicated artifacts solely to support access control. Figure 4.1(B) illustrates the use of crosscutting concerns for access control. Access is only controlled for the restricted fragments of Artifact 1. A public developer may access all shared code and the public fragment of Artifact 1, while a private developer may access all shared code and the private fragment. Furthermore, our approach implements the access control policy in a single location, managing distribution selection and storage control solely in the access control monitor. Note that our technique is
Figure 4.1: Conventional and crosscutting access control techniques.

designed to control access only to source code artifacts, not to pre-compiled target code.

We have divided the implementation of our approach for concern-based access control of software artifacts into two parts: a developer interface prototype where users interact with fragments through visualization and fragment specification, and an access control monitor that filters transmissions between developers and artifact stores based on the requesting entities and the configured access control policies. Our developer interface, SaJE, is a simple Java integrated development environment, including a Java source code editor and compiler interface. Fragments are visualized for developers using color-coded highlights in the source code editor. Creation and deletion of fragments are accomplished via menu functions.

The access control monitor, GitBAC, is the sole interface between developers and Git repositories, using a secure communication protocol. The monitor receives standard Git commands from developers, determines permissions from configured access control policies, communicates with the Git repositories, and transforms artifact streams to and from the developers. The transformation mapping and removed fragments are stored in a database local to the access control monitor.
4.1 Front-end: SaJE

SaJE is a simplified Java Integrated Development Environment (IDE) prototype that provides the basic functionality for a developer to code, build, and execute a Java-based software project (see Figure 4.2). Most of SaJE’s functionality is available in modern IDEs like Eclipse and Visual Studio. However, SaJE also has special access control features built into it. We created SaJE instead of extending other IDEs because we wanted our test subjects to focus on access control-related tasks for our experiment and not be distracted by the interface and functional complexity of other IDEs. We also chose to build our own IDE because we could precisely tailor its fragment interaction capabilities to suit our vision of fragment annotation.

The left side of SaJE is the Project Explorer (4.2A), which allows a developer to interact with the projects, directories, and files within the SaJE workspace. Figure 4.2B shows the tabbed source code editor where a developer works directly with the Java source code and access control fragments. The debug console in Figure 4.2C is used for debugging and program input and output.
The console also possess a “Kill” button which will forcibly terminate the currently executing program in SaJE. Figure 4.2D is the SaJE menu which houses all of the basic functions for project, directory, and file creation, modification, and deletion. The SaJE menu also has a source code text searching function, program run and build functions, and access-restricted fragment creation and deletion functions. SaJE’s access restriction functions also work on directories and files selected in the Project Explorer.

4.1.1 Fragment Specification

While our concern model can support a variety of methods for concern specification and maintenance (e.g., XML or relational databases), we currently are using embedded, protected comments within source code artifacts to specify fragment boundaries. Listing 4.1 shows the syntax for fragment specification within source code. The use of the Java comment delimiters /* and */ are only appropriate for languages that respect them as comment delimiters, e.g., C, PHP. Different fragment boundary delimiters can be used based on the type of source code artifact that contains the fragment. <fragment label> is a unique identifier for the fragment and is automatically generated and stored by SaJE in a project database outside of the source code workspace. Since fragments are only relevant to the access control monitor with respect to the particular source code project, fragment identifier generation should be moved to the access control monitor and we consider this within our future work.

```
/*@START <fragment label>*/
fragment body
/*@END <fragment label>*/
```

**Listing 4.1**: Source code fragment syntax.

To enhance the usability of our technique, the SaJE source code editor does not show the actual fragment boundary comments. They instead are used to mark the beginning and end of the fragment highlighting. For example, listing 4.2 shows the code for two fragments specified in the artifact `AppMain.java`. Note that blank lines have been removed from the listing code for brevity.
Fragment \( f_1 \) has a public level of access in the access control policy and fragment \( f_2 \) has a private level of access. Figure 4.3 shows \texttt{AppMain.java} in the SaJE source code editor with fragment \( f_1 \) visualized in green and \( f_2 \) in orange. Following our design characteristics, SaJE’s finest unit of fragment resolution is a line of source code.

\begin{lstlisting}[language=Java]
package main;

public class AppMain {
  public AppMain() {
  }
  public static void main(String[] args) {
    @START f1*/
    System.out.println("This is a public statement!");
    @END f1*/
    System.out.println("This is a private statement!");
  }
}
\end{lstlisting}

\textbf{Figure 4.3:} Visualization of fragments in the SaJE source code editor.

\textbf{Listing 4.2:} Example fragment source code.

4.1.2 Assigning Access Permissions to a Fragment

In the SaJE source code editor, access permissions are assigned to a fragment at the time the fragment is created. The developer selects a range of text in the source code editor and then clicks
on “Access” in the menu (see Figure 4.4).

SaJE presents the developer with a modal dialogue box containing a list of access groups that have been configured in the project’s access control policy (see Figure 4.5). The developer selects the desired access group from the list and clicks “OK”.

Once the developer selects the desired level of access for the new fragment, SaJE creates the new fragment in the source code by:

1. Determining a unique identifier for the new fragment

2. Inserting fragment boundaries (along with the new identifier) before and after the selected source code

Our implementation of fragment visualization automatically hides the fragment boundaries from the developer in the source code editor and highlights the fragment contents with a color associ-
ated with the access group in the access control policy. We accomplish the hiding of fragment boundaries using an extension of the Java DefaultStyledDocument class. Figure 4.6 shows the new Public fragment created in this example.

By implementing fragment specification into the SaJE source code editor, we have met our design characteristics for fragment granularity and annotation. Developers may create fragments as fine as a single line of source code and assign access permissions for read and write operations to those fragments. Fragments may also be deleted within the SaJE source code editor using a similar menu command. Deletion of a fragment preserves the contents within. It is important to note that fragment deletion results in previously restricted content becoming unrestricted.

4.1.3 Access-aware Execution

SaJE has the capability to execute a Java project at a designated level of access, i.e., execute a program as a member of a selected access group. We incorporated a number of access-aware execution options based on the access control policy configured for our own internal testing and our human subject experiment (see Figure 4.7). These execution options are “Run as Public” and “Run as Employee”. As an implementation of SAAC$_1$, executing a program at a Public-level of access creates an access control subject that belongs to the Public group and passes that subject to the SaJE build method. The SaJE build method filters directories, artifacts, and fragments based
Figure 4.7: Access-aware execution menu options in SaJE.

on the provided subject. Executing at an Employee-level of access is similar to Public except that SaJE first tries to instantiate a Subject at a Local Employee-level of access. If the user’s IP address is not considered local, then SaJE tries to instantiate a subject at a Private-level of access.

Since fragments may be hidden from different access groups, the composition of all fragments in an artifact may not allow for traditional, access-ignorant execution of the program. For example, two fragments with return statements in the same method body of an artifact may cause an “unreachable code” error during a traditional build of the artifact (see Figure 4.8). Another type of build error common to access control techniques involves duplicate class declarations. Since conventional techniques involve duplicating and partitioning classes into separate access-controlled directories, if the compiler tries to build the composition of all classes, it will see more than one class declared with the same name and generate an error. Because any developer can potentially create fragments, we introduce the notion of access-aware building and execution of a program. The process we implement in SaJE for performing an access-aware execution of a program involves the following steps:

1. Compose a list all Java-based compilation units (CUs) within the project’s subdirectories, excluding the project’s bin directory
2. Remove any CUs that are in access-restricted directories

3. Remove any CUs that are access-restricted files

4. Remove all access-restricted fragments in the remaining CUs

5. Send the CU list to the Java compiler

During execution in SaJE, this process filters out any artifacts and fragments that the designated access group cannot see, i.e., to which the access group does not have read access. By exclusively restricting access to each of the return statements in Figure 4.8, SaJE’s access-aware execution of the code avoids an “unreachable code” error during the building of the target code (see Figure 4.9).

Figure 4.10 shows the result in the SaJE console of executing our example code in this section at an Employee level of access. The green fragment of source code in Figure 4.6 has been filtered out by our access-aware execution process. Current IDEs have workarounds for accomplishing results similar to our access-aware program execution, except that they only handle the exclusion of files and directories. They simply require the developer to restrict his/her build path to exclude any directories containing duplications, thus preventing a duplicate class declaration error. However, the build path must be manually modified each time the developer wishes to change the access...
Figure 4.9: Compilable source code with mutually exclusive access-restricted fragments.

```java
package stuff;
public class MyStuff {
    public MyStuff() {
    }
    public static String test() {
        return "an Employee's test string";
    }
    return "a Remote Employee's test string";
}
```

Figure 4.10: Output of the example program executing as an Employee.
4.2 Back-end: GitBAC

GitBAC, or Git-Based Access Control, is the back-end server part of our concern-based software artifact access control technique. GitBAC uses the Git version control system as an artifact store and acts as the sole interface between the Git client software that the developers use and the Git server software and artifact repository. Git supports three communication protocols: HTTP, SSH, and the Git protocol (for read only). GitBAC only uses the SSH protocol, which is satisfactory for maintaining confidentiality within an access controlled software development project. While GitBAC is a standalone proxy, all of our implementation and testing occurred within Eclipse using the Git command line tools as our Git client. The Git client authenticates only to GitBAC, oblivious of the location and credentials necessary to access the Git repository. Thus, GitBAC is the client’s only avenue of access to the source code artifacts. GitBAC is written in Java using the jGit and Apache MINA SSH libraries.

4.2.1 Overview of the GitBAC System

Figure 4.11 shows an overview of the seven primary components of GitBAC: the Git client interface, the access control monitor, the implemented access control policy, the artifact store, the artifact access interface, the transformation map, and the audit log.

The Git Client Interface

Git clients (e.g., eGit, Git command line tools) access Git repository artifacts transparently through GitBAC’s client interface using the SSH protocol (Figure 4.11A). GitBAC returns a data stream containing only the fragments of the artifacts that satisfy the subject’s access privileges. Access to fragment data occurs in different ways (e.g., cloning a project, pushing a commit). Directory and package listings show only the artifacts to which the requesting user has access. Similarly,
Figure 4.11: GitBAC overview.
editor access requests may only read and write content to an artifact fragment where the user has appropriate access.

The Access Control Monitor

The access control monitor is the primary workhorse of GitBAC (Figure 4.11B). The access control monitor receives a request from a developer and determines access privileges for the requested object (e.g., directory, file, or fragment) based on the developer’s credentials, desired operation (e.g., read or write to the object), and the configured access control policy. The monitor then sends the object to the transformation map for filtering based on the developer’s privileges and routes the resulting filtered data to either the developer or the artifact store.

Access Control Policies

GitBAC provides modular support for SAAC-based access control policies, allowing a variety of customizable policy implementations (Figure 4.11C).

Artifact Stores

The Git server’s artifact store (Figure 4.11D) is maintained outside of GitBAC. However all client access to the artifact store occurs through GitBAC. Direct access between GitBAC and the artifact store is protected using secure, authenticated channels.

Artifact Access Interface

GitBAC communicates with the artifact store (Git) directly using SSH (Figure 4.11E). Only GitBAC has full access to the repositories maintained by the Git server.

Transformation Map

Artifact-fragment transformations occur when read-restricted artifacts and fragments are requested by a developer (Figure 4.11F). When the developer sends updates through GitBAC to the Git
server, the transformation is reversed, applying our maintenance heuristics to restore any altered fragments, so that write-restricted artifacts and fragments are restored with their original contents. Removed fragments are stored in a transformation database so that they may be restored in future transmissions back to the sending entity. This restoration process enhances GitBAC’s transparency and usability.

Audit Log

For forensics and testing, the access control monitor maintains an audit log of all client access attempts to artifacts, fragments, and the implemented access control policies (Figure 4.11G).

4.2.2 Access Control

Our concern model treats access control policies abstractly, allowing the use of interchangeable policies to suit different organizational strategies. GitBAC uses the SAAC_1 access control model described in Table 3.1 in the approach section of this dissertation. We configured SAAC_1 using ABAC_α for its flexibility and openness to specify dynamic attributes, such as user location [31]. We chose SAAC_1 as an initial access control model for our concern-based approach as it allows us to assign per-fragment permissions for a set of simple, hierarchical groups. We can also implement a default permission scheme for all fragments that will simplify permission assignment to new fragments or newly imported fragments. Administrators can assign users to groups based on their organizational and functional purpose, applying the appropriate permission mapping immediately. Dynamic attributes (e.g., a user’s location) are assigned values at the time of authentication to determine the final access group to which the entity belongs (e.g., the entity’s IP address used for communication). Additionally, it is simple to change an individual user’s access control permissions as it merely involves a change in the user’s group.
Table 4.1: GitBAC access control group attributes.

