ABSTRACT

HWANG, JEEHYUN. Improving the Quality of Security Policies. (Under the direction of Dr. Laurie A. Williams).

Systems such as web applications, database systems, and cloud services regulate users' access control to sensitive resources based on security policies. Organizations often manage security policies in an ad-hoc and inconsistent manner due to a lack of budget, resources, and staff. This management could cause crucial security problems such as unauthorized access to sensitive resources.

A security policy is a set of restrictions and properties that specify how a computing system prevents information and computing resources from being used in violation of an organization's security laws, rules, and practices. In computer systems, security policies are enforced to ensure correct functioning of access control such as "who" (e.g., authorized users or processes) can perform actions under "what" conditions.

Policy authors may follow common patterns in specifying and maintaining security policies. Researchers applied data mining techniques for deriving (implicit) patterns such as a group of users (i.e., roles in RBAC policies) who have the same access permissions. Policy authors reuse common patterns to reduce mistakes. Anomalies of those patterns are candidates for inspection to determine whether these anomalies expose faults.

Faults (i.e., misconfigurations) in security policies could result in tragic consequences, such as disallowing an authorized user to access her/his resources and allowing malicious users to access critical resources. Therefore, to improve the quality of security policies in

terms of policy correctness, policy authors must conduct rigorous testing and verification during testing and maintenance phases of software development process. However, manual test-input generation and verification is an error-prone, time-consuming, and tedious task.

In this dissertation, we propose approaches that help improve the quality of security policies automatically. *Our research goal is to help policy authors through automated pattern mining and testing techniques in the efficient detection and removal of faults.* This dissertation is comprised of three research projects where each project focuses on a specific software engineering task. The three research projects are as follows:

Pattern Mining. We present an approach to mine patterns from security policies used in open source software products. Our approach applies data mining techniques on policy evolution and specification data of those security policies to identify common patterns, which represent usage of security policies. Our approach uses mined patterns as policy specification rules and detect faults in security policies under analysis as deviations from the mined patterns..

Automated Test Generation. We present a systematic structural testing approach for security policies. Our approach is based on the concept of policy coverage, which helps test a policy's structural entities (i.e., rules, predicates, and clauses) to check whether each entity is specified correctly. Our approach analyzes security policies under test and generates test cases automatically to achieve high structural coverage. These test cases can achieve high fault-detection capability (i.e., detecting faults).

Automated Test Selection for Regression Testing. We present a safe-test-selection approach for regression testing of security policies. Among given initial test cases in access control systems under test, our approach selects and executes only test cases that could expose different policy behaviors across multiple versions of security policies. Our approach helps detect unexpected policy behaviors (i.e., regression faults) caused by policy changes efficiently.

These three research project have resulted in the following contributions:

- Patterns characterizing correlations of attributes in security policies help detect faults.
- Structural coverage for security policies is closely related to fault-detection capability. An original set of test cases with higher structural coverage often achieves higher fault-detection capability. Furthermore, its reduced set of test cases while maintaining the same structural coverage achieves similar fault-detection capability with the original set.
- Substantial number of test cases for regression testing can be reduced to help improve performance.

© Copyright 2014 JeeHyun Hwang

All Rights Reserved

Improving the Quality of Security Policies

by
JeeHyun Hwang

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2014

APPROVED BY:

_______________________________	_______________________________
Dr. Laurie A. Williams Committee Chair	Dr. Mladen A. Vouk
_______________________________	_______________________________
Dr. Gregory T. Byrd	Dr. Xuxian Jiang

UMI Number: 3584006

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI 3584006

Published by ProQuest LLC (2014). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

DEDICATION

To my wife, Da Young.

BIOGRAPHY

JeeHyun Hwang was born in Seoul, Korea. He completed his Master of Science degree in 2005 at Stony Brook University and his Bachelor of Science degree at Korea University, Seoul, Korea, in 2003, all in Computer Science. After Stony Brook, He attended North Carolina State University for his Doctoral work. His primary research interests are in the area of Software Engineering and Security with use of ideas from data analytics, testing, mining, and formal methods. His research goals are to develop techniques and tools that can help in improving software security and reliability efficiently. His industry internship experiences include Samsung Data Systems (Summer 2002), ABB (Summer 2007/2008), Fermi National Accelerator Laboratory (Summer 2009), and Cisco (Summer 2010). He is a member of NCSU's Realsearch group, led by Dr. Laurie Williams. He is also a student member of IEEE.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Laurie Williams, for her guidance, encouragement and financial support. Her outstanding professional advice helped me improve my research skills, technical writing, and effective presentations. I thank Dr. Mladen Vouk for his help and inspirational, valuable feedback on my research. I thank Dr. Gregory Byrd for finding errors and grammatical mistakes in my dissertation draft. I thank Dr. Xuxian Jiang for serving on my dissertation committee. I thank Dr. Tao Xie for his valuable advice and financial support over years.

I would like to thank my wife, Da Young, for supporting me with a big smile and prayers.

I would additionally like to thank my collaborators Dr. Alex Liu (Michigan State Univ.), Dr. Fei Chen (Michigan State Univ.), Dr. Vincent Hu (NIST), Dr. Mine Altunay (Fermilab), Donia Elkateb (Univ. of Luxembourg), and Dr. Tejeddine Mouelhi (Univ. of Luxembourg) for their help in publishing our research papers.

My deepest thanks goes to Chris, Eric, Jason, John, Maria, Pat, Patrick, and Rahul in the Realsearch group for their valuable comments and discussion on both research and life. Special thanks goes to Madhuri, Suresh, Kunal, Xusheng, Yoonki, Evan and Chris for sharing during my PhD journey.

I am thoroughly thankful for my family. My family Junghee Choi (mother), Hwa-Sook Hwang (father), Danam Hong (mother-in-law), Youngho Lee (father-in-law), Hyemin Hwang (sister), Yonho Son (brother-in-law), Injae Hwang (brother), Jiyeon Yoo (sister-in-law), Seeun Lee (sister-in-law), and Samuel Ahn (brother-in-law) were my constant source of prayers and encouragement throughout my PhD journey. Without their prayers and support, I would not have completed this dissertation.

I would like to thank God for giving me wisdom and guidance. Some trust in chariots and some in horses, but we trust in the name of the Lord our God (Psalm 20:7).

TABLE OF CONTENTS

LIST OF TABLES x

LIST OF FIGURES xi

GLOSSARY xiii

1 INTRODUCTION 1

1.1 CONTRIBUTIONS 5

1.2 SCOPE 7

1.3 DISSERTATION ORGANIZATION 8

2 BACKGROUND 9

2.1 ACCESS CONTROL POLICY 9

2.1.1 Concept 9

2.1.2 Access Control Policy Models 9

2.1.3 Access Control Processing 10

2.1.4 Example 12

2.2 FIREWALL POLICY 14

2.2.1 Concept 14

2.2.2 Firewall Policy Model 15

2.2.3 Example 16

3 RELATED WORK 19

3.1 MINING PATTERNS 19

3.2 ACP EVOLUTION AND SOFTWARE EVOLUTION 19

3.3 TESTING OF SECURITY POLICIES 21

3.4 REGRESSIONS TEST GENERATION 23

4 MINING PATTERNS VIA ASSOCIATION RULE MINING 25

4.1 INTRODUCTION 25

4.2 EXAMPLE 26

4.3 FAULT DETECTION WITH ASSOCIATION PATTERNS 28

4.4 EVALUATION 33

4.4.1 Research Questions and Metrics 33

4.4.2 Evaluation Setup 35

4.4.3 Evaluation Subjects 37

4.4.4 Results 38

4.5 CHAPTER SUMMARY 41

5 MINING PATTERNS THROUGH ANALYSIS OF HISTORICAL DATA 43

5.1 INTRODUCTION 43

5.2 EXAMPLE 46

5.3 RESEARCH METHODOLOGY 47

5.3.1 Data Collection 48

5.3.2 Metrics and Approach 51

5.4 RESULTS 56

5.4.1 Threats to Validity and Limitations 68

5.5 CHAPTER SUMMARY 69

6 SYSTEMATIC STRUCTURAL TESTING 70
6.1 INTRODUCTION 70
6.2 EXAMPLE 72
6.2.1 Definition 73
6.2.2 Structural Coverage 76
6.2.3 Structural Coverage and Fault Detection 78
6.3 APPROACH 78
6.3.1 Test Packet Generation 79
6.3.2 Test Reduction 86
6.3.3 Measuring Fault-Detection Capability 87
6.3.4 Implementation 88
6.4 EVALUATION 90
6.4.1 Instrumentation 92
6.4.2 Comparison of Structural Coverage 92
6.4.3 Comparison of Fault-Detection Capability 96
6.5 LIMITATION 100
6.6 CHAPTER SUMMARY 101
7 AUTOMATED REGRESSION TEST SELECTION FOR REGRESSION TESTING FOR SECURITY POLICY EVOLUTION 102
7.1 INTRODUCTION 102
7.2 EXAMPLE 104
7.3 REGRESSION TEST SELECTION APPROACH 104

7.3.1 Test Selection based on Mutation Analysis 105
7.3.2 Test Selection based on Coverage Analysis 106
7.3.3 Test Selection based on Recorded Request Evaluation 107
7.3.4 Safe Test-Selection Techniques 108
7.4 EVALUATION 108
7.5 CHAPTER SUMMARY 113
8 CONCLUSIONS AND FUTURE WORK 114
8.1 CONCLUSIONS 114
8.2 FUTURE WORK 115
REFERENCES 118

LIST OF TABLES

Table 2.1. Summary of ACP Models 11

Table 4.1. Policy Behavior with regards to *ExternalGrade*. 29

Table 4.2. Policy Behavior with regards to *InternalGrade*. 29

Table 4.3. Subjects used in our evaluation 37

Table 4.4. Fault-detection capability results of Change-Rule Decision (CRD) mutants 39

Table 5.1. Systems used in our study 49

Table 5.2. Result classification 55

Table 5.3. ACP evolution trends. 58

Table 5.4. System evolution trends 58

Table 5.5. ACP evolution patterns in our subject ACPs. 61

Table 5.6. Permissions in SELinux ACP 62

Table 5.7. Permissions in VCL ACP 64

Table 5.8. Permissions use in Snort ACP 65

Table 6.1. Summary of notations 74

Table 6.2. Mutation operators for policy mutation testing 89

Table 6.3. Experimental results on firewall policies. 93

Table 7.1. The number of selected test cases on average for each policy group. 111

Table 7.2. Elapsed time (millisecond) for each test-selection technique. 113

LIST OF FIGURES

Figure 2.1. An Example XACML Policy 13

Figure 2.2. An example firewall policy. 17

Figure 4.1. An example policy 28

Figure 4.2. Fault-detection ratios of faulty policies for each policy, each fault type, and each technique/approach 40

Figure 5.1. The number of policy lines (Bottom) and system LOC (Top) for SELinux (Left), VCL (Middle), and Snort (Right) 58

Figure 5.2. SELiux ACP evolution prediction precision, recall, and F-measure by choosing the most likely first 1, first 2, …, first 9 states based on ranking by SELinux ACP 67

Figure 5.3. Snort ACP evolution prediction precision, recall, and F-measure by choosing the most likely first 1, first 2, …, first 9 states based on ranking 67

Figure 6.1. Example firewall rules. 76

Figure 6.2. Sample packets for all combinations of true and false values of predicates p_1 and p_2 77

Figure 6.3. Sample packets for all combinations of true and false values of clauses c_1 and c_2. 77

Figure 6.4. Framework overview. 80

Figure 6.5. Rule coverage achieved by each packet set. 95

Figure 6.6. Predicate coverage achieved by each packet set. 97

Figure 6.7. Clause coverage achieved by each packet set. 97

Figure 6.8. Mutant-killing ratios for all operators by subjects. 99

Figure 6.9. Mutant-killing ratios for all subjects by operators. 100

Figure 7.1. An example policy specified in XACML. 105

Figure 7.2. An example mutant policy by changing the first rule's decision (i.e., effect). ...107

Figure 7.3. LMS1 (LMS2), VMS1 (VMS2), and ASMS1 (ASMS2) show test-reduction percentages for our subjects with modified policies, respectively, using TS_M and TS_C (TS_R). Y-axis denotes the percentage of test reduction. X-axis denotes the number of policy changes on our subjects. 112

GLOSSARY

- access control. access control ensures that resources are only granted to those users who are entitled to them.
- Access Control Policy (ACP). An ACP consists of a set of rules, each of which describes permission of accesses to resources by specified users or processes.
- Clause Coverage Criterion (CCC). CCC assures that a Boolean expression in each clause (e.g., Source IP address field) in a firewall policy to be evaluated to true and false at least once with packets. Because all decisions of the clauses are binary, McCabe cyclomatic complexity is $v(G) = c + 1$ where c is the number of the clauses. Therefore, the cyclomatic complexity of satisfying CCC is $c + 1$.
- firewall policy. A firewall consists of a set of rules that examine and filter packets passing in a network.
- McCabe cyclomatic complexity. McCabe's cyclomatic complexity is a quality metric that measures the number of linearly independent paths through the program. Cyclomatic complexity (i.e., McCabe number) is defined as $v(G) = e - n + 2$ where e and n are the number of edges and nodes in the control flow graph of the program. v refers to cyclomatic number and G indicates control flow graph. For simplified complexity calculation, if all decisions of predicates are binary, McCabe cyclomatic complexity is $v(G) = p + 1$ where p is the number of binary predicates.
- mutant. A mutant is a faulty policy that has been purposely altered from an original policy.
- mutation operator. A set of instructions for making a simple change to an original policy.

- mutation testing. A testing methodology in which two or more mutants are evaluated using the same test cases to evaluate the ability of the test cases to detect differences in the mutants.
- Predicate Coverage Criterion (PCC). PCC assures that a Boolean expression in each predicate of the rules in a firewall policy to be evaluated to true and false at least once with packets. Because all decisions of the predicates are binary, McCabe cyclomatic complexity is $v(G) = p + 1$ where p is the number of the predicates. Therefore, the cyclomatic complexity of satisfying PCC is $p + 1$.
- Rule Coverage Criterion (RCC). RCC assures that a Boolean expression in each predicate of the rules in a firewall policy to be evaluated to true at least once with packets. Because all decisions of the predicates of the rules are binary, McCabe cyclomatic complexity is $v(G) = p + 1$ where p is the number of the predicates. The cyclomatic complexity of satisfying RCC is p because we remove a case where all predicates are evaluated to be false from $V(G)$ based on the definition of RCC.
- Security Enhanced Linux (SELinux). SELinux provides the mechanism for supporting access control security policies in Linux.
- Security Policy. A security policy is a set of restrictions and properties that specify how a system prevents information and computing resources from being used in violation of an organization’s security laws, rules, and practices.
- Virtual Computing Lab (VCL). VCL provides cloud services such as reservations, management, or access (called checkout) to virtual machine images.

- OASIS eXtensible Access Control Markup Language (XACML). XACML is an XML-based policy specification language.

1 Introduction

Systems such as web applications, database systems, and cloud services regulate users' access to sensitive resources based on security policies (such as firewall policies and access control policies). Organizations often manage security policies in an ad-hoc and inconsistent manner due to a lack of budget, resources, and staff [5, 70]. As security policies are crucial elements in securing resources in organizations, ad-hoc and inconsistent management can create security problems (e.g., misconfigurations) such as unauthorized access to sensitive resources.

Wool [59] examined 37 network security policies (i.e., firewall policies) in production enterprise network in 2004. The security policies had between 5 and 2,671 rules. Wool reported that all of the security policies included at least one configuration error, which could allow unauthorized access.

NIST [67] defines *security policy* as "a set of restrictions and properties that specify how a computing system prevents information and computing resources from being used in violation of an organizational security laws, rules, and practices."

In computer systems, security policies specify "who" (e.g., users or processes) can perform actions under "what" conditions according to which access control must be regulated. A misconfiguration (i.e., fault) of security policies could cause severe damages to an organization, including financial and reputational losses [3].

Correctly specifying security policies is a critical and yet challenging task for building reliable security policies with three factors. First, the rules in a security policy are could be complex because of organization regulations and structure. Second, a security policy may consist of a large number of rules. For example, Wool [59] examined firewall policies with up to 2,671 rules in production enterprise network. Third, a security policy often consists of rules that are written by multiple policy authors, at different times, and for different reasons, which make maintaining security policies even more difficult [5]. Bauer et al. [5] interviewed 13 professional policy authors of large organizations. They reported that policy authors are concerned about potential mistakes due to security-policy management by multiple policy authors. Consider that a policy author may change security policies without notifications to a peer policy author. The peer policy authors may make mistakes in future due to a lack of information about this security policy change.

To help facilitating manage security policies correctly, policy authors may follow common patterns in specifying and maintaining security policies. However, these patterns may not be documented. For example, researchers applied data mining techniques [56] for deriving patterns such as a group of users (i.e., roles in RBAC policies) who have the same access permissions. Policy authors reuse common patterns and reduce mistakes. Anomalies of those patterns are candidates for inspection to determine whether these anomalies expose faults.

In addition, to improve the quality of security policies in terms of correctness, policy authors must conduct rigorous testing and verification during testing and maintenance phases of software development process. However, manual test-input generation and verification is an error-prone, time-consuming, and tedious task.

Our research goal is to help policy authors through automated pattern mining and testing techniques in the efficient detection and removal of faults.

To improve the quality of security policies in terms of correctness, researchers and practitioners have developed various policy analysis and testing tools. The main function of these policy analysis tools is to detect "bad smell" (i.e., "anomalies") in security policies based on some common patterns of configuration mistakes [4, 39]. The main drawback of these tools is that the "anomalies" may be false-positives and the number of "anomalies" could be too large to be practically useful. Several security policy testing techniques have been proposed [22, 37]. However, these security policy-testing techniques are not based on well-established testing techniques in software engineering. For example, these techniques do not consider coverage criteria [61] for security policy testing.

We focus on mining patterns and detecting faults effectively and efficiently in security policies under analysis. This dissertation is comprised of three research projects.

- **Pattern Mining**. We mined patterns [28] from security policies of open source software products.
    - In the first study, we applied association rule mining to mine patterns, called *patterns* characterizing correlations of policy behaviors with regards to attribute values. For example, in the security policy, called codeD [20], for a grading system, based on similar policy behaviors of a lecturer and a faculty member, our approach mines a property: if a lecturer is permitted to conduct actions (e.g., assign/modify) on grades, a faculty member is likely to be permitted to conduct the same actions on grades. Our approach gives alerts if the faculty member is permitted to conduct different actions (e.g., denied to modify) on grades.
    - In the second study, we mined patterns, called *evolution patterns,* which characterize common patterns of security policy evolution. We first empirically observe evolution trends of security policies by measuring growth (i.e., lines of code related to security policies) trends. We extract evolution patterns characterizing changes of permissions (i.e., rights to perform certain actions). A evolution pattern $st_1 \rightarrow st_2$ presents that st_1 (e.g., “read”) evolves into st_2 (e.g., “read” and “write”) indicating that policy authors add “write” permission in addition to existing “read” permission. Our approach helps specify security policies effectively and efficiently by recommending how to change policies based on evolution patterns.
- **Automated Test Generation**. We developed a systematic structural testing approach [26, 27]. Our approach is based on the concept of policy coverage, which helps test a

policy's structural entities (i.e., rules, predicates, and clauses) to check whether each entity is specified correctly. As manual test-packet generation is tedious, our approach is an automated test generation tool (that can generate test cases such as packets) for achieving high structural coverage. The reason for achieving high structural coverage is that a set of test cases with higher structural coverage (including rule, predicate, and clause coverage) investigates a large portion of policy entities for fault detection.

- **Regression Test Selection**. We developed a safe-test-selection approach [29] for regression testing of security policies. Our approach helps improve the quality of security policies by detecting unexpected policy behaviors (i.e., regression faults) caused by policy changes efficiently. With the change of security requirements, developers may modify security policies to comply with the requirements. After the modification, policy authors validate and verify the given system to determine that this modification is correct and do not introduce unexpected behaviors (i.e., regression faults). Among given initial test cases in access control systems under test, our approach selects and executes only test cases that could expose different policy behaviors across multiple versions of security policies.

1.1 Contributions

In summary, this dissertation makes the following contributions:

- Approach of Pattern Mining.

- We develop two approaches that mine patterns. The first one is to mine usage patterns via association mining approach. The second one is to mine evolution patterns by analyzing changes across multiple versions of security policies.
- We conduct mutation testing [41]. Mutation testing creates faulty versions of policies by making small syntactic and semantic changes. We first identify *patterns* of the faulty versions, and determine whether the anomalies of *patterns* expose faults (i.e., fault-detection) of the faulty versions. Our evaluation results show that these anomalies help detect faults.

- Approach of Automated Test Generation.
  - We develop a systematic structural testing approach based on the concept of policy coverage.
  - Our approach generates a set of test cases automatically based on well-established testing techniques in software engineering, such as an existing constraint solver.
  - We conduct mutation testing to measure fault-detection capability (i.e., detecting more injected faults). Our evaluation results show that a packet set with higher structural coverage has higher fault-detection capability
- Approach of Regression Test Selection.
  - We develop a test selection approach to select only test cases (from existing test cases) that reveal different policy behaviors due to policy changes.
  - Our approach uses three techniques; the first one is based on mutation analysis, the second one is based on coverage analysis, and the third one is based on evaluated decisions of requests issued from test cases.

    - Our evaluation results show that our test selection techniques achieve up to 51%~97% of test reduction for a modified version with given synthesized 5~25 policy changes for three Java programs.
- Empirical Study of Policy Evolution.
    - We conduct an empirical study of policy evolution for security policies (such as access control policies) of three open-source systems. We empirically observed that (1) the number of lines in policies continue to increase over time and (2) some of the evolution patterns appear to occur more frequently.

1.2 Scope

We aim to improve the quality of security policies such as access control policies and firewall policies. Both access control policies and firewall policies govern access controls. An access control policy selectively permits or denies certain users or processes to critical resources. Policy authors may use access control policies for various purposes such as authorization of users based on roles and filtering traffic on network interfaces. A firewall policy has only one purpose: it is used to examine traffic passing in a network and it makes decisions about whether these packets are allowed to pass.

In this dissertation, we evaluate our pattern mining and regression test selection approaches on access control policies. We evaluate our automated test generation on firewall policies.