<table>
<thead>
<tr>
<th>Group</th>
<th>Private Developer?</th>
<th>Working Local?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>No</td>
<td>Yes/No</td>
</tr>
<tr>
<td>Private</td>
<td>Yes</td>
<td>Yes/No</td>
</tr>
<tr>
<td>Employee</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Remote Employee</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Admin</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

GitBAC’s Access Control Policy

Git has no built-in access control mechanism. It relies on external tools or mechanisms inherent in the physical storage system (e.g., file system permissions) to implement an access control policy. Three tools provide repository or branch-level read/write access control: gitosis, gitolite, and gerrit. GitBAC differs from these tools as it applies an access control policy to software artifacts (the original artifacts and their change histories) within the Git repository, crosscutting all development branches without modularizing the codebase.

Table 4.1 shows a matrix of SAAC\textsubscript{1} group attributes representing the GitBAC access control groups used in the examples in this dissertation: Public, Private, Remote Employee, Employee, and Admin. The SAAC\textsubscript{1} group, No Access, is not represented in the table as it is merely a default group assumed for code fragments without any other access restriction, and no user or subject may be assigned the No Access group.

The attribute \textit{ugroup} is specified for each authorized user in GitBAC’s user list using the form: \textit{(User name, Group)}. Listing 4.3 shows an example of a GitBAC user list containing two users: Bob, an Employee, and Sue with public access.

Recall that each user’s location attribute is determined dynamically by GitBAC for each client request. If the client’s IP address is not a local IP address and the user is a member of the Employee or Admin groups, then the user’s group will be downgraded to Private for that request. If our example Employee, Bob, sends a request to GitBAC using a local IP address, then Bob’s request
belongs to the Employee group, otherwise Bob’s request belongs to the Remote Employee (i.e., Private) group. Employee dominates the Private access group, demonstrating support for hierarchical access control policies. Through the use of the SAAC model and dynamic attributes, GitBAC can implement more expressive and flexible access control policies than with conventional access control models alone.

<table>
<thead>
<tr>
<th>1</th>
<th>{Bob, Employee}</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>{Sue, Public}</td>
</tr>
</tbody>
</table>

**Listing 4.3**: Example GitBAC access entity list (users).

**Fragment Specification**

GitBAC uses encoded comments to delimit fragments. Similar to *ifdefs* used for controlling versions in target code, we use comments to control confidentiality and integrity in source code. Because the fragment boundaries are embedded in the source code, added and removed code automatically expands and contracts the fragment body. Typically, fragments are created and removed using a front-end like SaJE that authorizes fragment-level operations. However, GitBAC also analyzes and restores fragment boundaries that a developer has modified when code changes are communicated to GitBAC. This prevents developers from trying to bypass front-end restrictions, such as creating an unauthorized fragment, and directly modifying the source code in some other text editor.

A fragment label can be any unique alphanumeric sequence providing a human-readable association for the fragment, although currently fragment labels are comprised of a six character unique identifier generated by SaJE when a fragment is created. Labels are stored in a separate fragment access table where each label is bound to a set of access privileges (i.e., operations with a comma-separated list of authorized access groups). Listing 4.4 shows the syntax for an entry in GitBAC’s fragment access table.

| 1 | <fragment label> [EX] read:{<entity list>} write:{<entity list>} |

**Listing 4.4**: GitBAC fragment access table entry syntax.
The object attribute, “read”, indicates the read access operation and “write” indicates the write access operation. Absence from the braces implies denial of the particular operation unless the requestor inherits access through the access control group hierarchy (e.g., the “Private” group grants access to the particular artifact operation for all employees, local or remote). “EX” is an optional argument that indicates an exclusive fragment. GitBAC’s exclusive fragments are accessible only to subjects with that exact access level, allowing users with inherited privilege the ability to build target code even when having access to fragments containing conflicting code (e.g., each fragment contains a return statement).

The nature of fragment boundary specification allows us to adjust fragment granularity between whole lines and single characters. If we want a fragment with a length of one character, we would surround the character with start and end fragment delimiters. Fragments may be nested within other fragments, although this feature is not currently implemented in the front-end, SaJE. Support for nested fragments within SaJE is a task for future work.

**GitBAC Fragment Specification Examples**

We now revisit the example in Listing 4.2 where we have a single Public fragment, \( f_1 \), printing “This is a public statement!”, followed by a single Private fragment, \( f_2 \), printing “This is a private statement!”. We define two access operations for our access control policy: read and write. “read” indicates the read access operation (pulling code from the Git repository and displaying in a source code editor) and “write” indicates the write access operation (pushing code changes to the Git repository).

```
1 f1 read: {Public} write: {Public}
2 f2 read: {Private} write: {Private}
```

**Listing 4.5**: GitBAC example fragment access entries.

Listing 4.5 shows the resulting fragment access table created in GitBAC after these fragments are created in SaJE. Our hierarchical policy allows a user with either Employee or Remote Employee access (i.e., Bob) to read and write \( f_2 \). As \( f_1 \) is a Public fragment, only Public or Admin
users (i.e., Sue) may read or write to that fragment. The lack of an exclusivity flag for \( f_2 \) indicates that all access groups that are children of the Private access group have access to that fragment.

Revisiting our example with mutually exclusive fragments in Figure 4.9, the source code for the artifact \( \text{MyStuff.java} \) is shown in Listing 4.6. This code was taken directly from the source code generated by SaJE. The fragment access table for Listing 4.6 is shown in Listing 4.7.

```java
package stuff;
public class MyStuff {
    public MyStuff () {
    }
    public static String test () {
        //@START f3@
        return "an Employee’s test string";
        //@END f3@
        
        //@START f4@
        return "a Remote Employee’s test string";
        //@END f4@
    }
}
```

**Listing 4.6:** GitBAC mutually exclusive fragment example.

```plaintext
f3 read: {Employee} write: {Employee}
f4 EX read: {Private} write: {Private}
```

**Listing 4.7:** GitBAC mutually exclusive fragment access entries.

Using the above fragment access table, if Bob accesses the source code using an IP address GitBAC considers local, Bob will have access to fragment \( f_3 \). If Bob uses an IP address that GitBAC does not consider local, then Bob will have access to fragment \( f_4 \). Sue, on the other hand, does not have access to either \( f_3 \) or \( f_4 \) since she is a member of the Public access group. Note that there is potential for a human factor security leak when Bob transitions from the Employee to the Remote Employee access group (e.g., takes a notebook computer from his office on vacation). Read-restricted Employee-exclusive fragments could be present on a local device used by an entity with non-Employee authorization. Security leak detection is an active area of research and we leave
4.2.3 Importing an Access-controlled Project using GitBAC

The Git client imports a project into Eclipse by cloning the project and its change history from the Git server to the local Eclipse workspace using the Git command `git upload-pack`. `git upload-pack` presents the Git client with a list of version branches from which the client may select and initiate the project download to the workspace. GitBAC inserts itself between the Git client and the Git server, so that Git projects are first crosscut by the requestor’s access credentials. In this way, the developer never sees any part of any artifact for which his/her access credentials do not allow. The sequence diagram in figure 4.12 illustrates the import dialogue between the Git client, GitBAC, and the Git server.

4.2.4 Pushing Updates to a Project through GitBAC

Pushing updates to GitBAC using `git-push` operates essentially in reverse of importing. The Git client uses `git-push` to send previously unsent updates to GitBAC. GitBAC examines each update
from the client, expands the offset of the update in the artifact back to its original value, and
determines if the update pertains to a write-restricted fragment for the updating user. GitBAC
discards any update that violates the access control policy and passes allowed updates to the Git
server for normal application to the remote repository.

4.2.5 Transforming Artifacts

GitBAC transforms the artifact packfiles between developers and the Git repository by restricting
access to the contained artifact data according to the requesting entity and the configured access
control policy. However, preserving the confidentiality and integrity of artifacts in the Git version
control system is not as simple as restricting access to a single snapshot of the artifacts stored
within. Git maintains detailed change histories for each artifact. We cannot deny developers
access to an artifact’s change history as this would violate our usability and transparency goals.
Therefore, we must apply our access control transformation not only to the requested artifacts, but
also the portion of Git’s change history that accompanies them. Listing 4.8 is our algorithm for
transforming an import request packfile from GitBAC.

```
1 Fetch the artifacts and their change history from Git
2 For each artifact in Git packfile:
3   If the artifact is read-restricted
4     Record it in the transformation database (db)
5     Remove it from the import request
6   Else for each change in the artifact's change history:
7     Read-restricted fragment changes are:
8       Recorded in the transformation db
9       And then removed from the artifact change
10    Write-restricted fragment changes are:
11       Recorded in the transformation db
12       And then replace with developer changes if any
13 Restore all artifacts from the transformation db not in import
```

**Listing 4.8**: GitBAC import transformation algorithm.

Note: that we currently only support the fragment access operations of reading and writ-
ing fragment contents. Execution permission for fragment contents is assumed if read access is
granted. Listing 4.8 lines 12 and 13 replace any write-restricted fragments and artifacts with the
developer’s modifications to them, if any. Otherwise, the import would not be consistent with the
developer’s local changes and could cause overwrite or synchronization problems.

Similarly, we apply a reversing transformation to all artifacts and artifact history being sent
from a developer as an update to the Git repository. Listing 4.9 is our algorithm for transforming
an update request packfile from a developer to GitBAC.

```
1 Receive the changed artifacts and their history from developer
2 For each artifact in developer packfile:
3   If the artifact is write-restricted
4     Record it in the transformation database (db)
5     Remove it from the update request
6   Else for each change in the artifact’s change history:
7     Apply heuristics to repair fragment violations
8     Restore read-restricted fragments from transformation db
9     Write-restricted fragment changes are:
10        Recorded in the transformation db
11        And then replaced from the transformation db
12    Restore all artifacts from transformation db not in update
```

Listing 4.9: GitBAC update transformation algorithm.

Listing 4.9 lines 8 and 12 replace any read-restricted artifacts and fragments that were removed
when transmitted to the developer. These steps maintain consistency with the codebase stored in
the Git repository. We cannot assume that fragment boundaries will remain intact when code is
sent from the developer. Thus, our update heuristics look at each insert and delete change made
by a developer to determine if the change affected fragment boundaries. If a boundary is violated,
GitBAC records the violation in the transformation database and modifies the update, similar to a
write-restricted modification.

GitBAC’s access control monitor component addresses our design characteristics as a single
point of entry between developers and artifact stores. The transformation map satisfies our char-
acteristics for fragment granularity and automatic maintenance heuristics. The audit log addresses
our design characteristic for an audit trail of developer access activity. GitBAC’s implementation
of the SAAC_1 access control model allows for hierarchical access groups with dynamic attributes,
providing flexible and customizable access control policies. Coupled with our front-end developer interface, SaJE, our implementation of our concern model approach for software artifact access control satisfies our design goals for confidentiality, integrity, accountability, transparency, usability, and flexibility.
Chapter 5: EVALUATION

We claim that crosscutting concerns can be used to correctly model and enforce access control policies of software artifacts. And by doing so, that the amount of work required to maintain the access-controlled artifacts will be less than conventional methods of enforcing access control.

We also claim that abstracting the access control mechanism away from the storage mechanism and using a more flexible access control model leads to increased policy expressiveness. Lastly, we believe that a concern-based technique provides a more natural and intuitive means for access control of crosscutting features.

We have formulated our above claims into the following research questions:

Q1 Can a concern model for access control reliably enforce an access control policy?

Q2 How do conventional access control techniques affect maintenance when a policy crosscuts the codebase?

Q3 How do concern-oriented techniques affect maintenance when a policy crosscuts the codebase?

Q4 Do users prefer one technique over the other?

Answering these research questions allows us to determine if our design and implementation of a concern-based access control system supports our claims, thereby achieving our research objectives. To address these research questions, we conducted two laboratory studies and one human subject experiment. Our first study addressed Question 1 regarding our concern model’s preservation of confidentiality and integrity. Our second study addressed Question 2. Our third study was a human subject experiment where the subjects used either a conventional access control technique or our technique to maintain a toy codebase as access control policy and code changes occur. This last study addressed Questions 2, 3, and 4.
5.1 Study 1: Fragment Correctness

Our first study investigated the use of crosscutting concerns to correctly enforce access control. We used precision and recall as the basis of our metric for determination of correct access control enforcement [4]. For “correct” access control behavior, all of a concern fragment’s contents must stay within its boundaries and no non-fragment content must be introduced into a concern fragment’s boundaries. Thus, when a fragment is retrieved, the contents have both perfect precision and perfect recall (i.e., 1.0). A less than perfect score in precision indicates that some of the retrieved fragment content does not belong in the fragment (i.e., non-fragment content leaking into the fragment). A less than perfect score in recall indicates that the retrieved fragment content does not contain all of the content that it should (i.e., content leaking out of the fragment).