- Our pattern mining approach mine patterns based on the observations that policy authors follow patterns in specifying and maintaining access control policies. However, firewall policies include many exception rules (such as whitelists and blacklists). Due to these exception rules, our approach could not be effective for detecting patterns for firewall policies.
- Our automated test generation approach generates test cases covering structure entities (such as rules) of firewall policies. This approach could generate test cases for an access control policy as well because an access control policy consists of a set of rules.
- Our regression test selection approach selects test cases by observing interactions between test cases and access control policies. Given firewall policies and test cases, this approach could select test cases that are impacted by firewall policy changes.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 provides background with respect to security policies such as access control policies and firewall policies. Chapter 3 presents related work. Chapter 4 presents our study to mine patterns via association rule mining. Chapter 5 presents our study to mine evolution patterns from security policies. Chapter 6 presents our systematic structural testing approach. Chapter 7 presents our test selection approach for regression testing of security policies. Chapter 8 summarizes our conclusions and future work.

2 Background

In this chapter, we discuss details of access control policies (ACPs) and firewall policies. We show examples of access control policies and firewall policies. An ACP typically governs access to critical resources in organizations. A firewall policy regulates access controls in inter or intra-networks by monitoring and filtering packets.

2.1 **Access Control Policy**

This sub-section provides background with regard to ACP concepts and terminology, access control architecture, and ACP models.

2.1.1 Concept

An ACP is a policy specification that defines "who" (e.g., users or processes) can perform actions under "what" conditions. An ACP typically consists of a set of rules, each of which describes permission or denial of accesses to resources by specified users or processes.

2.1.2 Access Control Policy Models

We present three popular ACP models: Role-Based Access Control (RBAC) [19, 50], Type Enforcement (TE) access control [52], and Network Access Control List (ACL) models [30]. ACP models are formal representations to express ACPs and their operations. For example, the RBAC model allows policy authors to express explicit ACPs based on role(s) (e.g., user groups) of a user instead of individual users. The benefit of the RBAC model is the reduction of the management costs by specifying ACPs based on roles instead of individual

users. Specifying access control of each individual user is tedious due to the increasing number of users.

Table 2.1 describes a summary of each ACP model. ACP models provide a means of fine-grained access control over given subjects in a system. Different ACP models formalize different entities and their relations to describe which subjects can take an action on objects. Typically, subjects are users or processes that use the system. Objects are resources to be protected from unauthorized access. Actions are the operations that subjects can perform (e.g., write or read) on the objects.

2.1.3 Access Control Processing

To facilitate ACP management, security mechanisms are designed to abstract and externalize ACPs to be a separate component. The benefit of this design is to reduce the management costs because policy authors can modify ACPs without changing functionality (e.g., business logic) in a system. The processing of access to critical resources in a system is as follows.

At an abstract level, program code interacts with ACPs. Program code includes security checks, called Policy Enforcement Points (PEPs), to check whether a given subject can have access to protected information. The PEPs formulate and send an access request to a security component, called Policy Decision Point (PDP) loaded with policies. The PDP evaluates the

Table 2.1. Summary of ACP Models

ACP Models	Subjects	Objects	Description
RBAC Model	Roles	Resources (e.g., sensitive information within an organization)	RBAC model is used to specify explicit access controls based on the role(s) of a subject. In organizations, a user typically can be assigned to one or more roles associated with permissions.
TE Model	System processes	Files, sockets, directories, etc.	TE model supports fine-grained control over processes and objects in operating systems. TE model defines a type of every process and object. TE model is used to specify which processes (grouped by subject types) can take an action on which objects (grouped by object types).
Network ACL model	Packet	System resources	Network ACL model controls access through network traffic by monitoring content in packet payload to detect malicious network traffic. The content typically includes source/destination IP addresses, source/destination port numbers, and protocol.

request against the policies and determines whether the request should be permitted or denied. Finally, the PDP sends the decision back to the PEPs to proceed.

Policy authors can specify and maintain ACPs in one place (in the form of a piece of code, configuration, dataset, or specification). Policy authors specify ACPs in various ways. One

common way is that policy authors specify ACPs using policy specification languages such as eXtensible Access Control Markup Language (XACML) [68] and Policy Description Language (PDL) [69]. Another common way is that policy authors use configuration files or relational databases to store ACPs.

2.1.4 Example

We illustrate an example ACP specified in XACML. XACML has become the de facto standard for specifying ACPs. Typically, XACML policies are specified separately from actual functionality (i.e., business logic) in program code.

An XACML policy consists of a policy set, which further consists of policy sets and policies. A policy consists of a sequence of rules, each of which specifies under what conditions C subject S is allowed or denied to perform action A (e.g., read) on certain object (i.e., resources) O in a given system.

More than one rule in an ACP may be applicable to a given request. A combining algorithm is used to combine multiple decisions into a single decision. There are four standard combining algorithms. The *deny-overrides* algorithm returns Deny if any rule evaluation returns Deny or no rule is applicable. The *permit-overrides* algorithm returns Permit if any rule evaluation returns Permit. Otherwise, the algorithm returns Deny.

```
1<Policy PolicyId="univ" RuleCombAlgId="first-applicable">
2       <Target>
3               <Subjects> <AnySubjects/> </Subjects>
4               <Resources> <AnyResource/> </Resources>
5               <Actions> <AnyAction/> </Actions>
6       </Target>
7       <Rule RuleId="1" Effect="Permit">
8               <Target>
9                       <Subjects><Subject> Faculty </Subject></Subjects>
10                      <Resources>
11                              <Resource> ExternalGrades </Resource>
12                              <Resource> InternalGrades </Resource>
13                      </Resources>
14                      <Actions><Action> View </Action>
15                              <Action> Write </Action></Actions>
16              </Target></Rule>
17      <Rule RuleId="2" Effect="Permit">
18              <Target>
19                      <Subjects><Subject> Student </Subject></Subjects>
20                      <Resources>
21                              <Resource> ExternalGrades </Resource>
22                      </Resources>
23                      <Actions><Action> View </Action></Actions>
24              </Target>
25      </Rule>
26      <Rule RuleId="3" Effect="Deny">
27              <Target>
28                      <Subjects><Subject> Student </Subject></Subjects>
29                      <Resources>
30                      <Resource> ExternalGrades </Resource>
31                      </Resources>
32                      <Actions><Action> Write </Action></Actions>
33              </Target>
```

Figure 2.1. An Example XACML Policy

The *first-applicable* algorithm returns what the evaluation of the first applicable rule returns. The *only-one-applicable* algorithm returns the decision of the only applicable rule if

there is only one applicable rule, and returns error otherwise.

Figure 2.1 shows an example XACML policy adapted from a sample policy used by Fisler et al. [20]. This example illustrates a policy that uses the first-applicable algorithm, which determines to return the evaluated decision of the first applicable rule. In this example, there are two subjects or roles (*Faculty*, *Student*), two resources (*ExternalGrades*, *InternalGrades*), and two actions (*View*, *Write*).

There are four rules in the XACML policy. Lines 7-16 define the first (permit) rule, which allows a faculty to view or write external or internal grades. Lines 17-25 define the second (permit) rule, which allows a student to view external grades. Lines 26-34 define the third (deny) rule, which denies a student to write external grades. Line 36 defines the last default (deny) rule, which denies any request that does not match any of the three preceding rules.

2.2 **Firewall Policy**

We next provide background of firewall policies. A firewall is typically placed at the point of entry between a private network and the outside Internet such that firewalls are responsible for filtering, monitoring, and securing packets [38].

2.2.1 Concept

A firewall policy is composed of a sequence of rules that specify under what conditions a packet is accepted or discarded while passing between a private network and the outside

Internet. In other words, the policy describes a sequence of rules to decide whether packets are accepted (i.e., being legitimate) or discarded (i.e., being illegitimate). A rule is composed of a set of fields (generally including source/destination IP addresses, source/destination port numbers, and protocol type) and a decision. Each field represents the range of possible values (to match the corresponding value of a packet), which are either a single value or a finite interval of non-negative integers.

A packet matches a rule if and only if each value of the packet satisfies the corresponding values in the rule. Upon finding a matching rule, the corresponding decision of that rule is derived. When evaluating a packet, the firewall policy follows the first-match semantic: the first matching rule is given the highest priority among all the matching rules.

2.2.2 Firewall Policy Model

This section illustrates a model of a firewall policy based on common generic features. A firewall policy is composed of a sequence of rules, each of which has the form (called the generic representation) as follows.

$$\langle predicate \rangle \rightarrow \langle decision \rangle$$

A <*predicate*> in a rule is a boolean expression over fields on which a packet arrives. The <*decision*> of a rule can be “accept” or “discard”; it is returned as the evaluation result when

the <*predicate*> is evaluated to be true. The <*predicate*> is represented as a conjunction form as follows.

$$F_1 \in S_1 \land \ldots \land F_n \in S_n$$

In a policy model, we represent a value in a field F_i (e.g., IP address) as its corresponding range S_i (e.g., $F_i \in [2,5]$) to simplify the representation format. We refer to each $F_i \in S_i$ as a <*clause*>, which can be evaluated to either true or false.

The first-match semantic (of a firewall policy) shows the same behavior with the execution of a series of IF-THEN-ELSE statements in program code. Given a sequence of rules, the following process is iterated until reaching the last rule: if a <*predicate*> in a rule is evaluated true, then the corresponding decision is returned; otherwise, the next rule (if exists) is evaluated.

2.2.3 Example

Figure 2.2 shows an example of a firewall policy. The symbol "*" denotes that the corresponding field's range (in a rule) is equal to the domain of the field and is satisfied by any packet. An IP address is a 32 bit value, which is represented as a four-part dotted-decimal address (e.g., 192.168.0.0). Classless Inter-domain Routing (CIDR) notation is used to represent IP ranges over an IP address with a subnet mask (e.g., /16 or /24). For example, the range of 192.168.0.0/24 implies IP addresses from 192.168.0.0 to 192.168.0.255. This

Rule	Source IP	Source Port	Destination IP	Destination Port	Protocol	Decision
r_1	*	*	192.168.0.0/16	*	*	accept
r_2	1.2.3.0/24	*	*	$[1, 2^8 - 1]$	TCP	*discard*
r_3	*	*	*	*	*	*discard*

Figure 2.2. An example firewall policy.

range consists of all possible IP addresses starting with the same left-most 24 bits (i.e., 192.168.0) on the given IP address. Each of the remaining 8 bits (which do not have fixed values) is either 0 or 1.

The example has three firewall rules r_1, r_2, and r_3. Rule r_1 accepts any packet whose destination IP address is the network 192.168.0.0/16 (which indicates the range [192.168.0.0, 192.168.255.255]). Rule r_2 discards any packet whose source IP address is the network 1.2.3.0/24 (which indicates the range [1.2.3.0, 1.2.3.255]) and port is the range $[1, 2^8 - 1]$ with the TCP protocol type. Rule r_3 is a tautology rule to discard all packets. Consider a packet whose destination IP address is 192.168.0.0 and protocol type is UDP. When evaluating the packet, we find that the packet can match both r_1 and r_3. Between the two rules, as r_1 is the first-matching rule, the packet is evaluated to be accepted (with regards to the decision of r_1). If a packet matches no rules in a firewall policy, there exists the last tautology rule to discard the packet.

Both ACPs and firewall policies govern access. ACPs selectively permit or deny certain users or processes to critical resources. Policy authors may use ACPs for various purposes such as authorization of users based on roles and filtering traffic on network interfaces. ACPs

are typically stateless. However, a firewall has only one purpose: a firewall is a device which examines traffic passing in a network and makes decisions whether these packets are allowed to pass or not.

3 Related Work

Our work builds on prior work in four areas: pattern mining, software evolution, policy testing, and regression test generation. In this chapter, we provide relate work in these area.

3.1 Mining Patterns

Martin et al. [40, 42] developed an approach for measuring the quality of policy properties in policy verification. Given user-specified properties, they developed an approach that measures the quality of the properties based on fault-detection capability. In addition, they developed an approach to use machine-learning algorithms (e.g., a classification algorithm) to mine policy properties automatically. Given request-decision pairs, this previous approach mines request-classification rules based on a statistical policy-behavior model. Therefore, faults are likely to be detected when the policy violates this model. Bauer et al.'s approach [6] proposed an approach to mine association rules, which are used to detect misconfiguration in a policy. Their approach considers only object attributes for mining patterns from historical access data.

3.2 ACP Evolution and Software Evolution

The closest research relating to our work is an empirical study on permission evolution/usage in the Android platform conducted by Wei et al. [57]. They used multiple Android platform releases and application versions. They reported that a list of permissions and usage for Android platforms and applications is growing over time. With the increasing

number of permissions, the number of dangerous permissions (e.g., personal data-related resource access privileges) increases over time. Our evolution study differs in the following ways. First, our empirical study relied on ACPs specified by policy authors for various systems such as operating and database systems. The collected ACPs are specified based on popular ACP models such as RBAC and TE policy models, which allow policy authors to add/delete subjects. However, Wei et al. studied permissions specific for Android application, such as permission on GPS location access of users. Android application developers are allowed to choose necessary permissions within the set of pre-defined permissions. They cannot add/delete subjects. Additionally, Wei et al. studied permission evolution based on concerns and behaviors related to the least privilege property and dangerous/secure permissions. Our study focuses on understanding why and how ACPs evolve in general over time.

Chia et al. [11] conducted a characterization study on user-consent permission systems in Facebook, Chrome, and Android applications. These permissions are given to applications based on user consent for granting explicit permissions upon the request of applications. They found that community and user ratings on applications' privacy were not reliable for determining privacy risks of the applications. They reported that free applications request more permissions than those necessary for the application. Their study focuses on permission characterization and effectiveness of single-release applications. In contrast, our evolution study focuses on evolution of ACP by analyzing multiple versions of ACPs.

Another research direction is the formalism of ACP evolution. Koch et al. [32] have proposed a model to formalize ACP and its evolution using graph transformations. Their formalism helps describe ACP evolution precisely. Pretschner et al. [46] have proposed the approach of an evolution model with regards to ACP usage control. However, their work focuses on a theoretical ACP evolution model without any empirical observations on ACPs in practice. Different from their study, our evolution study focuses how ACPs evolve in practice. Moreover, we propose a model to help predict how ACPs evolve.

Software evolution is a very active research area in software engineering. Kemerer et al. [31] conducted an empirical study to understand characteristics of software evolution and developed taxonomy of software maintenance. They studied the historical growth and changes of 23 software applications over 20 years. They categorize maintenance causes of software evolution. Buckley et al. [9] proposed taxonomies of software evolution based on characterizing the mechanisms of change. This taxonomy helps identify and evaluate tools, methods and formalisms for a given software change. They conducted an empirical approach to understand various aspects such as driving factors, impact, taxonomy and processes in software evolution. In this dissertation, different from general software evolution, we focus on the evolution of ACPs.

3.3 Testing of Security Policies

A firewall policy is translated to program code (i.e., IF-THEN-ELSE statements) that includes a large number of conjunctive logical expressions to illustrate rules. Ammann et al

[2] proposed coverage criteria for such logical expressions. For example, they proposed predicate and clause coverage criteria in notions of logical expressions. Although they proposed such criteria, they did not generate test suites for real program code to show the effectiveness of their coverage criteria. We propose logical coverage criteria that are suitable for a firewall policy. We also develop test packet generation and mutation testing techniques to show the effectiveness in terms of fault-detection capability. Our work targets test generation and mutation testing especially for a large number of logical expressions (in a firewall policy).

Black et al. [7] and Wimmel et al. [58] proposed mutation testing for specifications. However, their mutation operators change operators (e.g., replacing an expression by its negation) and pre/post conditions of specifications. In our automated test-generation approach, instead of changing operators and pre/post conditions, we mutate clauses and a rule's decision, where policy authors could make mistakes in specifying rules (e.g., specifying incorrect values). For testing access control policies such as XACML policies [68], Martin et al. [43] proposed to mutate policies [41], and generate random requests automatically. Their proposed structural coverage criteria and mutation operators are not directly applicable to firewall policies due to the semantic and syntactic differences between access control policies and firewall policies. While firewall policies consist of a set of ranges (intervals) in rules, access control policies consist of structural elements such as policies, rules, subjects, objects, and actions. They do not use a well-established test generation technique to cover certain entities.

Some researchers proposed firewall testing with test cases generated based on their proposed criteria. Jürjens et al. [30] proposed specification based testing, which generated test sequences to cover a state transition model of a firewall and its surrounding network. El-Atawy et al. [17] proposed policy criteria identified by interactions between rules, called "policy segmentation" identified by interactions between rules. Different from their approaches, we use structural coverage criteria in each rule to help detect which entities are specified incorrectly. In addition, we also use mutation testing to evaluate our automated test-generation.

Several firewall policy testing techniques [22, 37] inject packets into a firewall and check whether the decisions of the firewall concerning the injected packets are correct. However, these techniques lack rigorousness in terms of the use of coverage criteria and effective mechanisms for generating covering packets. Furthermore, these testing techniques are inefficient when a tester needs to inject a large number of packets and examine their decisions. In contrast, our automated test-generation approach is based on solid foundations and advanced test-packet generation techniques.

3.4 Regressions Test Generation

Various techniques have been proposed on regression testing of software programs [16, 21, 48]. These techniques aim to select test cases that could reveal different behaviors after modification in programs. These techniques are related to regression-test selection [21, 48], and test-suite prioritization [16]. Note that these techniques focus on changes at code level.

None of these techniques consider potential changes that can arise from code-related components (such as security policies specified separately). Polices and general programs are fundamentally different in terms of structures, semantics, and functionalities, etc. Therefore, techniques for regression testing of programs are not suitable for addressing the test-selection problem for policy evolution. Our regression-test-selection approach is the automatic test-selection approach for policy evolution.

Fisler et al.'s approach [20] developed a tool called Margrave that enabled conducting change-impact analysis between two XACML policies. We could use Margrave to identify semantic policy changes between two policies. However, Margrave supported only limited functionality of XACML. Moreover, Margrave did not support test selection as our work does.

4 Mining Patterns via Association Rule Mining

4.1 Introduction

Systems adopt access control mechanisms to offer access control to sensitive resources based on ACPs. An ACP consists of a set of rules, each of which describes permission of accesses to resources by specified users or processes. Identifying discrepancies between ACPs and their expected functions is crucial because these discrepancies may result in unexpected access controls such as allowing malicious users to access sensitive resources. To increase our confidence on the correctness of ACPs, ACPs must undergo rigorous verification and testing.

To help improve the quality of ACPs in terms of correctness, we develop an approach to mine patterns (in ACPs) that policy authors often follow implicitly. The reason for identifying these patterns is based on observations that the policy authors often follow these patterns in specifying and maintaining ACPs. Anomalies of those patterns should be inspected to determine whether these anomalies expose faults.

More specifically, we apply association-rule-mining [8] to mine patterns from subject, object, and action attribute values used in ACPs. Association rule mining searches patterns, which are in the form (A, D) ⇒ (B, D) where A and B are sets of attribute values and D is either a "Permit" or "Deny" decision. The form represents that (A, D) implies (B, D) where (A, D) is <premise> and (B, D) is <conclusion>. In other words, if <premise> holds true,

<conclusion> is likely to hold true. Consider that o_1, o_2, o_3, and o_4 are object attribute values (e.g., file name) in an ACP. An example is (o_1, "**Permit**") ⇒ (o_2, "**Permit**"), which represents that a user with access to o_1 is likely to have access to o_2. Another example (o_3, "**Permit**") ⇒ (o_4, "**Deny**") represents that a user with access to o_3 is **<u>not</u>** likely to have access to o_4. From all possible patterns, we collect patterns with a pre-defined probability value or above.

We define policy behaviors as a set of all possible access requests and their corresponding access decisions (e.g., Permit or Deny). Our approach mines such patterns that may not be true for all the policy behaviors, but are true for most of the policy behaviors. Therefore, *patterns* may lead to a small number of anomalies. As these anomalies are deviations from the policy's normal behavior, these anomalies should be inspected to determine whether these anomalies expose faults.

4.2 Example

Figure 4.1 illustrates an example policy [20] for a grading system in a university as if-else statements in code. This example is the RBAC policy used by Fisher et al. [20]. We next explain this example policy. The policy includes six rules. Lines 1-3 include rules that allow a faculty member to assign or modify *ExternalGrade* or *InternalGrade*. Lines 4-6 include rules that allow a Teaching Assistant (TA) to assign or receive *InternalGrade*. Lines 7-9 include rules that allow a student to receive *ExternalGrade*. Lines 10-12 include rules that allow a family member to receive *ExternalGrade*. Lines 13-15 include rules that allow a

lecturer to assign or modify *ExternalGrade* or *InternalGrade*. Line 16 is a tautology rule to deny requests that are not applicable in the preceding rules.

We next describe an example *pattern* related to the two action attribute values of “Modify” and “Receive”. Table 4.1 and Table 4.2 describe access decisions (i.e., “P” as Permit or “D” as Deny) associated with object attribute values, *ExternalGrade* and *InternalGrade*, respectively. In these tables, column 1 shows all possible subject (role) attribute values. Columns 2-3 describe corresponding access decisions of a subject attribute value associated with the action attribute value “Modify”, or “Receive”, respectively. For example, in the second row of Table 4.1, given a role attribute value (Faculty) and an object value (*ExternalGrade*), the table describes access decisions associated with action attribute values “View” and “Receive”. The corresponding decisions are “P” and “D.”

We consider an example pattern_1 as follows.

- $\text{Pattern}_{1:}$ If a subject (e.g., *Student*) is *Permitted* to Receive a grade (e.g., *ExternalGrade or InternalGrade*), the subject is *Denied* to Modify the grade.

In Table 4.1 and Table 4.2, column 4 describes whether Pattern_1 is satisfied for given attribute values. We found that three cases satisfy the <premise> of Pattern_1. The three cases are combinations of subject and object attribute values: (1) a student with *ExternalGrade* and (2) a family member with *ExternalGrade*, and (3) a TA with *InternalGrade*). All of these

1 If role = *Faculty*
2 and resource = (*ExternalGrade* or *InternalGrade*)
3 and action = (*Modify* or *Assign*) then Permit
4 If role = *TA*
5 and resource = (*InternalGrade*)
6 and action = (Assign or Receive) then Permit
7 If role = Student
8 and resource = (*ExternalGrade*)
9 and action = (*Receive*) then Permit
10 If role = *Family*
11 and resource = (*ExternalGrade*)
12 and action = (*Receive*) then Permit
13 If role = *Lecturer*
14 and resource = (*ExternalGrade* or *InternalGrade*))
15 and action = (*Assign* or *Modify*) then Permit
16 Deny

Figure 4.1. An example policy

cases satisfy the <conclusion> as well. Conditional probability, which measures the probability of satisfying <conclusion> given that <premise> is satisfied, is 100%.