Perfect precision and recall are necessary components for access control as less-than-perfect results indicate a potential security breach. Developers with insufficient access privilege should never be able to see the contents from read-restricted fragments, preserving the confidentiality of the restricted content. Perfect fragment recall indicates that all fragment content remains within the fragment boundaries, providing confidentiality of the restricted fragment source code. Developers with insufficient access should not be able to modify the contents of write-restricted fragments, i.e., integrity. Precision measures the integrity of a fragment through the ratio of restricted fragment content to total fragment content. The goal of this study is to build confidence in our approach as an access control mechanism for maintaining the confidentiality and integrity of restricted fragments after numerous read and write operations to the artifact and measuring the precision and recall of restricted fragment content. Note that this study does not consider operations other than read and write.

5.1.1 Method

This study used a single test artifact, a1, with six non-nested fragments \{f_1, f_2, f_3, f_4, f_5, f_6\} distributed within the artifact’s body, b. Each fragment body was comprised of a single line of
Table 5.1: Study 1 character assignment.

<table>
<thead>
<tr>
<th>Artifact Section</th>
<th>Initialization Character</th>
<th>Insertion Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td>“A”</td>
<td>“G”</td>
</tr>
<tr>
<td>f2</td>
<td>“B”</td>
<td>“H”</td>
</tr>
<tr>
<td>f3</td>
<td>“C”</td>
<td>“I”</td>
</tr>
<tr>
<td>f4</td>
<td>“D”</td>
<td>“J”</td>
</tr>
<tr>
<td>f5</td>
<td>“E”</td>
<td>“K”</td>
</tr>
<tr>
<td>f6</td>
<td>“F”</td>
<td>“L”</td>
</tr>
<tr>
<td>b</td>
<td>“Y”</td>
<td>“Z”</td>
</tr>
</tbody>
</table>

Table 5.2: Study 1 access control policy for GitBAC test artifact a1.

<table>
<thead>
<tr>
<th></th>
<th>f1</th>
<th>f2</th>
<th>f3</th>
<th>f4</th>
<th>f5</th>
<th>f6</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>rw</td>
<td>r</td>
<td>rw</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>private</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
</tr>
</tbody>
</table>

characters with an assumed newline character at the end. Fragments were initialized to 320 characters unique to each fragment and $b$ was initialized to 3,500 of its own unique character, evenly distributed around the fragments (see Table 5.1). We chose this data format for our artifact and fragment content as this experiment did not need to create executable programs. Also, unique characters to each fragment and the artifact body made leak detection easier.

Table 5.2 provides our access control policy as an HRU matrix [27]. We defined our access subjects as $S = \{public, private\}$, the objects to be accessed as the set of fragments $O = \{f1, f2, f3, f4, f5, f6\}$, and the read and write operations (i.e., rights) $R = \{r, w\}$. We did not consider “private remote” access in this evaluation. Allowed operations for an entity for that fragment are indicated in each matrix cell. If the entity has no access to a fragment, then the cell is blank.

We began the test by downloading a public access view of the artifact through GitBAC as $a1$, i.e., fragments 3 and 6 were not present in the artifact. We iterated more than 1,500 times, each time randomly selecting one of the following operations:
1. Delete a random number of characters (from 1 to 50) from a randomly determined point in $a_1$ (including fragment content).

2. Insert a random number of unique characters (from 1 to 50) at a randomly determined point in $a_1$. Unique insertion characters were determined based on Table 5.1.

After each modification to the artifact, we updated the source code repository by pushing the artifact through GitBAC using public access privileges. During the update, GitBAC applied its fragment maintenance heuristics and the assigned access control policy to the received artifact, $a_1$, generating a modified artifact, $a_2$, which was stored in the Git repository. Recall that GitBAC restores any fragments excluded due to read-only restriction as part of its update process. Thus, GitBac restored $f_3$ and $f_6$ before saving $a_2$ to the repository.

Next, we applied a manual set of public access heuristics to $a_1$, generating $a_1'$. The manual heuristics, similar to GitBAC’s, simply guaranteed that fragment boundaries themselves are immutable (although their positions in the artifact may change), and that modifications to a write-restricted fragment were discarded. Next, we retrieved $a_2$ through GitBAC using public access privilege. GitBAC transformed $a_2$ using our access control policy, giving us $a_2'$. We compared $a_2'$ to $a_1'$. If the two were not equal, the entire test would fail, as this would indicate incorrect behavior in our heuristics for maintaining fragment structure integrity.

We then calculated a recall score for $a_1'$. We define a perfect recall score as 1.0 for each readable fragment. No score could be calculated for the read-restricted fragments in $a_1'$ as the public artifact did not contain them. These scores were calculated later using a private access artifact (see below). We calculated a recall score for each fragment by counting the number of each fragment’s allowed characters (e.g., “A” and “G” for $f_1$, “B” for $f_2$) and dividing each count by the total number of the fragment characters in the entire artifact. For example, the recall score for fragment $f_1$ was calculated as:

$$recall(f_1) = \frac{\text{count}\left(\{“A”, “G”\}, f_1\right)}{\text{count}\left(\{“A”, “G”\}, a_1'\right)}$$  \hspace{1cm} (5.1)
In order to calculate a precision score, we downloaded a private access copy of a2 (our access control policy gave a private entity access to all fragments). Since we are testing all fragments for leakage into their content either through a failure in our maintenance heuristics or our access control policy for the public entity, we define a perfect precision score as 1.0 for each writable fragment over the entire artifact using its initial letter and its insertion letter, and 1.0 for each write-restricted fragment using only its initial letter (e.g., the only character in f2 should be a “B”). To illustrate, the precision scores for fragments f1 and f2 were calculated as:

\[
\text{precision}(f1) = \frac{\text{count}\left(\{“A”, “G”\}, f1\right)}{\text{count}(f1)}
\]

\[
\text{precision}(f2) = \frac{\text{count}\left(\{“B”\}, f2\right)}{\text{count}(f2)}
\]

(5.2)

(5.3)

Additionally, we calculated recall scores for the two read-restricted fragments in a2: f3 and f6, as these two fragments were missing in the public copy of the artifact.

Lastly, we set a1 ← a1′ and continued to the next iteration. As the goal of this test required a perfect score in recall and precision, a failure of either in a single iteration would have caused the entire test to fail.

**5.1.2 Results**

Research Question 1 asks if our approach correctly enforces a specified access control policy. The access control policy for this study stated that a public user has read/write access to fragments f1 and f4, read-only access to fragments f2 and f5, and no access to fragments f3 and f6. To determine if our approach correctly enforces read and write access, we calculated fragment precision and recall scores every iteration. Fragments f1, f2, f4, and f5 all had perfect precision and recall scores, i.e., 1.0. Using private user access, fragments f3 and f6 also had perfect precision and recall scores. Thus, we can answer Question 1 affirmatively that our approach is capable of correctly enforcing a specified access control policy. Please note that GitBAC’s maintenance heuristics are considerably different than other concern maintenance heuristics. Other heuristic techniques can-
not be completely automated as they try to maintain the semantic aspect of the concern and have trouble automatically determining when new code has semantic relevance to the concern [55, 63]. Our heuristics only protect content within fragment boundaries. Tracking restricted content outside a fragment is beyond the scope of this paper and falls into future work involving declassification, policy evolution, and security leak detection.

5.2 Study 2: Maintenance Effect

In our second study, we examined the use of a conventional modularization technique for access control of software artifacts to determine if the technique affects the amount of software maintenance work. We define “work” as the number of separate changes made to the relevant code.

5.2.1 Method

We based our second study on a fictitious, but realistic scenario using a Freemium business model. We took three open source software applications and an application’s language extensions, and pretended that these applications were closed source. We removed a subset of features from each of them to create a feature-limited public distribution of each application for a bogus open source release.

The open source software applications we used were jEdit 4.3, JHotDraw 7, ArgoUML 0.35, and the Argo Java Language module. For each of these software packages, we arbitrarily selected one to two features from the published feature lists for the core software source code to remove from our study’s open source distributions. As this study focused on the additional maintenance created by duplicated features that crosscut a codebase, we analyzed each feature within the most current SVN revision for tangling. We define tangling as how many lines of code a feature comprises in an artifact. To gain confidence that we selected representative features in terms of tangling, we selected five to six additional features from each feature list and analyzed them as well. All analyzed features crosscut at least one artifact where the feature comprised less than 10% of the artifact’s total lines of code. Lastly, we confirmed that all selected features existed for
at least half of the change history. For GUI-based features, we replaced the code with a modal dialog message suggesting the purchase of the full package (Figure 5.1). Non-GUI features were simply removed. We shared as many artifacts as possible between private and public variants of the software, duplicating only the artifacts containing the feature code fragments.

For each selected feature, we traced backwards into the software’s SVN change history, identifying the earliest revision where the feature occurs. We created external markers identifying the fragment boundaries encompassing each feature’s code in the artifacts. The granularity of fragment specification in this study was a single line of code. To create two access-controlled variants of each software application, we imported the earliest version of each application into Eclipse as our bogus fully-featured private variant of the software. We applied modularized access control by extracting and duplicating the restricted artifacts into separate public and private packages. In two cases, some of the artifacts were completely removed from the public packages. To control which implementation was used during execution, we altered the project’s build path in Eclipse. This is a simple and common mechanism for switching between access controlled variants of a project. Both the public and private distributions of the applications were tested to ensure that none of our changes prevented the software from executing and that the selected features were removed. Below are the application-specific features that we removed for our bogus public distributions:

**jEdit**

We modified 2 fragments of jEdit’s GUI to display a status bar message to the user when the hypersearch features were selected in the menu. Six artifacts were entirely devoted to the hypersearch feature so we removed these completely from the public distribution.

![Figure 5.1: Message for removed GUI feature.](image-url)
JHotDraw

We modified 4 fragments in 2 GUI artifacts to display a modal dialog when the user tries to draw arrows or text areas, or change arrow tips. Two artifacts were entirely devoted to arrow and text area features so we removed these completely from the public distribution.

ArgoUML

We removed all graphic file format options, other than EPS, from the Export Graphics and Export All Graphics menu choices. This involved modifying one GUI artifact and one non-GUI artifact totaling five fragments.

ArgoUML Java Module

We removed Java method body code generation for ArgoUML class diagrams. Thus, when a user generates Java code for his/her class diagram in ArgoUML, classes and attributes are generated, but not methods. This involved modifying one fragment in one non-GUI artifact.

Next, we mined the SVN change history for each project and monitored any changes to the artifacts containing the feature code. Any change that occurred within a fragment’s boundaries was considered a unique change, meaning that the change would only have to be made to the private variant of the implementation and not to the public duplicate. Any change outside of the fragment’s boundaries would have to be completely re-applied to the public duplicate.

5.2.2 Results

Table 5.3 contains the results of our second study using a conventional file-based modularization technique for access control. Each artifact contained one to three fragments. These fragments contained the code that was specific to the removed feature and was the only code that was necessary to change between the public and private distributions of each project. As stated previously, we removed several entire artifacts from the public distributions of jEdit and JHotDraw as these arti-
facts were completely modularized with respect to the removed features. The “% Tangled” row in Table 5.3 shows the percentage of the entire artifact contained within its fragments. “% Duplicate Changes” shows the percentage of “Total Changes” that must be re-applied to artifact duplicates, i.e., \(1 - \left(\frac{\text{Fragment Changes}}{\text{Total Changes}}\right)\). The higher the percentage, the greater the number of redundant tasks that must be re-applied. “Total Changes” lists the number of changes pertaining to each artifact in the change history. “Fragment Changes” are the portion of “Total Changes” that apply to the code within the artifact’s fragments.