4.3 Fault Detection with Association Patterns

This section first presents definitions for attribute values and relations that our approach is based on. This section next presents our approach for detecting faults in a policy with our pattern mining techniques. Our approach includes three steps: (1) generate relation-table (2) identify association patterns, and (3) prioritize anomalies.

- **Generate Relation-table.** The relation-table generation component takes a policy *p* as an input and generates tables based on attribute values in the policy *p*. The association rule

Table 4.1. Policy Behavior with regards to *ExternalGrade*.

	Modify	**Receive**	**$Pattern_1$**
Faculty	P	D	
TA	D	D	
Student	D	P	Yes
Family	D	P	Yes
Lecturer	P	D	

Table 4.2. Policy Behavior with regards to *InternalGrade*.

	Modify	**Receive**	**$Pattern_1$**
Faculty	P	D	
TA	D	P	Yes
Student	D	D	
Family	D	D	
Lecturer	P	D	

mining component takes attribute values (from the table produced by the previous component) and mines patterns *r*.

- **Identify Association Patterns.** The pattern identification component takes p and r as inputs and verifies *p* against *r*. The component produces verification reports based on whether the given patterns *p* are satisfied; when a property is violated, anomalies are generated accordingly.
- **Prioritize Anomalies.** The policy authors inspect anomalies to determine whether they expose faults. To detect faults effectively, we propose a prioritization technique to

recommend that the policy authors inspect anomalies by the order of their fault-detection likelihood.

4.3.1.1 Generate Relation-Table

Our approach first analyzes a policy *p* and generates a policy behavior report showing all possible request-response pairs in the policy *p* where a request is an access request and its response is its evaluation (Permit or Deny). Our approach next analyzes the policy behavior report, and then generates relation tables (including all request-response pairs) that can be used as input for an association rule-mining tool. For example, to mine *patterns*, we generate a relation table that organizes all possible policy behaviors. Based on this table, we generate our proposed *patterns* used to mine relations of attribute values. For example, Table 4.1 and Table 4.2. describe relation-table with regards to the two action attribute values of “Modify” and “Receive”.

4.3.1.2 Identify Association Patterns

Let $s \in S$, $o \in O$, and $a \in A$, respectively, denote the set of all the subject attribute values (e.g., user’s role or rank), objects (e.g., file) and actions (e.g., write or read) in an access control system.

In this dissertation, we propose three types of patterns based on subjects, actions, and subject-action attribute values, as presented next. Each of these relations focuses on mining relations of specific attribute values.

- **Subject relations (S-pattern type).** We denote this relation as {s_1, d_1} ⇒ {s_2, d_2} where s_1 and s_2 are subjects (i.e., $s_1 \in S$ and $s_2 \in S$). An example (s_1, "**Permit**") ⇒ (s_2, "**Permit**") represents that s_2 is likely to inherit all of the permissions from s_1.
- **Action relation (A-pattern type).** We denote this relation as {a_1, dec_1} ⇒ {a_2, dec_2} where a_1 and a_2 are actions (i.e., $a_1 \in A$ and $a_2 \in A$). An example (a_1, "**Permit**") ⇒ (a_2, "**Permit**") represents that given an object o, a user who is permitted to take an action a_1 on o is likely to be permitted to take an action a_2 on o.
- **Subject-Action relation (SA pattern type).** We denote this relation as {s_1, a_3, d_1} ⇒ {s_2, a, dec_2} where s_1 and s_2 are subjects, and a_3 is an action (i.e., $s_1 \in S$, $s_2 \in S$, and a $\in$ A). An example (s_1, a_3, "**Permit**") ⇒ (s_2, a_3,"**Permit**") represents that if s_1 is permitted to take a_3, s_2 is likely to be permitted to take a_3. These subject-action relations subsume subject and action relations.

In association rule mining, thresholds such as support and confidence are used to constrain generating association relations. Let *t* denote the total number of transactions corresponds to the number of rows in a relation table. For example, the sum of transactions in Table 4.1 and Table 4.2 includes 10 transactions. Let *d* denote the number of transactions including an attribute item X (that is attribute values and decision set). The **support** supp(X) of X is $\frac{d}{t}$. We measure **confidence**, which is the likelihood of a relation: confidence (X ⇒ Y) = $\frac{\text{supp}(X \cup Y)}{\text{supp}(X)}$. These relations are *patterns* if support values and confidence in the patterns are above pre-defined thresholds.

4.3.1.3 Prioritize Anomalies

Our approach next verifies the policy with the patterns to check whether the policy includes a fault. Our rationale is that, as patterns are true for most of the policy behaviors, anomalies (which do not satisfy the patterns) deviate from the policy's normal behaviors and should be inspected.

Basic and Prioritization Techniques. A basic technique is to inspect anomalies without any inspection order among the anomalies. Since the number of generated anomalies can be large, manual inspection of the anomalies can be tedious. To address the preceding issue, we propose a prioritization technique that classifies anomalies into a ranked order based on their fault-detection likelihood. The technique evaluates anomalies in each of the anomalies by the order of their fault-detection likelihood until a fault is detected. The prioritization technique maintains the same level of fault-detection capability of the basic technique when the policy contains a single fault.

We next describe how we classify anomalies into a ranked order CS_{du}, CS_1, ..., CS_n, based on their fault-detection likelihood. First, we give the highest priority to duplicate anomalies, which are classified to CS_{du}. When multiple patterns triangulate on a single rule, this rule may be more likely to contain a fault. Second, we investigate the number of anomalies produced by patterns to rank an order among anomalies. As a pattern may lead to anomalies, the policy authors are required to verify anomalies to ensure the correctness of a given policy. Given a property that has w anomalies, we classify these anomalies to CS_w ($1 \leq w \leq m$

where m is the largest number of anomalies generated for a pattern). The policy authors first inspect anomalies in CS_{du}. The policy authors then inspect anomalies in CS_i by the order of CS_1, ..., CS_m ($1 \leq i \leq m$) until a fault is detected.

4.4 Evaluation

We next describe the evaluation results to show the effectiveness of our approach with four policies. These policies are codeD, continue-a, continue-b, and univ policies (in Table 4.3).

4.4.1 Research Questions and Metrics

In our evaluation, we try to address the following research questions:

RQ1: In terms of the percentage of faults detected, how does our approach compare to a decision-tree-classification-based approach [42]?

RQ2: In terms of distinct anomalies, how does our approach compare to the decision-tree-classification-based approach [42]?

RQ3: For cases where a fault in a faulty policy (i.e., mutant) is detected by our approach, does percentage of distinct anomalies (for inspection) are reduced by our prioritization technique (in terms of detecting the first-detected fault) over our basic technique?

To measure fault-detection capability, we synthesize faulty policies, $f_1, f_2, ..., f_n$ by seeding faults into a subject policy f_o, with only one fault in each faulty policy.

Then, the chosen approach generates anomalies (i.e., counterexamples) for each faulty policy to detect the seeded fault. Note that we seed a single fault for f_i. For n faulty policies, n faults exist. Let CP (f_i) be distinct anomalies generated by the chosen approach for f_i. Let Count(f_i) be the number of distinct anomalies in CP (f_i) for f_i. Let DE (f_i) be the reduced number of distinct anomalies by the prioritization technique to detect the fault in f_i for cases where the fault in f_i is detected by our approach.

We next describe our metrics for the evaluation:

- **Fault-detection ratio (FR).** Let p be the number of True Positives (i.e., injected faults) detected by anomalies (generated by the chosen approach) for $f_1, f_2, ..., f_n$. The FR is $\frac{p}{n}$. The FR is measured to address RQ1.
- **Anomalies count (AP) for each policy.** This metrics is the number of anomalies generated by the chosen approach for each policy.
- **Anomalies count (AM) for our generated mutants.** This metric is the average number of distinct anomalies generated by the chosen approach for each faulty policy. The anomalies count is $\frac{\sum_{i=1}^{n} Count\ (fi)}{n}$. Note that an anomaly is synonymous to a

request. The AM is measured to address RQ2. The AM is used to define the ARB metric below.

- **Anomalies-reduction ratio (ARB) for our approach over the existing approach.** Let AM_1 and AM_2 be anomalies counts (AM_s) by our approach and the existing approach, respectively. The ARB is $\frac{\text{AM2} - \text{AM1}}{\text{AM2}}$. The ARB is measured to address RQ2.
- **Anomalies-reduction ratio (ARP) for the prioritization technique over the basic technique.** Let $f_{p1}, f_{p2}, \ldots, f_{pm}$ be faulty policies that are detected by our generated anomalies. The ARB is a percentage that measures the reduction ratio in terms of the number of the anomalies for inspection to detect the first fault by the prioritization technique over the basic technique. The ARP is $\frac{\sum_{i=1}^{n} Count\ (fpi) - \sum_{i=1}^{n} DE(fpi)}{\sum_{i=1}^{n} Count\ (fpi)}$. The ARP is measured to address RQ3.

4.4.2 Evaluation Setup

We use four fault types to automatically seed a policy with faults for synthesizing faulty policies (i.e., mutants), with only one fault in each policy for ease of evaluation: Change-Rule Decision (*CRD*), Rule-Target True (*RTT*), Rule-Target False (*RTF*), and Removal Rule (*RMR*).

- A *CRD* fault inverts a decision (e.g., change Permit to Deny) in a rule.
- An *RTT* fault indicates changing a rule to be applicable for any request.
- An *RTF* fault indicates changing a rule to be applicable for no request.

- An *RMR* fault indicates that a rule is missing.

We seed one fault to form each of mutants, i.e., each mutant includes only a single fault.

For the inspection for our approach, we use a verification tool [66] for XACML policies. Margrave also has a feature that statically analyzes an XACML policy and produces all possible request-decision pairs in a summarized format. Given a mutant, Margrave generates all possible request-decision pairs to be used for generating relation tables. We next mine relations from the relation tables using an association rule-mining tool [8].

4.4.2.1 Mining Patterns based on Decision Tree Classification

We compare the results of our approach with those of a previous related approach [42]. The related approach uses a decision-tree-classification approach to mine patterns. Let a decision tree (DT) denotes the related approach. Given request-decision pairs, DT learns policy behaviors and generates request-classification rules. Therefore, incorrectly classified requests (i.e., anomalies) deviate from normal policy behaviors, and are required to be inspected. We specify a confidence threshold as 0.4% based on our tuning of evaluation setup for DT to generate similar anomalies as our approach for the small sample of mutants used in the tuning of evaluation setup. In our evaluation, inspection of anomalies (to determine whether the anomalies expose faults) is automatically conducted by comparing the two decisions evaluated by a mutant and its corresponding original policy (that is assumed to

Table 4.3. Subjects used in our evaluation

Policy	# Rules	# Roles	# Actions	# Resources
codeD2	12	5	3	2
continue-a	298	5	5	26
continue-b	306	5	5	26
univ.	27	7	7	8

be correct). However, in general, this inspection is often a manual process conducted by the policy authors.

4.4.3 Evaluation Subjects

In our evaluation, we use four policies written in XACML [68]. XACML is an access control policy specification language. Table 4.3 summarizes the characteristics of each policy. Columns 1-5 show the evaluation subject name, the number of rules, and distinct attribute values in the subject, resource, and action attributes in the policy, respectively. A subject attribute corresponds a role attribute since the policies are based on the RBAC model [50]. We denote the number of roles, actions, and resources as # roles, # actions, and # resource, respectively.