**Table 5.3:** Study 2 results.

<table>
<thead>
<tr>
<th></th>
<th>jEdit</th>
<th>JHotDraw</th>
<th>ArgoUML</th>
<th>ArgoUML Java</th>
</tr>
</thead>
<tbody>
<tr>
<td># Revisions</td>
<td>Artifact 1</td>
<td>Artifact 1</td>
<td>Artifact 1</td>
<td>Artifact 1</td>
</tr>
<tr>
<td># Fragments</td>
<td>17,469</td>
<td>505</td>
<td>505</td>
<td>19,120</td>
</tr>
<tr>
<td>% Tangled</td>
<td>2.23%</td>
<td>2.62%</td>
<td>3.10%</td>
<td>7.64%</td>
</tr>
<tr>
<td>% Duplicate Changes</td>
<td>97.6%</td>
<td>81.8%</td>
<td>97.4%</td>
<td>89.0%</td>
</tr>
<tr>
<td>Total Changes</td>
<td>700</td>
<td>44</td>
<td>384</td>
<td>182</td>
</tr>
<tr>
<td>Fragment Changes</td>
<td>17</td>
<td>8</td>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>

In our jEdit test, there were 700 changes made to Artifact 1. 17 of those changes involved the feature code within our two fragments. This means that 97.6% of the 700 changes would have to be duplicated in the public distribution of Artifact 1. In the JHotDraw test, 81.8% of the 44 changes in Artifact 1 would have to be duplicated in the public distribution. 97.4% of the 384 changes to Artifact 2 would have to be duplicated. In the ArgoUML test, Artifact 1 had 182 changes with 20 of them occurring within its 3 fragments. Thus, 89% of its changes would have to be duplicated in the public distribution. Artifact 2 requires 76.9% of its changes to be duplicated in the public distribution. ArgoUML’s Java Module Artifact 1 had only 45 changes over time. However, the feature code was only changed once (during the initial commit). This means that 97.8% of its changes would have to be duplicated in the public distribution of Artifact 1. These results suggest that conventional access control techniques increase the number of change tasks
in software maintenance when an access control policy crosscuts the codebase, thereby increasing
the amount of maintenance.

Table 5.3 shows that the greatest contributor to the percentage of duplicated change tasks is the
amount of tangling of the feature in the artifact. A low percentage of entanglement suggests a high
percentage of non-restricted code involved in the artifact, increasing the likelihood that any one
change to the artifact will involve non-restricted code and require application to each duplicated
artifact implementation. On the other hand, if a feature comprises all of the artifact code, then
the number of duplicated change tasks is zero. An example of this is the set of six hypersearch
artifacts that were not distributed in the public distribution of our jEdit test. Any change made
to these artifacts in the private distribution would not have to be duplicated on the public side as
they do not exist in the public distribution. This is a case where our access control policy nicely
overlaps with jEdit’s modularization of the hypersearch functionality, except for a single GUI-
oriented artifact. Thus, when using conventional techniques, completely modularized features
are good candidates for access control (e.g., Tasktop connectors). However, such modularization
may not be appropriate for an access control policy intended to reduce insider threat of a single
algorithm or method. Additionally, a finer-grained approach to access control may promote more
liberal selection of features for Freemium-like business models.

5.3 Study 3: Usability

Our third study was a human subject experiment and looked at how our concern-based technique
compares to a conventional file-base technique when an access control policy crosscuts a codebase.
We evaluated the aspects of maintenance reduction, policy implementation correctness, and usabil-
ity for the two techniques. We recruited undergraduate and graduate Computer Science students
at UTSA to perform access control-related tasks on a toy Java codebase (consisting of 9 packages
and 78 source code files) with a plain-language (English) access control policy, using SaJE and
a randomly selected access control technique (either file-based or concern-based). SaJE recorded
the time the subjects took to successfully complete each task, the number of change operations for
successful completion of each task, and the number of attempts before successful completion of each task. Each subject also completed an exit survey, gathering experiential and usability data, after the experiment.

5.3.1 Description of Subjects

This experiment involved 14 human subjects. To balance the number of subjects in each technique group, the students were assigned to either a conventional access control technique (the experimental control) or our concern-based technique based on their order in the experiment schedule. Only one subject could perform the experiment at a time, so odd-numbered subjects used the conventional technique and even-numbered subjects used the concern-based technique. The subjects were not required to know a particular programming language, but they were asked to have a basic understanding of software development concepts, such as source code, compiling, and executing within an integrated software development environment (e.g., Eclipse). Subjects were recruited solely through the department student email list.

5.3.2 Method

The experiment lasted no more than one hour and thirty minutes, and had 3 parts:

1. Access control training: 15 minutes

2. Performance of 4 experimental tasks: 20 to 70 minutes

3. Exit questionnaire: 5 minutes

5.3.3 Access control training

Our subjects were not required to have any prior training for, or experience in, access control of software artifacts. Before performing the experimental tasks, we provided each subject with a background of software artifact access control, SaJE training, and a description of the experimental scenario. The background and training were tailored to suit the particular access control technique
that each subject used in the experiment (conventional or concern technique). The format of the training was a narrated slideshow with instructional video captures of SaJE.

**Access control primer**

We provided a background in software artifact access control for each subject centered around the experimental scenario: a company’s desire to increase revenue by retrofitting an existing software product to the Freemium business model. We explained how access control should be used to protect artifacts in the scenario and tailored the explanation based on the subject’s particular experimental technique. Subjects using the conventional file-based technique were shown how to restrict access to artifacts by duplicating and grouping files into access-restricted directories. Subjects using our concern-based technique were shown how to restrict access to only the parts of the code within an artifact that were necessary (i.e., creating a fragment).

**SaJE training**

Each subject was shown an instructional video capture of the SaJE IDE. We showed each subject directory and file operations, the source code editor, code execution, text searching, and access control operations. The access control operations were appropriate to the technique to be used by the subject. File-based access control showed the duplication of artifacts into separate directories and restricting access to those directories. Concern-based access control showed access control of code fragments.

**Experimental scenario**

We described the scenario in which the experimental tasks would be performed and the subject’s role: the subject would perform the duties of an access control administrator for a fictitious software development company. We showed the experimental project’s codebase that would be used for the experiment and explained its file organization, loosely based on the model/view/controller design pattern [25]. The subjects were also reassured that they were not expected to create source
code. All code changes and locations for changes were provided to them. Lastly, we explained the requirements for successful completion of the experiment: the subject must correctly implement a provided access control policy on an existing codebase and correctly maintain the access control policy as a series of changes are made to the software project.

5.3.4 Experimental Tasks

The subjects were asked to perform four access control tasks on the provided codebase. For each task, the subject was given a text-based description of the task, including a plain language access control policy and all source code necessary for accomplishing the changes to the codebase. The subjects were given as much time as necessary to read through and comprehend each task description. We were available to answer questions regarding task clarifications and IDE-specific questions (e.g., how to search for text). Timing for each task did not start until the subject was finished reading the task description.

When a subject was ready to complete a task, he/she requested SaJE to test his/her solution via the SaJE menu. Successful completion of each task was determined by SaJE using the following process:

1. Automatically filter artifacts based on the testing access group in the menu selection

2. Automatically search all relevant artifacts for markers (e.g., method declarations)

3. Build and execute the codebase

4. Ask subject to perform a visual inspection of executing program (e.g., is menu option present for a removed feature)

Failure during any one of these steps prevented the task from completing successfully. The subject was not told specifically what was preventing successful completion, but he/she could not proceed to the next task until the issue was resolved and testing was successful.
Task 1: Remove a feature

The first task was to implement an access control policy on an existing codebase by removing a feature for the Public access group (see Figure 5.2). The access control groups in the access control policy were the Public group and the Private group. The Public group was for source code exclusive to the Public. All source code that was not restricted to an access group was shared by all access groups. The subject was asked to remove the text searching feature entirely from the codebase for the Public access group. The Private access group retained access to the text searching feature. The text search feature was selected for this task as it involved removing the entire “Search” option from the main menu, easing visual inspection by the test subject. This feature also reflected the feature selection choice for jEdit in Study 2. This step involved modifying five source code files.

Task 2: Add a feature

The second task asked the subject to add a “Build” feature to the codebase for both Public and Private access groups. All Java code for adding the feature, as well as all change locations were provided to the subject. “Build” was chosen as it is a common feature in other IDEs and was simple to implement in SaJE as it involved duplicating most of the code for executing a program. This task required changes to five source code files.
Figure 5.3: Study 3 Task 3 access control groups.

Task 3: Change the access groups

The third task involved a hierarchical change to the access control groups, and introduced two new groups: Private (Exclusive) and Contractor (see Figure 5.3). This task reflected the scenario of a company hiring a contract programmer to assist with development but wishing to protect certain intellectual property from the contractor. Both the Contractor and Private (Exclusive) access groups shared access to the Private access group. This task required changes to one source code file.

Task 4: Refactor a feature

The fourth task involved a refactoring of the Build feature to reduce redundant code. The Build feature was shared by all access groups. This task required changes to two source code files.

5.3.5 Exit Questionnaire

After each subject completed the four experimental tasks, he/she was asked to answer a questionnaire regarding his/her personal training and experience with overall software development, security, and access control, as well as provide usability feedback regarding the access control technique used during the experiment. The questionnaire’s qualitative data supplements the task
**Table 5.4:** Software artifact access control \( \text{SAAC}_0 \) configuration using \( \text{ABAC}_\alpha \).

### Basic sets and functions

- \( \text{UA} = \{ u\text{group} \} \)
- \( \text{SA} = \{ s\text{group} \} \)
- \( \text{OA} = \{ r\text{group}, w\text{group}, \text{exclusive} \} \)

- \( \text{P} = \{ \text{read, write} \} \)

- \( \text{Range}(u\text{group}) = \text{Range}(s\text{group}) = \text{Range}(r\text{group}) = \text{Range}(w\text{group}) = G \)

- \( \text{G} = \{ \text{No access, Public, Private, Employee, Contractor} \} \)

- Partial Order = No access \(<\) Public, No access \(<\) Private, Private \(<\) Employee, Private \(<\) Contractor

- \( \text{attType}(u\text{group}) = \text{attType}(s\text{group}) = \text{attType}(r\text{group}) = \text{attType}(w\text{group}) = \text{atomic} \)

- Thus, \( u\text{group}: U \rightarrow G \), \( s\text{group}: S \rightarrow G \), \( r\text{group}: O \rightarrow G \), \( w\text{group}: O \rightarrow G \), \( \text{exclusive}: O \rightarrow \text{true/false} \)

### Configuration points

1. **Authorization policies**

   - Authorization\textsubscript{\textit{read}}(s:S, o:O) \( \equiv (\neg \text{exclusive}(o) \land \text{rgroup}(o) \leq \text{sgroup}(s)) \)
     \( \lor (\text{exclusive}(o) \land \text{rgroup}(o) = \text{sgroup}(s)) \)

   - Authorization\textsubscript{\textit{write}}(s:S, o:O) \( \equiv (\neg \text{exclusive}(o) \land \text{wgroup}(o) \leq \text{sgroup}(s)) \)
     \( \lor (\text{exclusive}(o) \land \text{wgroup}(o) = \text{sgroup}(s)) \)

2. **ConstrSub**(u:U, s:S, \{\( s\text{group}, \text{val1} \}\):SASET) \( \equiv \text{val1} \leq u\text{group}(u) \land \text{val1} \neq \text{No Access} \)

3. **ConstrObj**(s:S, o:O, \{\( r\text{group}, \text{val1} \), \( w\text{group}, \text{val2} \), \text{exclusive}, \text{val3} \}\):OASET)
   \( \equiv \text{val1} \leq \text{sgroup}(s) \land \text{val2} \leq \text{sgroup}(s) \)

4. **ConstrObjMod**(s:S, o:O, \{\( r\text{group}, \text{val1} \), \( w\text{group}, \text{val2} \), \text{exclusive}, \text{val3} \}\):OASET)
   \( \equiv \text{val1} \leq \text{sgroup}(s) \land \text{val2} \leq \text{sgroup}(s) \)

Performance data recorded during the experiment to provide an overall gauge as to the usability of our technique compared to the conventional technique (i.e., task completion, efficiency, and satisfaction).

### 5.3.6 Access Control Model

Table 5.4 is the ABAC\( \alpha \) configuration for \( \text{SAAC}_0 \), used in this experiment to model the access control policy shown in Figure 5.3. It differs from \( \text{SAAC}_1 \) in that this model explicitly separates the internal groups, Private (Exclusive) and Contractor. Thus, the object exclusive flag in this experiment is not used. Also, \( \text{SAAC}_0 \) does not possess a location attribute for users as the human subject’s location was not a factor in the experiment.
5.3.7 Metrics

We measured maintenance overhead in three ways:

1. The time (in seconds) that a subject took to successfully complete each task

2. The number of artifact change operations that a subject performed to successfully complete each task

3. The number of failed attempts that a subject made before successfully completing each task

We defined a change operation as any operation that affected the physical contents of a source code file or directory, or the creation or deletion of an access rule. Specifically, these operations were adding a file/directory, deleting a file/directory, copying a file/directory (via copy/paste), moving a file/directory (via cut/paste), saving a file in the editor, adding an access restriction to a file/directory, removing an access restriction to a file/directory, creating a fragment, and deleting a fragment.

As stated previously: a task was successfully completed when 1) SaJE determined that all task markers were satisfied for each task’s access groups (i.e., code that needed to be removed for each access group had been removed, and code that needed to be added had been added), 2) the code successfully compiled, and 3) the subject visually inspected the GUI.

We measured qualitative usability of the access control technique in our exit questionnaire through the four criteria proposed in the USE questionnaire [49] and implemented by Kamalrudin, et al [33]. For our evaluation, we defined the criteria for the experimental access control techniques as:

1. Usefulness: how helpful and effective was the technique towards successful completion of the tasks

2. Ease of Use: how easy and simple was the technique to use (i.e., did the technique get in the subject’s way of completing a task)
3. Ease of Learning: how easy was the technique to learn and how much additional help was needed to successfully complete the tasks

4. Satisfaction: how complete were the technique’s capabilities and how willing would the subject be to use the technique again

Each usability question on the exit questionnaire used a five-part Likert scale: 1 = Strongly agree, 2 = Agree, 3 = Neither, 4 = Disagree, and 5 = Strongly disagree [44]. Lastly, our exit questionnaire asked questions pertaining to each subject’s experience and training in software development, security, and access control. The experiential questions used a four-part scale based on years of experience: 1 = None, 2 = Less than one year, 3 = Between one and three years, and 4 = more than three years.

5.3.8 Results

The group “Conventional” represents all of the subjects using the conventional technique in this experiment, i.e., the control group. Note that 2 of the 7 conventional technique subjects did not complete all experimental tasks within one hour and thirty minutes. The group “Concern” contains the subjects who used our concern-based technique for access control, i.e., the experimental group. All subjects using the concern-based technique completed the experiment.

For each quantitative data set (i.e., time, number of operations, and failures) and task, we used the Shapiro-Wilk test to determine if the data follow a normal distribution [61]. For normalized data, we used the Welch Two Sample t-test to determine statistical significance as we were only comparing 2 data sets. Our hypothesis is that our experimental technique reduces the amount of time, number of operations, and number of failures compared to the control technique. The null hypothesis is that our experimental technique does not lead to a reduction. Formally, we state the null and alternative hypotheses for the completion time data set as:

\[
H_0^{Time}: \mu_{Experimental}^{Time} - \mu_{Control}^{Time} = 0
\]

(5.4)
For data sets that did not follow a normal distribution, we used the Kruskal-Wallis non-parametric test. Most of the quantitative results of our evaluation indicated statistical significance and the rejection of the null hypothesis $H_0: \mu_{\text{Experimental}} = \mu_{\text{Control}}$ for a particular task, however we cannot be confident in this significance as our sample size was small.

**Subject Experience Levels**

Overall, the experience levels of both techniques’ subjects were close (see Figure 5.4). There was a 9% difference in programming experience, with the median subject having between 1 and 3 years of programming experience. There was no difference in security-oriented programming experience, with the median subject having less than 1 year of experience. There was no difference in computer network experience (including administrative and support experience), with the median subject having less than 1 year of experience. There was an 29% difference in team software development experience, with the median subject having less than 1 year of experience. There was no difference in access control policy implementation and maintenance experience, with the me-
median subject having no experience. There was a 14% difference in crosscutting concern experience (e.g., experience with concern tools like Mylyn or Tasktop), with the median subject having no experience. Only one subject had any concern-based experience (less than 1 year). While this subject was randomly selected to use the concern-based technique, the subject’s performance was average. We concluded that the subject’s experience with concern tools had little impact on the subject’s performance in the experiment.

**Task Performance: Completion Time**

Figure 5.5 shows the average performance results of the subject groups (in seconds) per task. Our concern technique performed better than the conventional technique in Task 1 (restrict access to a feature), with an average completion time of 422 seconds. The average completion time for the conventional group was 1203 seconds (test = Welch, $p = 0.00519$). Our concern technique performed better than the conventional technique in Task 2 (add a new feature), with an average completion time of 418 seconds, compared to the conventional group with 632 seconds (test = Welch, $p = 0.02239$). In Task 3 (change access groups), our concern technique had an average completion time of 335 seconds and the conventional technique fared slightly worse with an aver-
age completion time of 385 seconds (test = Welch, $p = 0.2114$). Our concern technique performed better than the conventional group in Task 4 (refactor a feature), with an average completion time of 342 seconds. The average time to finish Task 4 for the conventional group was 565 seconds (test = Welch, $p = 0.02007$).

**Task Performance: Number of Change Operations**

Figure 5.6 shows the average number of change operations performed by each technique’s subjects per task. Our concern technique performed better than the conventional technique in Task 1, with an average of 9.6 operations. The average number of operations for the conventional group was 39 (test = Kruskal-Wallis, $p = 0.00168$). Our concern technique performed better than the conventional technique in Task 2, with an average number of 7.3 operations, compared to 11.1 operations for the conventional group (test = Kruskal-Wallis, $p = 0.02004$). Our concern technique performed better than the conventional group in Task 3, with an average of 7 operations compared to 9.3 for the conventional group (test = Kruskal-Wallis, $p = 0.05967$). In Task 4, our technique outperformed the conventional technique with an average of 2.9 operations, compared to 10.2 operations for the conventional group (test = Welch, $p = 0.0002$).
Figure 5.7: Study 3 average number of failures by technique.

**Task Performance: Number of Failed Attempts**

Figure 5.7 shows the average number of failed test attempts performed by each technique’s subjects per task. Our concern technique performed better than the conventional technique in Task 1, with an average of 0.4 failures. The average number of failures for the conventional group was 2.7 (test = Kruskal-Wallis, \( p = 0.00429 \)). Our concern technique performed better than the conventional technique in Task 2, with an average number of 0.1 failures, compared to 0.4 failures for the conventional group (test = Kruskal-Wallis, \( p = 0.2542 \)). Our concern technique performed better than the conventional group in Task 3, with an average of 0.3 failures. The conventional group averaged 1.7 failures (test = Kruskal-Wallis, \( p = 0.02206 \)). In Task 4, our technique outperformed the conventional technique with an average of 0.6 failures, compared to an average of 2 failures for the conventional group (test = Kruskal-Wallis, \( p = 0.08913 \)).

**Technique Usability**

Figure 5.8 shows the average usability scores grouped by our four usability categories: usefulness, ease of use, ease of learning, and satisfaction. A score of 1 indicates that subjects strongly agree
with the technique’s usability in the particular category. A score of 5 indicates strong disagreement. The subjects more strongly agreed (average of 1.5) that our concern-based technique was more useful than the conventional technique, which had an average of 2. The subjects strongly agreed that our concern technique was easier to use (average of 1.3) compared to the conventional technique which had an average of 2. The subjects more strongly agreed (average of 1.6) that our concern technique was easier to learn and remember than the conventional technique, with an average of 2.1. Both techniques scored roughly the same in satisfaction for all groups (average of 1.9).

**IDE Influence**

Finally, our survey asked two questions about the SaJE IDE to help us determine if the IDE itself affected the results for one technique over the other. The first IDE question in the survey asked if the IDE facilitated use of the access control technique (i.e., did not get in the subject’s way). The conventional subjects felt the IDE was more helpful than the concern subjects with a difference of 17% and a median score of 1 (Strongly Agree). The second IDE question asked if the IDE had all necessary functionality for use of the technique. The two groups’ responses were separated by 1%
and a median score of 2 (Agree). Comments provided by test subjects indicated a desire for better text searching within artifacts and an “Undo” feature. These results lead us to believe the IDE may have slightly favored the conventional technique subjects.

5.3.9 Discussion

Based on the results of this experiment, our concern-based technique outperformed the conventional file-level technique for access control in terms of time, number of change operations, and failed testing attempts. The test subjects also preferred our concern-based technique in all categories, except satisfaction which was equivalent to the conventional technique.

We believe that the large disparity in time performance in Task 1 between conventional and concern-based groups is due to the time it took the conventional subjects to decide on an access-control modularization decomposition strategy for the codebase and implement the strategy. The disparity between technique groups diminished as the tasks progressed, indicating that the conventional subjects became increasingly familiar with the new modularization strategy. Our concern-based approach for access control did not seem to suffer this additional overhead as no modular decomposition was required and the subjects could immediately begin restricting the code necessary to complete the task. We believe that modularization memory would decrease, along with performance, in subsequent tasks if the conventional subjects switched their focus to other non-access control tasks and then returned to the access control tasks later.

The concern-based subjects’ performance was close to the conventional subjects’ in Task 3, with respect to time and operations. We believe this was due to SaJE’s current lack of support for embedded fragments, i.e., a fragment within another fragment. The access control policy provided to the subjects for Task 3 was a hierarchical access control policy. This task required the subjects to further restrict access to code within an already restricted fragment. When the test subjects were informed of the lack of embedded fragment support for Task 3, they had to discover an alternative means to express the hierarchical nature of the task. All subjects eventually discovered that they could use contiguous fragments to achieve the same result as embedded fragments. However, we
believe that the time taken by the concern-based subjects to make this discovery added to the time for completion of the task. Creating contiguous fragments also involved more change operations than simply embedding two new fragments into an existing fragment.

We analyzed the quantitative usability data for relationships in the variables using Pearson’s correlation coefficient. The two quantitative variables that showed the most correlation were task completion time and # of operations. The conventional technique showed high correlation \((r = 0.83834)\) and the concern technique showed medium correlation \((r = 0.46841)\). All other variable permutations showed low to no correlation \((-0.3 < r < 0.3)\). Figures 5.9 and 5.10 illustrate the correlations between task completion time (the horizontal axis) and # of operations (the vertical axis) as scatter plots for both the conventional and concern techniques.

Lastly, the test subjects tended to prefer our concern-based technique in 3 of the 4 usability categories: usefulness, ease of use, and ease of learning. Both techniques had equal scores in satisfaction. However, the concern-based subjects expressed dissatisfaction with our technique because of the lack of embedded fragment support. If our technique had included support for embedded fragments, the satisfaction score for our concern-based technique may have been better.

![Figure 5.9: Study 3 Conventional time and operations correlation.](image)
5.4 Evaluation Summary

The results from our two case studies and human subject experiment have provided answers to our four research questions. From our first study we conclude that our design and implementation of a concern-based approach is an effective means of controlling access to software artifacts, ensuring their confidentiality and integrity. Our second study showed that conventional artifact-based techniques for applying a crosscutting access control policy to three open source projects resulted in an increased amount of maintenance as code changes occur. Our third study demonstrated that file-level techniques required more maintenance than our concern-based technique as code and access control policy changes occur. Our third study also suggested that the users generally preferred the concern-based technique over the file-based technique.

Figure 5.10: Study 3 Concern time and operations correlation.
Chapter 6: CONCLUSION

When an access control policy perfectly overlays the dominant decomposition of a project’s modularization strategy, then conventional techniques for controlling access to software artifacts may suffice without bringing additional maintenance. But if the access control policy crosscuts the codebase, additional work is required to maintain consistency due to artifact duplication caused by the coarse granularity of the technique. Extra maintenance can increase errors, cause security leaks, and limit feature selection. Conventional techniques are also limited in policy expressivity as they are commonly implemented using static access control models built into artifact storage systems. Policy limitations can lead to incomplete or incorrect policies, which can increase the risk of security incidents or inhibit collaborative opportunities.

This thesis describes an approach using crosscutting concerns to implement and enforce access control policies of software artifacts. Because our approach is finer-grained than conventional techniques, the additional maintenance brought by conventional access control techniques is reduced. The abstracted nature of our access control monitor also allows for the use of flexible and dynamic access control attributes, in turn bringing greater policy expressiveness.

We have designed a concern model for applying software artifact access control policies and enforcing them. We have implemented our design in a front-end user interface prototype, SaJE, and a back-end access control monitor, GitBAC. SaJE is a simple Java IDE that allows software developers to visually assign access permissions to fragments as small as a single line of source code. GitBAC is a transparent access control monitor that enforces the confidentiality and integrity of access-restricted source code fragments according to pre-loaded ABAC$_{α}$ access control policy configurations.

We have evaluated our implementation in two case studies and one human subject pilot study. The results of these studies show that our approach provides a viable alternative to conventional access control techniques. Furthermore, the studies demonstrate that when an access control policy crosscuts a codebase, a concern-based access control technique is capable of sharing more code
than conventional techniques, preserving the original code modularization and reducing the main-
tenance required to keep shared code consistent. Our human subject experiment shows that our
 technique outperforms, is more intuitive, and easier to learn than a conventional technique. These
results suggest that, for crosscutting access control policies, our concern-based approach for soft-
ware artifact access control can reduce insider threat and increase opportunities for collaboration.