The largest policy consists of 306 rules. The $codeD_2$ is a modified version of the codeD[11] by adding rules for a Lecturer role. For grading, a Lecturer role has the same privileges as a Faculty role. Two of the policies, namely continue-a and continue-b, are examples used by

1. http://www.cs.brown.edu/research/plt/software/margrave/versions/01-01/examples/college

Fisler et al. [20] to specify access control policies for a conference review system. The *univ* policy is an RBAC policy used by Stoller et al. [53].

4.4.4 Results

We conducted our evaluation on a laptop PC running Windows XP SP2 with 1G memory and dual 1.86GHz Intel Pentium processor. We also measure the total processing time of request-response-pair generation, pattern generation, anomalies generation, and automated inspection for correctness of given anomalies. For each mutant (with at most 306 rules), our results show that the total processing time is less than 10 seconds.

We first show our detailed evaluation results for only Change-Rule-Effect mutants. We then show our summarized evaluation results in Figure 4.2 for Rule-Target-True, Rule-Target-False, and Removal-Rule mutants. Table 4.4 summarizes the detailed results for Change-Rule-Decision (*CRD*) mutants of each policy. Columns 1-2 show the evaluation subject name and the number of *CRD* mutants. Columns 3-8 show fault-detection ratio (denoted as “%FR”), anomalies count for each policy (denoted as “#AP”), and anomalies count for our generated mutants (denoted as “#AM”) for DT approach and our approach, respectively. Columns 9-10 show ARB and ARP for our approach.

Results to address RQ1. In Table 4.4, we observe that DT and our approach detect 34.5% and 62.3% (in Column “% FR”) of CRD mutants, respectively. Let Basic and Prioritization denote our basic and prioritization techniques, respectively. Our approach

Table 4.4. Fault-detection capability results of Change-Rule Decision (CRD) mutants

		DT Approach			Our Approach				
Policy	#MT	%FR	#AP	#AM	%FR	#AP	#AM	%ARB	%ARP
code2D	12	66.6	3	3.5	83.3	0	2.4	31.4	58.3
continue-a	201	35.7	84	84.4	58.2	37	42.9	49.2	67.8
continue-b	209	35.7	84	84.4	55.9	37	42.8	49.3	68.0
univ.	27	0	26	26	51.8	8	8.5	67.3	84.7
AVERAGE	112.3	34.5	49.3	49.6	62.3	20.5	24.2	49.3	69.7

- FR: fault-detection ratio
- AP: the number of anomalies generated by the chosen approach for each policy
- AM: the average number of anomalies generated by the chosen approach for mutants
- ARB: anomalies reduction ratio for our approach over the existing approach [42]
- ARP: anomalies reduction ratio for the prioritization technique over the basic technique

(including *Basic* and *Prioritization* techniques) outperforms DT in terms of fault-detection capability. Our approach uses relations based on similar policy behaviors of different attributes values (e.g., Faculty and Lecturer). Therefore, if a faulty rule violates certain patterns of attribute items, our techniques have better fault-detection capability than that of DT. However, DT constructs classification rules based on the number of the same decisions without taking into how different attribute values interact. Therefore, these generated rules are rigid and often may easily miss certain correct policy behaviors.

Results to address RQ2. Our goal is to detect a fault with anomalies for inspection possible. Intuitively, with more anomalies to be inspected, fault-detection capability is likely to be improved. Our results show that our approach reduced the number of anomalies by 49.3% (in Column "% ARB") over DT. As a result, we observe that our approach

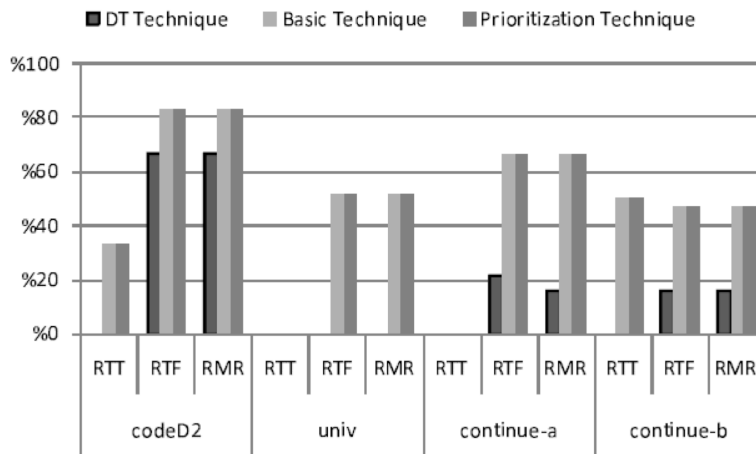

Figure 4.2. Fault-detection ratios of faulty policies for each policy, each fault type, and each technique/approach

significantly reduced the number of anomalies while our approach detected a higher percentage of faults (addressed in RQ1). Given N_s subject, N_a action, and N_r resource values, the maximum number MAX_c of possible anomalies is $N_s \times N_a \times N_r$. For example, for the continue-b policy, MAX_c is $5(N_s) \times 5(N_a) \times 26(N_r) = 650$ anomalies. However, our approach generated only averagely 24.2 anomalies (in Column "# AC") for inspection.

Results to address RQ3. *Prioritization* is a technique that inspects anomalies by the order of their fault-detection likelihood while keeping the same level of fault-detection capability of the *Basic* technique. Table 4.4 shows that *Prioritization* reduced 69.7% of anomalies (for inspection) (in Column "% ARP") over *Basic*.

Note that inspecting anomalies could not always detect faults. The continue-a policy consists of 298 rules and is complex enough to handle corner cases for granting correct

decisions to different roles (e.g., an Administrator and a Member for paper review). Consider that rel_3 {Write, Permit} ⇒ {Read, Permit} represents a pattern of "Write" and "Read" attribute values. For the continue-a policy (without any seeded fault), 41 cases satisfy both <premise> and <conclusion> of rel_3. However, we found that three cases (anomalies) that do not follow rel_3. One false positive is that Members are *Denied* to read their Password resources, while they are *Permitted* to write Password resources. Considering a Password resource as a critical resource and are *Denied* to be read, this anomaly does not reveal a fault in the policy. We suspect that inspecting these cases of policy behaviors would still provide value in gaining high confidence on the policy correctness, reflected by the preceding password example.

In addition, Figure 4.2 illustrates the average fault-detection ratios for each policy; each other fault type, and each technique/approach. For other fault types, our results show that *Prioritization* and *Basic* achieve the highest fault-detection capability.

4.5 **Chapter Summary**

We have developed an approach that analyzes a policy under verification and mines patterns based on relations of subject, action, and subject-action attributes values via association rule mining. We compared our two techniques in our approach with a previous related approach [42] in terms of fault-detection capabilities in four different XACML policies. Our results showed that our approach has more than 25% higher fault-detection capability than that of the previous related approach. Our results showed that our basic and

prioritization techniques reduce a significant percentage of anomalies for inspection compared to the related technique. Moreover, the prioritization technique further reduced a number of anomalies (for inspection) to detect a first fault over the basic technique.

5 Mining Patterns through Analysis of Historical Data

In this chapter, we present our empirical study of ACP evolution by providing observations about ACP evolution trends, practices, and patterns.

5.1 Introduction

As operational and security requirements of a system evolve, ACPs must evolve, thereby requiring maintenance. Policy authors may add permissions or remove unnecessary permissions. The US National Institute of Standards and Technology (NIST) recommends that policy authors periodically update ACPs by reviewing the current ACPs and access control procedures [71].

To facilitate the management of access controls, policy authors extract ACPs from the functionality (i.e., business logic) of a system and typically maintain ACPs in one place (in the form of a piece of code, configuration, dataset, or specification) [18]. Forrester Research [55] reported that such centralized ACPs improve not only security and privacy, but also the effective management of access controls. The report recommended that organizations use centralized ACPs instead of decentralized access controls that are implemented in custom code scattered across multiple locations.

In this chapter, our research goal is to assist policy authors to improve the quality of ACP evolution based on the understanding of trends, practices, and evolution patterns in ACPs through the mining of historical data.

To achieve this goal, we conducted an empirical study of the evolution of centralized ACPs by providing observations about ACP evolution trends, practices, and their accompanying evolution patterns. We conduct an empirical study of the ACPs of three large, open source systems (as shown in Table 5.1): Security Enhanced Linux (SELinux) [72]; a Virtual Computing Lab platform (VCL) [65]; and a network intrusion detection system [73] called Snort. These ACPs govern system resources, network traffic, and organizational operations on system usages. The ACPs of the three systems had between 2723 and 94,652 policy lines with between 238 and 5,489 policy changes over a 1-3 year period.

We first empirically observe growth trends of ACPs and corresponding systems in terms of the number of policy lines and lines of code (LOC), respectively. A well-fitting regression model is useful for predicting expected growth in the future. To assess "goodness of fit" of a regression model, we perform statistical testing of our observed growth against the linear regression model.

We then extract *evolution patterns* identifying how permissions (i.e., rights to perform certain actions such as read and write) change in rules. More specifically, to extract evolution patterns, we analyze differences of permissions of a subject (e.g., users) with respect to an object (e.g., sensitive resource). Let st_1 and st_2 are original permissions and modified permissions of a subject with respect to an object, respectively. An evolution pattern $st_1 \rightarrow st_2$ represents that st_1 evolves into st_2. For example, in addition to existing "read" permission for files, policy authors add "write" permission to allow the application to write files. In this

situation, an evolution pattern is st_1 = {read} → st_2 = {read, write}. Our rationale is that, as policy authors maintain ACPs through their lifecycles, they are likely to follow evolution patterns with higher frequency. These evolution patterns help understand which parts of ACPs are prone to evolve and how they evolve.

To evaluate the predictive power of our prediction model, we divide our evolution patterns into training and testing sets for cross validation.

Our study was designed to answer the following research questions:

RQ1. How do the number and growth rate of policy lines of ACPs change?

RQ2. How many of the rules in ACPs evolve?

RQ3. What are the frequent evolution patterns of ACPs?

RQ4. How effective is our model at predicting the change of ACPs?

Our empirical results will help practitioners develop tools (e.g., policy management and refactoring tools) to support these practices. In addition, our empirical results regarding evolution patterns are directed towards policy authors to better understand which parts of ACPs are prone to evolve and how they evolve.

5.2 **Example**

This section provides background with regard to ACP concepts and terminology, access control architecture, ACP models, and software change types.

ACP evolution refers to the specification and modification of ACPs. ACPs evolve due to various reasons such as the addition of new rules and ACP fault fixing. We introduce ACP and ACP evolution concepts through an example ACP from one of our systems, VCL. VCL provides cloud services such as reservations, management, or access (called checkout) to virtual machine images. The example ACP of a system is a set of rules. Each rule specifies access control of subjects based on roles such as virtual user groups (e.g., students enrolled in cloud computing course in Fall 2013). These roles are associated with specific permissions of operations (e.g., check out) to objects (e.g., virtual machine images).

The example ACP consists of two rules. We illustrate each rule to help understand its operation.

- Rule 1: user group s_1 is permitted to check out virtual machine images o_1.

Under the example ACP, s_1 can check out o_1. To specify specific access controls, policy authors add or modify rules. For example, the policy authors may modify the Rule 1 as follows:

- Rule 1^+: s_1 is permitted to check out and administer (denoted by "AdminImage") virtual machine images o_1.

Rule 1 evolves to Rule 1^+ with the addition of "administer" permission (e.g., administrative tasks). In addition to the "check out", s_1 can administer o_1.