6.1 Future Work

There are several promising areas of future research where we feel that we can extend the contri-
butions of our software artifact access control technique:

6.1.1 Mainstream IDE Support

The user interface for our technique, SaJE, was created for our human subject experiment. Our
goal in creating SaJE was to limit the IDE functionality to only those functions necessary for the
experiment (Eclipse and Visual Studio can be overwhelming to the uninitiated). However, we
want to make our concern-based approach available to industry developers and conduct a real-
world experiment, monitoring and surveying developers’ use of our technique, both quantitatively
and qualitatively.

6.1.2 Automatic Mock Generation

Currently, mocks and stubs for testing using our concern-based technique can be manually spec-
ified within separate, exclusive fragments. This process can be tedious and introduce errors that
increase the threat of security leaks. To address the tedium of manual test code generation, Islam
and Csallner present a technique for dynamic mock class generation in programs that use Java
interfaces [29]. We see the possibility of automatically generating mocks in SaJE and GitBAC for
areas of restricted code that are assigned a “test only” access privilege and that can be automatically
restricted from release builds.
6.1.3 Security Leak Detection

Even with an access control policy in place, software development efforts may still suffer accidental and intentional fragment content declassifications (e.g., copying and pasting a block of text between fragments of different access levels). Therefore, our technique should provide an extensible mechanism for detection and notification of violations of an access control policy. GitBAC’s audit history can play an important role in determining the nature, magnitude, and accountability of a security incident.
Appendix A: USABILITY STUDY IRB APPROVAL

The University of Texas at San Antonio
Office of Research Integrity

Notice of Approval by Expedited Review

Date: February 8, 2013
IRB#: 13-092
Study Title: “Evaluating the usability of concern-based software artifact access control through empirical studies and analysis” (Funding Source: None)

To: Mark Robinson, M.S.
    Jianwei Niu, Ph.D., Faculty Sponsor, Department of Computer Science

From: Judith W. Grant, Ph.D., CIP, Director, Institutional Review Board

Date of Approval: February 8, 2013  Date of Expiration: February 7, 2014

The above referenced protocol was reviewed and approved by expedited review on behalf of the Institutional Review Board in accordance with the federal regulations (45 CFR 46 and all applicable subparts). This protocol was approved under expedited category 7.

No modifications may be made to the research plan, methodology, or any other aspect of the study without prior approval from the IRB, except in cases where changes are necessary to remove an immediate hazard to subjects. When modifications are made to eliminate an immediate hazard to subjects the IRB must be notified immediately.

If you wish to continue the research project beyond the expiration date you must submit a progress report at least three weeks before the expiration date. As a courtesy the IRB will send reminder notices; however, it is the responsibility of the investigator to submit the required information with ample time for IRB review. In addition, you are required to submit a closure report upon completion of the research project.

Should you have any questions regarding this letter, or need further assistance, please contact the IRB office at 210-458-6473 or send an email to irb@utsa.edu.

Items Approved: Protocol (v.initial.2/8/13), One consent form (v.2/8/13), One recruitment item (v.recruitment.2/8/13)

Study Sites: UTSA

JWG/jw

---

One UTSA Circle • San Antonio, Texas 78249 • (210) 458-6587 • (210) 458-6176 fax

Please retain this document for your study file.
IMPORTANT INFORMATION REGARDING YOUR APPROVED STUDY

PI: Mark Robinson

Study Title: Evaluating the usability of concern-based software artifact control through empirical studies and analysis

1) Your IRB # is _________________. It will help if you include this number and the title of your study on all correspondence relating to this protocol. Your study will have this number throughout its life.

2) Any change to the protocol requires prior IRB approval prior to implementation.

3) You have been approved to enroll _______________/00 subjects. If more subjects than originally approved are needed, please submit an amendment to increase accrual. Include a justification for the increase.

4) Changes to study personnel (addition or deletion) must be submitted to the IRB through the amendment process prior to any new personnel interacting with subjects or their private, identifiable information. All new study staff must meet the human subjects training requirement.

5) You are approved to conduct the research at: UTSA
   If need to add or delete sites, inform the IRB with an amendment request.

6) Your IRB approval expires on __________________.
   If you need to continue the research, submit a continuing review progress report prior to expiration. The IRB will send you a reminder as the expiration date approaches.

7) Records retention policy: The UTSA institutional policy for the retention of research records requires you to retain data and documentation related to the protocol for a minimum of three years following the completion date.

8) Submit a final closure report when all human subjects research activities have been completed.

9) Your options if you leave UTSA before the completion of the research are:
   ♦ Transfer the study to another UTSA investigator (amendment request) or
   ♦ Close the study at UTSA by informing the IRB (submit final report) and
   ♦ Cease research activity until you obtain IRB approval from the IRB at your new institution

10) Special Note: ________________________________
Appendix B: USABILITY STUDY CONSENT FORM

CONSENT TO TAKE PART IN RESEARCH AS A HUMAN SUBJECT
The University of Texas at San Antonio
Title of Project: Evaluating the usability of concern-based software artifact access control through empirical studies and analysis
Study sites: UTSA

Principal Investigator(s): Mark Robinson, M.S.

You are being asked to participate in a research study. This research study is part of a research project conducted by members of The University of Texas at San Antonio (UTSA). This form provides you with information about the study. You will also receive a copy of this form to keep for your reference. The Principal Investigator or his/her representative will provide you with any additional information that may be needed and answer any questions you may have. Read the information below and ask questions about anything you do not understand before you decide whether or not to take part. Your participation is entirely voluntary, and you can refuse to participate or withdraw at anytime without penalty or loss of benefits to which you are otherwise entitled.

Funding Source: not applicable

What is the purpose of the study? The purpose of the study is to evaluate how a concern-based technique affects the correctness of enforcing and implementing access control policies and maintenance. We are asking you to take part in this study because, as a Computer Science student, you have at least a basic understanding of software development within an Integrated Development Environment. 100 subjects are expected to take part in this study.

What will be done if you agree to take part in this research study? You will receive brief training regarding access control in the software development process and how to use the experimental tools. You will then be asked to perform a series of four tasks involving access control of an existing software project and minor code changes to the software artifacts. All code changes will be provided to you. The study will take place at the UTSA 1604 campus in the Science Building. The study should take less than 60 minutes to complete.

What are the possible discomforts and risks? There are no expected risks or discomforts associated with this study, however, some people may experience frustration while performing the tasks.

What are the possible direct benefits to the participant for taking part in this research? No benefit exists at this time.

What are the possible benefits to society from this research? The knowledge gained from this study may decrease the maintenance overhead associated with access control of software artifacts and increase the security of shared software projects.

Will there be any costs related to the research? There will be no cost to you if you participate in this study.
CONSENT TO TAKE PART IN RESEARCH AS A HUMAN SUBJECT
The University of Texas at San Antonio

Title of Project: Evaluating the usability of concern-based software artifact access control through empirical studies and analysis
Study sites: UTSA

**Will there be any compensation for participation?**
You will receive $10 USD for your participation in this experiment.

**If you do not want to take part in this study, what other options are available to you?** Your participation in this study is entirely voluntary. You are free to refuse to be in the study or to withdraw from the study at any time. Your refusal will not influence current or future relationships with The University of Texas at San Antonio.

**How will your privacy and the confidentiality of your research records be protected?**
We will not record any personally identifiable information about you in the study. The data and information gathered during this study may be used by UTSA and published and/or disclosed by UTSA to others outside of UTSA but your identity will not be disclosed.

Your research records will not be released without your consent unless required by law or a court order. Your records may be viewed by the Institutional Review Board, but the confidentiality of your records will be protected to the extent permitted by law. The data resulting from your participation may be used in publications and/or presentations but your identity will not be disclosed.

**How can you withdraw from this research study and whom should you call if you have questions?**
If you wish to stop your participation for any reason, please contact the principal investigator Mark Robinson at (210) 843-0130 or tell the research personnel. Throughout the study, the researchers will notify you of new information that may become available and that might affect your decision to remain in the study.

If you have questions now, you may ask the principal investigator (or representative staff). If you have questions later, you may contact Mark Robinson, M.S. at (210) 843-0130 or Jianwei Niu, Ph.D. at (210) 458-7360.

If you have questions pertaining to your rights as a research participant, or if you have complaints, concerns, or questions about the research, you should contact the UTSA Institutional Review Board at Email: irb@utsa.edu. Phone: 210-458-6473.

The UTSA Institutional Review Board (IRB) has approved the use of human participants for this study. Your participation in this research is voluntary. You may discontinue participation at any time during the research activity.
CONSENT TO TAKE PART IN RESEARCH AS A HUMAN SUBJECT
The University of Texas at San Antonio
Title of Project: Evaluating the usability of concern-based software artifact access control through empirical studies and analysis
Study sites: UTSA

You have been informed about this study's purpose, procedures, possible benefits and risks. You have been given the opportunity to ask questions before you sign, and you have been told that you can ask other questions at any time.

You voluntarily agree to participate in this study. By signing this form, you are not waiving any of your legal rights.

You will be given a copy of this form to keep.

__________________________
Printed Name of Subject

__________________________
Signature of Subject

__________________________
Date

Mark Robinson
Printed Name of Person Obtaining Consent

__________________________
Signature of Person Obtaining Consent

UTSA IRB #13-092
APPROVED
2/11/2013
Appendix C: USABILITY STUDY QUESTIONNAIRE

Research Study: Evaluating the usability of concern-based software artifact access control through empirical studies and analysis

Exit Questionnaire

Subject # ________

The access control technique makes policy implementation simple (i.e., few steps).
○ Strongly agree ○ Agree ○ Neither ○ Disagree ○ Strongly Disagree

The access control technique makes code maintenance simple.
○ Strongly agree ○ Agree ○ Neither ○ Disagree ○ Strongly Disagree

The access control technique is intuitive (i.e., instinctive).
○ Strongly agree ○ Agree ○ Neither ○ Disagree ○ Strongly Disagree

The access control technique is easy to use (i.e., doesn't get in the way of your tasks).
○ Strongly agree ○ Agree ○ Neither ○ Disagree ○ Strongly Disagree

Does the IDE allow for easy use of the access control technique?
○ Strongly agree ○ Agree ○ Neither ○ Disagree ○ Strongly Disagree

The access control technique is easy to learn.
○ Strongly agree ○ Agree ○ Neither ○ Disagree ○ Strongly Disagree

I did not need any additional help during the experiment.
○ Strongly agree ○ Agree ○ Neither ○ Disagree ○ Strongly Disagree

I am satisfied with the access control technique's capabilities (i.e., would not change anything).
○ Strongly agree ○ Agree ○ Neither ○ Disagree ○ Strongly Disagree

I would encourage others to use the technique for access control tasks.
○ Strongly agree ○ Agree ○ Neither ○ Disagree ○ Strongly Disagree
The access control technique is fun to use.

- Strongly agree
- Agree
- Neither
- Disagree
- Strongly Disagree

The IDE has all functionality necessary to use the access control technique.

- Strongly agree
- Agree
- Neither
- Disagree
- Strongly Disagree

How much programming experience do you have?

- None
- < 1 year
- between 1 and 3 years
- > 3 years

How much security-oriented programming experience do you have?

- None
- < 1 year
- between 1 and 3 years
- > 3 years

How much computer network experience do you have?
For example: admin, managing user accounts, help desk

- None
- < 1 year
- between 1 and 3 years
- > 3 years

How much team software development experience?
This includes design, documentation, testing, management, and support.

- None
- < 1 year
- between 1 and 3 years
- > 3 years

How much experience with access control policy implementation/maintenance, e.g., restricting access to web pages, source code, documents, wikis, etc.?

- None
- < 1 year
- between 1 and 3 years
- > 3 years

How much experience with concern-oriented techniques and/or tools, such as Mylyn or Tasktop?

- None
- < 1 year
- between 1 and 3 years
- > 3 years
Appendix D: USABILITY STUDY TASKS

Human Subject Experiment Task 1 Description

In an effort to gain new users of its IDE, FooSoft decides to open source the product using a freemium approach. They will release to the Public all of the source code EXCEPT for the Search feature.

Access Control Policy:
There are 2 access groups: Private (those developers within FooSoft) who will have access to all of the codebase, and Public (those developers outside of FooSoft) who will have access to all of the codebase EXCEPT for the Search feature. No trace of the Search feature will remain in the Public source code.

You must:
1. Create two variants of the IDE codebase: one suited for Public developers and one suited for Private developers.

2. Test both your Public and Private codebase variants by clicking on "Task 1" in the Menu and "Test Public" and then "Test Private". When you have successfully completed both of the Public and Private tests, you may proceed to the next task.