Access Control Policy Rules (rules): ACP is a set of rules based on a concept of access control matrix. Formally, let O and S denote a set of objects (e.g., specific file name and directory name) and a set of subjects (e.g., users and processes), respectively. Let R denote a set of permissions (i.e., rights to perform certain operations such as read and write). Let D denote a set of domains of objects (e.g., file, directory, and virtual image). ACP is a set of rules, each of which has the form (s, o, r(s,o), d) where $s \in S$, $o \in O$, $r(s,o) \subseteq R$, and $d \in D$. In the example ACP, the policy authors add an additional permission to, Rule 1+. We represent the example ACP as a set of rules: Rule 1 is represented as (s_1, o_1, {*CheckOut*}, virtual image) and Rule 1^+ is represented as (s_1, o_1, {*CheckOut*, *AdminImage*}, virtual image).

5.3 **Research Methodology**

We next describe our collected data, methodology, and metrics for studying the evolution of ACPs.

5.3.1 Data Collection

We collected the ACPs of three large open source systems: Security Enhanced Linux (SELinux), a Virtual Computing Lab platform (VCL), and a network intrusion detection system, called Snort. Table 5.1 shows our subject statistics of ACPs. Columns 2-3 show the time frame we consider for each ACP and the number of revisions, respectively. Columns 4-7 show ACP size in terms of the number of files (tables) and policy lines (records), for the first and last releases within the time frame, respectively. As VCL uses ACPs that are stored in MySQL database instead of files, we measure ACP size in terms of tables and records (i.e., rows) related to ACPs. The last column shows the ACP model that each system uses.

We selected ACPs with the following criteria. First, because our focus is ACP evolution, the ACPs should have a long release history. Second, the ACPs should have significant size (i.e., policy lines). Third, the systems that use ACPs should be widely used open-source projects under active maintenance. Fourth, the open source projects archives contain a large number of software artifacts such as change logs, bug reports, documents, or patches for the ACPs. The criteria help conduct empirical study based on meaningful statistical trends of ACP evolution.

- SELinux ACP [72] is a default ACP of SELinux. SELinux ACP controls access to subsystems and applications in Security Enhanced Linux (SELinux). SELinux is a default security mechanism in various Linux distributions including Redhat (Version 6 and higher), Gentoo, Fedora, and Debian. SELinux ACP uses the TE policy model.

Table 5.1. Systems used in our study

ACP	Time Frame	# Rev	First Release		Last Release		ACP Model
			# Files/ Tables	Policy Lines	#Files/ Tables	Policy Lines	
SELinux ACP	2010~ 2013	5489	83	57550	116	94652	TE
ACP	2012~ 2013	896	1	4093	1	4989	RBAC
Snort ACP	2010~ 2013	238	57	2723	129	13394	ACL

SELinux ACP enhances flexible security and privacy for Linux systems. For example, policy authors specify permissions (e.g., open, read, search, and lock) for any directory. SELinux ACP consists of rules that are described using declaration of types (i.e., groups), inheritance of types, and interfaces (to facilitate specifying permissions of a type that are repeated in multiple times).

- VCL ACP [65] is a user-specified ACP of VCL maintained by Apache. VCL ACP specifies users, roles, roles' relation, and users' permissions (e.g., users' permissions to reserve, use, or administer virtual computing images in VCL) in database. For example, "userpriv" table includes records of user's permissions. VCL ACP uses the RBAC model.
- Snort ACP [73] is a default ACP of Snort that identifies potential intrusion attempts. Snort ACP uses the network ACL policy model. Snort inspects a packet against Snort ACP by examining incoming/outgoing addresses, ports, protocol types (e.g., TCP),

and contents of the packet. If an incoming/outgoing packet matches against Snort ACP, Snort gives alerts that the packet is suspicious for intrusion attempts.

We collected files and database records with regards to ACPs. SELinux ACP is freely-available via its version control system and websites such as Koij [64] that distribute SELinux ACP as a format of source-code and pre-compiled Linux packages. SELinux ACP is specified in its application-specific ACP specification format. We obtained change histories and patches of SELinux ACP from its version control system [72].

The VCL system and its sample ACP are freely available on the Apache VCL website. Organizations use VCL to provide cloud services (e.g., virtual computing, email and storage). We collected VCL ACP, used by faculty, staff, and students in one department. We examined "userpriv" and "querylog" tables: the "userpriv" table includes records of user's permissions, and the "querylog" table includes records of logs that store users and their permission changes.

Snort ACP is freely available via the Snort website. We obtained Snort ACP change histories from the Snort website. Policy authors typically update and release Snort ACP several times a week. Snort ACP is mainly written in application-specific configuration formats in text files. Snort ACP is a set of Snort rules. Each Snort rule includes its unique identifier called Snort ID number (SID).

In addition to ACPs, we collected releases of Linux (that includes SELinux), Snort, and VCL systems: 5 Linux kernel (versions 3.6-3.12 released Aug 2012-Sep 2013), 5 VCL systems (versions 2.1-2.3.2 released Oct 2010-Mar 2013), and 46 Snort systems (released Aug 1999-Sep 2013).

5.3.2 Metrics and Approach

To answer RQ1 (How do the number and growth rate of policy lines of ACPs change over time?), we measure the number and growth rate of policy lines of ACPs over time. We exclude comments and empty lines when we measure the number of policy lines. As VCL ACPs are stored in the MySQL database instead of files, we measure the number and growth rate of records (i.e., rows) in the "userpriv" table of VCL ACPs. To compare a system and its ACP in terms of growth rate, we measure the number and growth rate of lines of code (LOC) of systems over time.

To answer RQ2 (How many of the rules in ACPs evolve?), we measure two metrics: the number of rules (denoted by ACP_{sur}) that remain (either unchanged or evolved) and the number of evolved rules (denoted by ACP_{evo}) for a given period of time. ACP_{sur} helps understand how many of the rules remain after a given period of time. ACP_{evo} helps understand how many of the rules evolve (i.e., showing differences of permissions that granted to a subject with respect to an object) over a given period of time.

We formally represent ACP_{sur} and ACP_{evo}. Let ACP_o and ACP_l refer to the earliest and latest ACP release in the established time frame in Table 5.1, respectively. Let R_o (R_l) refer to a set of rules in ACP_o (ACP_l). Recall that *O* and *S* are a set of objects (e.g., specific file name and directory name) and a set of subjects (e.g., users and processes), respectively. *R* is a set of permissions (i.e., rights to perform certain operations such as read and write). *D* is a set of domains of objects (e.g., file, directory, and virtual image). ACP is a set of rules, each of which has the form *(s, o, r(s,o), d)* where $s \in S$, $o \in O$, $r(s,o) \subseteq R$, and $d \in D$.

ACP_{sur} is a set of rules $\{r_1, r_2, r_3, \ldots, r_n\} \subseteq R_o$ such that (1) $r_n = (s_n, o_n, r(s_n, o_n), d_n) \in R_o$ and (2) there exists $r_l = (s_l, o_l, r(s_l, o_l), d_l) \in R_l$ where $s_n = s_l$, $o_n = o_l$, and $d_n = d_l$.

ACP_{evo} is a set of rules $\{r_1, r_2, r_3, \ldots, r_n\} \subseteq Ro$ such that (1) $r_n = (s_n, o_n, r(s_n, o_n), d_n) \in R_o$ and (2) there exists rule $r_l = (s_l, o_l, r(s_l, o_l), d_l) \in R_l$ where $s_n = s_l$, $o_n = o_l$, $d_n = d_l$, and $r(s_n, o_n) \neq r(s_l, o_l)$.

We extract rules from an ACP by mapping between attributes in a given ACP and attributes in a rule. The mapping step for SELinux ACP (based on the TE policy model) and VCL ACP (based on the RBAC model) is straightforward. Both the TE policy model and the RBAC model use concepts of subjects S, objects O, and a set of permissions R as shown in Table 2.1. To construct a rule (s, o, r(s,o), d), we analyze an ACP and collect all of permissions r(s,o) for a pair of a subject s and an object o where $s \in S$, $o \in O$, and $r(s,o) \subseteq R$. Given an object, its corresponding object domain d is explicitly specified in SELinux ACP.

However, VCL ACP does not explicitly state object domains. Therefore, we use "virtual image" as a default object domain d for rules in VCL ACP.

Snort ACP (based on network ACL) uses attributes to indicate how Snort inspects certain parts of a packet. We use SIDs (Snort ID numbers) as subjects S, packet destinations as objects O, and attributes called rule option keywords as a set of permissions R. We use "packet" as a default object domain d for rules in Snort ACP.

To answer RQ3 (What are the evolution patterns of ACPs?), we find a set of evolution patterns $P = \{p_1, p_2, p_3, \ldots, p_n\}$ where p_n represents changes in terms of permissions. Recall from the example ACP of VCL (in Section 5.2), Rule 1 is that user group s_1 is permitted to check out virtual machine images o_1 in Rule 1. Rule 1^+ is that s_1 is permitted to check out and administer (denoted by "AdminImage") o_1. From the example ACP, we observe that evolution pattern $p_1 = st_1 \rightarrow st_2$ where st_1 = {check out} and st_2 = {check out, ImageAdmin}.

Formally, we define p_n as follows:

- State Pattern: p_n is $state_n \rightarrow state_l$ such that (1) $state_n = r(s_n, o_n)$ and $state_l = r(s_l, o_l)$, (2) $r_n = (s_n, o_n, r(s_n, o_n), d_n) \in R_o$ and (3) there exists rule $r_l = (s_l, o_l, r(s_l, o_l), d_l) \in R_l$ where $s_n = s_l$, $o_n = o_l$, $d_n = d_l$, and $r(s_n, o_n) \neq r(s_l, o_l)$.

- Transition Pattern: p_n is transition set that is $state_l$ - $state_n$ such that (1) $state_n = r(s_n, o_n)$ and $state_l = r(s_l, o_l)$, (2) $r_n = (s_n, o_n, r(s_n, o_n), d_n) \in R_o$ and (3) there exists rule $r_l = (s_l, o_l, r(s_l, o_l), d_l) \in R_l$ where $s_n = s_l$, $o_n = o_l$, $d_n = d_l$, and $r(s_n, o_n) \neq r(s_l, o_l)$.

A set of evolution patterns is viewed as a directed graph G= (V, E) where V is a set of nodes (i.e., states that represent a set of permissions) and E is a set of edges. Each element of E is a pair of nodes that represent a link between two nodes. Suppose that we have an additional evolution pattern $p_2 = st_2 \rightarrow st_3$ where st_3 = {check out, ImageAdmin, block}.

An edge represents a move from one state to another state. We calculate frequency and probability for each evolution pattern. Frequency f (P_n) is the number of occurrences of pn. Probability $Pr(P_n)$ is $\frac{f(pn)}{\sum_{i=1}^{\infty} f(pi)}$. For $p_1 = st_1 \rightarrow st_2$, we calculate $f(P_1) = 1$ and $Pr(P_1) = 0.5$.