Task steps:
FOR THE PUBLIC ACCESS USER ONLY:
STEP 1. Remove access to the file src/action/FindAction.java
STEP 2. Remove access to the search code in src/action/ActionFactory.java
   Note: there are 2 sections in this file to change.
   Search for the word "SEARCH".
   Remove everything to the ending comment "\END SEARCH FEATURE"
STEP 3. Remove access to the search code in src/controller/ControllerMain.java
   Note: Search for the word "SEARCH"
   Remove everything to the ending comment "\END SEARCH FEATURE"
STEP 4. Remove access to the search code in src/editor/MarcosEditor.java
   Note: Search for the word "SEARCH".
   Remove everything to the ending comment "\END SEARCH FEATURE"
STEP 5. Remove access to the search code in src/gui/AppMenuBar.java
   Note: Search for the word "SEARCH".
   Remove everything to the ending comment "\END SEARCH FEATURE"
Human Subject Experiment Task 2 Description

A developer with Public access adds a Build function to the IDE, allowing one to compile a project without executing it (a common feature in other IDEs). This new feature should be available to both the Public and Private access groups.

Access Control Policy:
All users may access the Build feature.

You must:
1. Modify your two variants of the codebase, Public and Private, so that both access groups can use the new Build feature.

2. Test both your Public and Private codebase variants by clicking on "Task 2" in the Menu and "Test Public" and then "Test Private". When you have successfully completed both of the Public and Private tests, you may proceed to the next task.

Task steps:
FOR ALL ACCESS USERS:
1. Add a new file: src/action/BuildAction.java
   (see code below)
2. Add code to src/action/ActionFactory.java
   2 sections
3. Add code to src/controller/ControllerMain.java
4. Add code to src/editor/MarcosEditor.java
5. Add code to src/gui/AppMenuBar.java

CODE TO ADD:
STEP 1: BuildAction.java

```java
package action;

import editor.MarcosEditor;
import editor.MarcosEditorPanel;
import gui.AppContentPane;
import java.awt.event.ActionEvent;
import javax.swing.AbstractAction;
import javax.swing.JOptionPane;
import javax.swing.JTree;
import access.MarcosAccessControlPolicy;
import model.MarcosDirectory;
import model.MarcosFile;
import model.MarcosNodeFactory;
import model.MarcosProject;
```
import model.MarcosTreeNode;
import model.MarcosWorkspace;

import controller.ControllerMain;
import controller.ControllerTreeNode;

class BuildAction extends AbstractAction {
    private static final long serialVersionUID = 1L;
    private ControllerMain myController = null;
    private AppContentPane myContentPane = null;

    public BuildAction(ControllerMain controller, AppContentPane contentPane) {
        //String text, ImageIcon icon,
        //        String desc, Integer mnemonic) {
        super("Build", null);
        //        String desc, Integer mnemonic) {
        putValue(SHORT_DESCRIPTION, "Build");
        //putValue(MNEMONIC_KEY, mnemonic);
        myController = controller;
        myContentPane = contentPane;
    }

    public void actionPerformed(ActionEvent e) {
        //for now, try to run the open file in the editor
        //if(myController.focusObject instanceof MarcosEditor) {
            MarcosEditorPanel panel = myController.getSelectedEditorPanel();
            if(panel == null) {
                JOptionPane.showMessageDialog(myContentPane,
                    "No file opened in an editor!",
                    "Invalid Selection",
                    JOptionPane.ERROR_MESSAGE);
                return;
            }
            if(panel.hasChanged()) {
                int n = JOptionPane.showConfirmDialog(myContentPane,
                    "Do you want to save your changes before running this
                    file?",
                    "Save Changes",
                    JOptionPane.YES_NO_CANCEL_OPTION);
                if(n == 0)
                    myController.saveFile();
                else if(n == 2)
                    return;
            }
            myController.buildFile(MarcosAccessControlPolicy.ACCESS_LEVEL_ADMIN);
        //}
    }
}
STEP 2: ActionFactory.java
Note: there are 2 sections in this file to change.
Search for "BUILD FEATURE"
First section
   public static final int ACTION_BUILD_PROJECT = 45;

Second section
   case ACTION_BUILD_PROJECT:
       return new BuildAction(controller, contentPane);

STEP 3: ControllerMain.java
Search for "BUILD FEATURE"
   public void buildFile(int accessLevel) {
       //run the file that is currently visible in the editor
       MarcosEditorPanel panel = getSelectedEditorPanel();
       panel.getEditor().buildNode(console, accessLevel, acp,
       getAccessLevelName(accessLevel));
   }

STEP 4: MarcosEditor.java
Search for "BUILD FEATURE"
   public void buildNode(ConsolePanel console, int accessLevel,
   MarcosAccessControlPolicy acp, String accessName) {
       console.clearOutput();

       console.appendLine("BUILDING AS " +
       ControllerMain.getAccessLevelName(accessLevel).toUpperCase(), true);

       if(myTreeNode == null) {
           System.err.println("No tree node found!");
           return;
       }
       MarcosTreeNode userNode = (MarcosTreeNode)
       myTreeNode.getUserObject();
       if(userNode == null) {
           System.err.println("No user node found!");
           return;
       }
       if(!(userNode instanceof MarcosFile))
           return;
       MarcosProject myProject = ((MarcosFile) userNode).getMyProject();
       if(!myProject.buildAll(accessLevel, acp, accessName)) {

112
//System.out.println(myProject.compileErrors.size());
String errOutput = "";
for(String err : myProject.compileErrors) {
    if(errOutput.length() > 0)
        errOutput += System.getProperty("line.separator");
    errOutput += err;
}
console.appendLine("BUILD ERRORS:", true);
console.appendLine(errOutput, true);
} else
    console.appendLine("Finished building", true);

STEP 5: AppMenuBar.java
Search for "BUILD FEATURE"
    JMenuItem = new
    JMenuItem(Factory.createAction(Factory.ACTION_BUILD_PROJECT,
    controller, contentPane));
    menuItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_B,
    ActionEvent.META_MASK));
    menuItem.setIcon(null);
    mainMenu.add(menuItem);
    mainMenu.addSeparator();

    mainMenu.addSeparator();
Human Subject Experiment Task 3 Description

FooSoft hires a developer as a contractor to assist with internal development of their IDE. However, FooSoft feels their text search method is proprietary and too sensitive for a contractor to see. At the same time, they wish to make text searching available to the contractor. FooSoft decides to implement a naive text searching algorithm for the contractor, in place of their proprietary algorithm.

Access Control Policy:
There are now 4 access groups: Public, Private, Private (Exclusive), and Contractor. Public, Private (Exclusive), and Contractor are exclusive of each other (i.e., nothing belonging to any one of these groups will be seen by the other two). The Private access group is shared by Private (Exclusive) and Contractor (i.e., anything belonging to the Private access group can be seen by the Private (Exclusive) and Contractor access groups).

The existing algorithm in the findSubstringPosition method in the MarcosEditor class should be accessible only by Private (Exclusive). The Contractor access group should have an alternative algorithm (see below). The Public access group should continue to not have access to the Find feature.

You must:
1. Create two variants of the codebase, Private (Exclusive) and Contractor. Both of these access groups will share all Private code up to this point. The code that they will not share is the algorithm within the method findSubstringPosition in MarcosEditor.

2. Test your Public, Private (Exclusive), and Contractor codebase variants by clicking on "Task 3" in the Menu and "Test Public" and then "Test Private" and
then "Test Contractor". When you have successfully completed all tests, you may proceed to the next task.

**Task steps:**
1. Modify findSubstringPosition method in src/editor/MarcosEditor.java for access group Contractor.
   Access group Private (Exclusive) will keep existing method code

**CODE TO CHANGE:**
**FOR THE CONTRACTOR ACCESS USER ONLY:**
**STEP 1:** MarcosEditor.java
   search for "public int findSubstringPosition"
   replace the line
   return s2.toUpperCase().indexOf(s1.toUpperCase(), p);
   with
   //naive search
   for(int i = p; i < (s2.length() - s1.length() + 1); i++) {
       boolean found = true;
       for(int j = 0; j < s1.length(); j++) {
           if(s1.toUpperCase().charAt(j) != s2.toUpperCase().charAt(i + j))
               found = false;
       }
       if(found)
           return i;
   }
   return -1;
Human Subject Experiment Task 4 Description

The Public developer who created the Build feature decides to refactor his implementation of the build method (in Task 2) to eliminate redundant code. This involves removing his previous Build method and better utilizing the existing method executeNode.

Access Control Policy:
All users may access the refactored Build feature.

You must:
1. Modify your three variants of the codebase, Public, Private (Exclusive), and Contractor, so that all access groups can use the refactored Build feature.

2. Test your Public, Private (Exclusive), and Contractor codebase variants by clicking on "Task 4" in the Menu and "Test Public" and then "Test Private" and then "Test Contractor". When you have successfully completed all tests, you may proceed to the Exit Questionnaire.

Task steps:
FOR ALL ACCESS USERS:
1. Modify code in controller/ControllerMain.java
   see below
2. Modify code in editor/MarcosEditor.java
   see below

CODE TO ADD:
STEP 1: controller/ControllerMain.java
search for "BUILD FEATURE"
replace
panel.getEditor().buildNode(console, accessLevel, acp, getAccessLevelName(accessLevel));
with
   panel.getEditor().executeNode(console, accessLevel, acp, getAccessLevelName(accessLevel), true);

STEP 2: editor/MarcosEditor.java
Note: there are 2 sections in this file to change.
search for "BUILD FEATURE"
STEP 2.a replace the entire buildNode method with
public void executeNode(ConsolePanel console, int accessLevel, MarcosAccessControlPolicy acp, String accessName) {
   executeNode(console, accessLevel, acp, accessName, false);
}

STEP 2.b just below the "BUILD FEATURE" section, replace the entire executeNode method with
public void executeNode(ConsolePanel console, int accessLevel, MarcosAccessControlPolicy acp, String accessName, boolean buildOnly) {
    console.clearOutput();
    console.appendLine("BUILDING AS " + ControllerMain.getAccessLevelName(accessLevel).toUpperCase(), true);
    if(myTreeNode == null) {
        System.err.println("No tree node found!");
        return;
    }
    MarcosTreeNode userNode = (MarcosTreeNode) myTreeNode.getUserObject();
    if(userNode == null) {
        System.err.println("No user node found!");
        return;
    }
    if(!(userNode instanceof MarcosFile))
        return;
    MarcosProject myProject = getMyProject();
    if(myProject == null)
        return;
    if(myProject.buildAll(accessLevel, acp, accessName)) {
        console.appendLine("Finished building", true);
        if(!buildOnly)
            ((MarcosFile) userNode).runMain(console, accessLevel);
    } else {
        //System.out.println(myProject.compileErrors.size());
        String errOutput = "";
        for(String err : myProject.compileErrors) {
            if(errOutput.length() > 0)
                errOutput += System.getProperty("line.separator");
            errOutput += err;
        }
        console.appendLine("BUILD ERRORS:", true);
        console.appendLine(errOutput, true);
    }
}
Appendix E: USING THE SAJE IDE

Figure E.1 shows a sample project opened in the SAJE IDE. When SAJE starts, it will open the default workspace, listing all available projects in the directory tree on the left side. To open a source file (a software artifact), locate the file in the directory tree and double-click the file’s name. The file will open in the source editor on the right side of SAJE. To run the source file, select “Run” in the top menu, and click on “Run” (Figure E.2). This will execute the currently open source file at an “Administrator” level of access (i.e., no access control restrictions). Figure E.3 shows the output in the bottom pane of SAJE when executing the source file in the sample project. The output pane also has a “Kill” button which will halt the executing source file.

To create a new source file in a selected directory, click on “File” in the top menu, and the click “New File” (Figure E.4). The user will be prompted to name the new source file (Figure E.5). Figure E.6 shows the newly created source file open in the SAJE editor. The user may select between multiple opened source files in the editor by clicking on one of the buttons at the top of the editor.

A directory (and all of its contained source files and sub-directories) may be copied by selecting the directory in the directory tree and clicking “Edit” in the top menu and then clicking “Copy” (Figure E.7). To finish the copying process, select a target project or directory in the directory tree and click “Paste” in the same “Edit” top menu. The copied directory will assume the name of the directory from which it was copied, unless that name already exists where it was copied. In such a case, an underscore and a digit will be automatically appended to the end of its name (Figure E.8). To rename a directory, select “Directory” from the top menu and then click on “Rename Directory” (Figure E.9). The user will be prompted to supply a new, unique name for the directory.

E.1 Restricting access to a project in SAJE using a conventional approach

SAJE supports the conventional software artifact access control technique of file-level modular decomposition. Below is a simple and very specific access control policy for the sample project.
**Figure E.1:** Sample project in the SAJE IDE.

**Figure E.2:** Running a source file in SAJE.
Figure E.3: The program output pane in SAJE.

Figure E.4: Creating a new source file in SAJE.

Figure E.5: Naming a new source file in SAJE.

Figure E.6: Viewing the new source file in the SAJE editor.
Figure E.7: Copying a directory in SAJE.

Figure E.8: Pasting a directory in SAJE.

Figure E.9: Renaming a directory in SAJE.
Figure E.10: File-level decomposition to control access of a restricted artifact.

Listing E.1: Example access control policy.

To implement this access control policy using file-level modular decomposition, the restricted class (AppMain) must be duplicated and stored in separate access-restricted directories. The result of this technique is that only public users will have access to the public “println” statement and only private users will have access to the private “println” statement in the AppMain class. The first step involves copying the main directory and then renaming both main directories to something more representative (Figure E.10).

Next, the package declarations in each source file should be changed to suit the new directory names. The public “println” statement should be removed from the private variant of the AppMain
class and the private “println” statement should be removed from the public variant of the AppMain class. Next, restrict access to the private directory so that only private access users may access its contents by selecting “Access” on the top menu and clicking on “Add Access Restriction” (Figure E.11). The user will be prompted for the access group that will be allowed to access the selected directory (Figure E.12). Then restrict access in a similar fashion to the public directory so that only public users may access the directory. Figure E.13 shows the access restricted results. Private access directories and files are colored orange. Public access directories and files are colored green. Blue directories and white files are the default colors and may be accessed by all users. Figure E.13 also shows how to test the correctness of the access restrictions for a project. Click on “Run” in the top menu and then click “Run Private” to execute the source file at a private level of access, or “Run Public” to execute the source file at a public level of access. Figure E.14 shows the output after running the public variant of the AppMain class.
Figure E.14: Output after executing the public variant of AppMain in SAJE.

Figure E.15: Restricting access to a code fragment.
Figure E.16: Selecting the access group for the code fragment.

Figure E.17: Public and private restricted fragments in the SAJE editor.

Figure E.18: Executing the public fragment of AppMain in SAJE.
Figure E.19: Output after executing the public fragment of AppMain in SAJE.
E.2 Restricting access to a project in SAJE using a concern-based approach

To implement this access control policy using concern-based access control, one restricts access to the lines of code (or fragment) that are specified in the access control policy. The users first open the restricted class in the SAJE editor (AppMain). After, selecting the source code line(s) of the fragment for restricting access, the user clicks on “Access” on the top menu and clicks on “Add Access Restriction” (Figure E.15). The user will be prompted for the access group that will be allowed to access the fragment (Figure E.16). Figure E.17 shows the access restricted results. Private access fragments are colored orange. Public access fragments are colored green. White lines of source code may be accessed by all users. To test the correctness of the access restrictions for a project, click on “Run” in the top menu and then click “Run Private” to execute the source file at a private level of access, or “Run Public” to execute the source file at a public level of access (Figure E.18). Figure E.19 shows the output after running the AppMain class at a public level of access.
Appendix F: SAAC\textsubscript{cw}: A CHINESE WALL VARIATION OF SAAC

A Chinese Wall access control policy tries to restrict access to information where conflicts of interest exist. For example, a software development firm that contracts with companies who compete with each other may try to protect their clients’ business interests by disallowing their developers from accessing more than a single client competitor’s specific software artifacts. Such a policy protects the development company’s clients by reducing the risk of insider attack and may be required by law for certain kinds of clients (e.g., companies that provide financial advice).

\(\text{ABAC}_\alpha\) does not currently support mutable attributes. Park, et al. show the application of mutable attributes using their UCON\textsubscript{ABC} usage control model, along with several simple example policies, including Chinese Wall [58]. Using the UCON\textsubscript{ABC} specification for attribute updates and Park’s Chinese Wall example, I have extended my SAAC model to provide mutually exclusive access to objects that belong to competing companies. Table F.1 is the \(\text{ABAC}_\alpha\) configuration for \(\text{SAAC}_{cw}\) and describes a very simple access control policy for a single conflict of interest scenario.

\(\text{CO}\) is a set of mutually exclusive companies, i.e., those where a conflict of interest exists. A subject has a set of allowed companies although the set will have at most two elements: null and an element in \(\text{CO}\). Drawing from UCON\textsubscript{ABC}’s attribute update specification, mutation points are functions that process either before (preUpdate) or after (postUpdate) the indicated configuration point. In my \(\text{SAAC}_{cw}\) configuration, the mutation point \(\text{Authorization}_{\text{read}}\) updates the subject’s \(\text{accessedCo}\) set following a successful read authorization (the \(\text{Authorization}_{\text{read}}\) configuration point). A subject may only access objects with a non-null \(\text{co}\) attribute if the subject has not yet accessed an object with a non-null \(\text{co}\) attribute (i.e., the subject’s \(\text{accessedCo}\) set contains only null) or the object’s \(\text{co}\) value is in the subject’s \(\text{accessedCo}\) set.
Table F.1: Software artifact access control SAAC\textsubscript{cw} configuration using ABAC\textsubscript{a}.

**Basic sets and functions**

- \( UA = \{ugroup\} \)
- \( SA = \{sgroup, accessedCo\} \)
- \( OA = \{rgroup, wgroup, exclusive, co\} \)
- \( P = \{read, write\} \)
- \( G = \{No access, Public, Private, Employee, Contractor\} \)

Partial Order = No access < Public, No access < Private, Private < Employee, Private < Contractor

- \( CO = \) set of mutually exclusive companies
- \( \text{Range}(ugroup) = \text{Range}(sgroup) = \text{Range}(rgroup) = \text{Range}(wgroup) = G \)
- \( \text{attType}(ugroup) = \text{attType}(sgroup) = \text{attType}(rgroup) = \text{attType}(wgroup) = \text{atomic} \)

Thus, \( ugroup: U \rightarrow G \), \( sgroup: S \rightarrow G \), \( rgroup: O \rightarrow G \), \( wgroup: O \rightarrow G \), exclusive: \( O \rightarrow \text{true/false} \), accessedCo: \( S \rightarrow \mathbb{2}^CO \), co: \( O \rightarrow CO \)

**Configuration points**

1. **Authorization policies**
   - Authorization\textsubscript{read}(s:S, o:O) \( \equiv ((\neg\text{exclusive}(o) \land rgroup(o) \leq sgroup(s)) \land (co(o) = \text{null} \lor\text{accessedCo}(s) = \emptyset \lor co(o) \in\text{accessedCo}(s))) \)
   - Authorization\textsubscript{write}(s:S, o:O) \( \equiv ((\neg\text{exclusive}(o) \land wgroup(o) \leq sgroup(s)) \land (co(o) = \text{null} \lor\text{accessedCo}(s) = \emptyset \lor co(o) \in\text{accessedCo}(s))) \)

2. **ConstrSub**(u:U, s:S, \{(sgroup, val1)\}:SASET) \( \equiv \) \( \text{val1} \leq ugroup(u) \land \text{val1} \neq \text{No Access} \)

3. **ConstrObj**(s:S, o:O, \{(rgroup, val1), (wgroup, val2), (exclusive, val3),(co, val4)\}:OASET) \( \equiv \) \( \text{val1} \leq\text{sgroup}(s) \land \text{val2} \leq sgroup(s) \)

4. **ConstrObjMod**(s:S, o:O, \{(rgroup, val1), (wgroup, val2), (exclusive, val3), (co, val4)\}:OASET) \( \equiv \) \( \text{val1} \leq\text{sgroup}(s) \land \text{val2} \leq sgroup(s) \)

**Mutation points**

1. **postUpdate**
   - Authorization\textsubscript{read}(s:S) : accessedCo(s) = accessedCo(s) \cup co(o)
   - Authorization\textsubscript{write}(s:S) : accessedCo(s) = accessedCo(s) \cup co(o)
Appendix G: SAJE POLICY FLEXIBILITY DEMONSTRATION

One of the driving motivations behind my research was that my approach provides enhanced access control policy expression capabilities. To demonstrate these capabilities, I constructed a short SaJE tool presentation for my thesis defense that showed the ease with which different types of access control policies may be interchanged on a single Java codebase without altering a single line of Java code.

Please note that while my technique enables flexible policy implementation and makes switching policies simple, there are other issues that must be addressed when changing access control policies. For example, what happens to an access-restricted fragment when its access group is removed or when the access group hierarchy changes. Code may suddenly not compile because of conflicting statements or users may find themselves without access to objects they need. These issues are beyond the scope of my research in this dissertation. However, special care and consideration must be given to access-restricted source code objects when changing policies to ensure that they transition correctly, securely, and that compositions of the objects do not prevent execution of the code.

The example code that I used contained a single Java main class and four helper classes. Each helper class printed a line specific to its class name. The main class called all of the helper classes. Access was initially restricted to each of the helper classes and their respective calling lines of code in the main class using one of the SAAC\textsubscript{0} access groups, not including Admin and Unrestricted (see Figure G.1). A Policy menu was added to SaJE to allow easy switching between policies, users, and to show access control policies represented as graphs (see Figure G.2). The Run menu was extended to allow the SaJE user to execute the code using the currently logged in user (Admin, Bob, or Sue) or to execute the code as either Bob or Sue (see Figure G.3). The SAAC\textsubscript{0} policy used for the demonstration is shown in Figure G.4. Note that Sue is a member of the Public access group and Bob is a member of the Employee access group.

During the demonstration, I changed the access control policy for the codebase to the SAAC\textsubscript{1}
Figure G.1: Demo codebase restricted using a SAAC\(_0\) policy.

Figure G.2: Policy menu for demo.

Figure G.3: Extended run menu for demo.
policy shown in Figure G.5. This policy shows Bob as an Employee if he logs in using a locally recognized IP address. If he is not using a locally recognized IP address then he becomes a member of the Private group. Logging in as Bob with a non-local IP address is shown in Figure G.6. By simply changing Bob's IP address and logging in again, Bob's access rights change to suit his new group membership (see Figure G.7). Each access control policy contains a fragment mapping function that is used to address the previously mentioned issue of subject/object rights transitions when policies change.

Lastly, I changed the access control policy for the codebase to the DAC policy shown in Figure G.8. This policy shows Bob and Sue as their own access entities. Because this policy removes the access groups for Employee and Remote Employee, the policy mapping function assigns objects previously restricted to those groups to the Admin access group. Figure G.9 shows an Administrator's view of the codebase. Bob's view of the codebase is shown in Figure G.10 and Sue's view of the code base is shown in Figure G.11. The DAC access control model allows for users to assign access rights to objects he/she owns, so Bob can allow Sue to access his fragments and files (see Figure G.12). Sue's view of the codebase once Bob grants her access is shown in Figure G.13. This image also shows the result of executing the code as Sue in the console pane at the bottom of the IDE.
Figure G.5: Demo SAAC_{1} policy.

Figure G.6: Bob logged in with a non-local IP address.

Figure G.7: Bob logged in with a local IP address.
Figure G.8: Demo DAC policy.

```java
package main;
import utilities.PublicStuff;
import utilities.PrivateStuff;
import utilities.EmployeeStuff;
import utilities.RemoteStuff;

public class AppMain {
    public AppMain() {
    }

    public static void main(String[] args) {
        PublicStuff.printSomething();
        PrivateStuff.printSomething();
        EmployeeStuff.printSomething();
        RemoteStuff.printSomething();
    }
}
```

Figure G.9: DAC Admin view of the demo codebase.

```java
package main;
import utilities.PublicStuff;

public class AppMain {
    public AppMain() {
    }

    public static void main(String[] args) {
        PublicStuff.printSomething();
    }
}
```

Figure G.10: Bob’s view of the DAC-restricted codebase.
Figure G.11: Sue’s initial view of the DAC-restricted codebase.

```java
package main;
import utilities.PrivateStuff;
public class AppMain {
    public AppMain() {
    }
    public static void main(String[] args) {
        PrivateStuff.printSomething();
    }
}
```

Figure G.12: Bob granting Sue access to his fragments.

Figure G.13: Sue’s revised view of the DAC-restricted codebase.
BIBLIOGRAPHY


[10] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code bub-


Courtney Comstock. 27-year old samarth agrawal to spend 3 years in prison for stealing socgen’s high frequency code. *Business Insider*, March 2011.


VITA

Dr. Mark A. Robinson is a Texas native and has lived in San Antonio since 1987. He has been a successful software engineer for over fifteen years. He could not make up his mind whether to work in industry or academia, so he decided to do both. He owns Fulgent Corporation, a software development firm, and enjoys educating tomorrow’s engineers as an adjunct professor. Mark received a Bachelor of Science with Honors from Trinity University and his Masters and Ph.D. in Computer Science from The University of Texas at San Antonio. He was awarded the Presidential Dissertation Fellowship in 2011. He is married to his lovely wife, Dawn, and has two fantastic children, Sarah and Jake.