To answer RQ4 (How effective is our model at predicting the change of ACPs?), we develop a prediction model of ACP evolution based on historical data of evolution patterns. Our rationale is that, as policy authors maintain and enhance ACPs through their lifecycles, they are likely to follow evolution patterns with higher frequency. Suppose that we found an evolution pattern $p_{ab} = A \rightarrow B$ that happened 20 times. Suppose that we found another evolution pattern $p_{ac} = A \rightarrow C$ that happened 5 times. Suppose that we find a state A that is to evolve. Based on our observations of evolution patterns in the past, A is likely to more likely to evolve to B (20 times) rather than C (5 times) based upon past occurrences. We calculate

Table 5.2. Result classification

	Predict next states	Not predict next states
Actually a current state evolves to be one of predicted states	True Positive	False Negative
Actually a current state does evolve to be one of predicted states	False Positive	True Negative

Pr (P_{ab}) = 0.8 (20/25) and Pr(P_{ac}) = 0.2 (5/25). We observe that the most likely state is B and the second most likely state is C.

Our prediction model generates an ordered rank of next states based on $P_r(P_n)$. Given a list of next states, our prediction model selects the most likely first 1, most likely first 3, …, most likely first 9 states based on ranking.

To measure the effectiveness of our prediction model, we classify our prediction results in four groups illustrated in Table 5.2: True Positive (TP), False Positive (FP), False Negative (FN), and True Negative (TN). We measure precision, recall, and F-measure: (1) $\text{precision} = \frac{\text{TP}}{\text{TP+FP}}$, (2) $\text{recall} = \frac{\text{TP}}{\text{TP+FN}}$, and (3) $\text{f} - \text{measure} = \frac{2*\text{precision}*\text{recall}}{\text{precision}+\text{recall}}$

If the precision and recall are good enough, our model is effective at predicting future trends of ACP evolutions.

For evaluating our prediction model, we classify evolution patterns into two sets: training and testing sets. For SELinux ACP, evolution patterns that happened for 2011/12/04-2013/01/02 are used as a training set while evolution patterns happened for 2013/01/02-2013/10/02 are used as a testing set. From Snort ACP, Snort rules with SID 1-15000 are used as a training set and Snort rules with SID 15001-16559 as a testing set.

5.4 **Results**

RQ1. How do the number and growth rate of policy lines of ACPs change?

Figure 5.1 shows that the number of policy lines (system LOC) of ACPs (systems) continues to increase over time. Our results show that the number of policy lines increases linearly. We observed that system LOC increases linearly as well. We assessed the statistical significance of growth of policy lines and system LOC using statistical testing methods against the linear regression model. We measured RSquare and p-value. The p-value [51] represents the probability of satisfying the linear regression model. For example, a test is statistically significant at 99% level if p-value = 0.01. The RSquare [51] are estimates of the 'goodness of fit' of the linear growth model. The RSquare values represent the variation of the data that fits to the linear growth model. For example, 90% of the variance is explained by the linear regression model if RSquare = 0.9.

Table 5.3 and Table 5.4 summarize our results according to the growth rate of policy lines of ACPs and system LOC over time, respectively. Columns 2-5 denote growth models, the

monthly increased number policy lines (system LOC) on average, RSquare, and p-value. In Table 5.3 and Table 5.4, we observed that our statistical testing results show statistical significance (p-value <= 0.0082 and RSquare >= 0.9) when we tested the growth rate of ACPs and system LOCs, respectively, against the linear regression model.

Policy authors maintain ACPs in response to evolving security and privacy requirements, such as the prevention of new security attacks. System developers actively maintained corresponding systems (with respect to LOC). The slope of the policy lines of ACPs is less than that of lines of code of their corresponding systems.

We measure monthly growth rates, which show the number of policy (lines of code) added each month for ACPs (systems). The monthly growth rates of ACPs on average are 886, 99, and 254 from SELinux, VCL, and Snort ACPs, respectively. The monthly growth rates of system LOC on average are 90870, 7739, and 1172 from SELinux, VCL, Snort systems, respectively.

In summary, the number of policy lines in ACPs continues to increase linearly over time. This result implies that ACPs are continually increased in response to changing/evolving security and privacy requirements. Furthermore, system developers maintain systems that ACPs rely on to achieve functionalities. We observed that the slope of the policy lines of ACPs is less than that of lines of code of their corresponding systems.

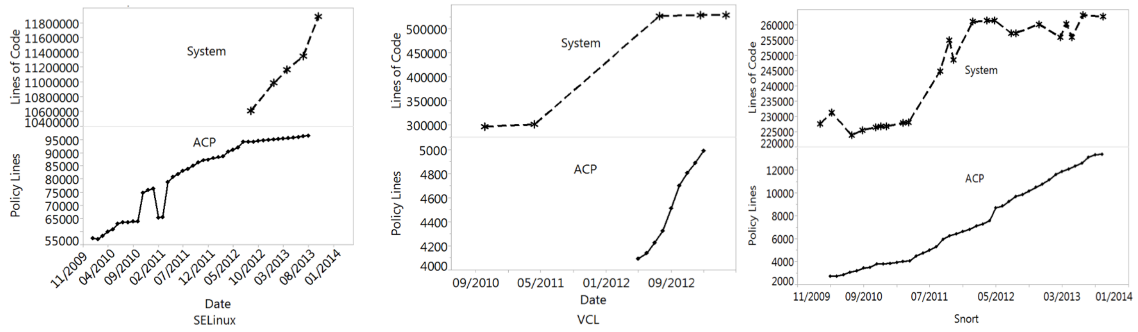

Figure 5.1. The number of policy lines (Bottom) and system LOC (Top) for SELinux (Left), VCL (Middle), and Snort (Right)

Table 5.3. ACP evolution trends.

ACPs	Growth Model	Avg	RSquare	p-value
SELinux ACP	Linear	886	0.91	0.0001*
VCL ACP	Linear	99	0.98	0.0001*
Snort ACP	Linear	254	0.97	0.0001*

Table 5.4. System evolution trends

Systems	Growth Model	Avg	RSquare	p-value
SELinux System	Linear	90870	0.95	0.0041*
VCL System	Linear	7739	0.92	0.0082*
Snort System	Linear	1172	0.9	0.0001*

RQ2. How many of the rules in ACPs evolve?

We found 83,304, 2,223, and 9,074 rules from SELinux ACP_o, VCL ACP_o, and Snort ACP_o, respectively. From these rules, we identified 57,848, 2,223, 7,677 rules that remain.

Among these rules, we identified 2,714, 7, and 2,019 rules that evolve. We noticed that only VCL ACP yielded a low number of ACP evolutions. We could find more evolved rules if we collect larger and longer historical data of VCL ACP. Rules in VCL ACP do not evolve often with the following reason. One reason is that policy authors identify desired permissions based on purposes of VCL user-groups' operations (e.g., administrator groups and user groups). Moreover, the number of permissions that are possible to use is small. Note that a VCL user/user-group could grant up to 14 permissions. Among these 14 permissions, policy authors can find which permissions should be granted for a user/user-group based on roles in advance.

In summary, we can find a large number of evolved rules from ACPs. We identified 2,714, and 2,019 rules that evolved from SELinux ACP and Snort ACP. This result implies that policy authors often modify existing rules in ACPs by adding permissions or removing unnecessary permissions.

RQ3. What are the frequent evolution patterns of ACPs?

We analyze ACPs and collect frequent evolution patterns. We collected frequent evolution patterns of which probability (defined in Section 5.3.2) is above a threshold value. Our threshold value is the percentage of occurrences of a certain evolution pattern out of a total number of evolution occurrences. The total number of frequent patterns depends on a

threshold. Given a pattern and a threshold, when its probability exceeds the threshold, we identify the pattern as a frequent pattern. We use 2% as a threshold.

Table 5.5 summarizes our results of ACP evolution patterns. Columns 2-5 the number of frequent evolution patterns FP ("# Frequent Patterns"), the percentage covering 86.2%, 100%, and 38.6% of evolved rules with FP (% Coverage"), the sum of frequencies of FP ("# Sum of Frequency"), and the highest frequency and probability of the most frequent pattern ("# Highest Frequency").

SELinux ACP specifies resources classified in one of object domains such as directory, file, sockets, and processes. We collect patterns from SELinux ACP changes related to directory permissions. Each of object domains uses different sets of permissions. Among the object domains, directory is an object domain with most frequently changed permissions. Out of a total of 2714 permissions changes, we observe that permissions related to directory, file, process, and socket object domains change 1222 (45.0%), 803 (29.5%), 199 (7.3%), and 146 (5.3%) times, respectively.

We observed that some of the evolution patterns appear to occur more frequently. We found five, one, and three evolution patterns covering 86.2%, 100%, and 38.6% of evolved rules for SELinux ACP, VCL ACP, and Snort ACP, respectively.

Table 5.5. ACP evolution patterns in our subject ACPs.

ACPs	# Frequent Patterns	% Coverage	# Sum of Frequency	# Highest Frequency
SELinux (Directory permission)	5	86.2	1066	568 (46%)
VCL	1	100	7	7 (100%)
Snort	3	38.6	780	488 (24%)

For SELinux ACP, in Table 5.6, we categorize permissions to four permissions sets according to purpose of a use (such as read and write). To control a directory, policy authors can use some of 23 permissions (that are pre-defined in SELinux). Policy authors often follow a pattern: “read”, “Ioctl”, and “lock” permissions together for allowing the reading of a directory. For each group, we use terms, “Basic Access Permission Set (BS)”, “Read File Permission Set (RS)”, “Write File Permission Set (WS)”, and “Advanced Permissions Set (AS)”. We describe top five most frequent patterns that appear in SELinux ACP.

- Remove Read Permission Pattern: The most frequently occurring evolution pattern is BS ∪ RS → BS. We observed that this evolution pattern happened 568 times (46%) out of the total number of directory permission changes. This pattern infers that the policy authors remove “Read File Permission Mode”. As a result of this evolution, subjects can perform access, open, and search directory described in “Basic Access Permission Mode”.

Table 5.6. Permissions in SELinux ACP

	Permissions	Description
Basic Access Permissions	• getattr: get file attributes for file, such as access mode • open: open a directory • search: search access	Access, open, and search directory
Read File Permissions	• read: read file contents • Ioctl: IO control system call requests • lock: set and unset file locks	Read, lock, and IO control for file contents
Write File Permissions	• write: general write access • add_name: add a file to the directory • remove_name: remove a file from the directory	Write file contents which required adding and removing the file
Advanced Control Directory (ACD) Permissions	• create: create new file. • link: create another hard link to file • rename: rename a file • reparent: rename into a different parent directory • rmdir: remove the directory • setattr: change file attributes such as access mode • unlink: Remove hard link (delete)	Advanced access controls such as remove, rename, change file locations, etc.

- Add Read Permission Pattern: BS $\rightarrow$ BS $\cup$ RS. We observed that this evolution pattern happened 234 times (19%) out of the total number of directory permission changes.
- Remove Read & Write Permission Pattern: BS $\cup$ RS $\cup$ WS $\rightarrow$ BS. We observed that this evolution pattern happened 86 times (7%) out of the total number of directory permission changes.

- Add Read & Write Permission Pattern: BS → BS ∪ RS ∪ WS. We observed that this evolution pattern happened 81 times (7%) out of the total number of directory permission changes.
- Add Write Permission Pattern: BS ∪ RS → BS ∪ RS ∪ WS. We observed that this evolution pattern happened 81 times (5%) out of the total number of directory permission changes.
- Add Advanced Permission Pattern: BS ∪ RS ∪ WS → BS ∪ RS ∪ WS ∪ AS happened 39 times (3%) out of the total number of directory permission changes.

Table 5.7 shows three permission groups for VCL ACP. From VCL ACP, we found only one evolution pattern, which happened seven times (100%) out of the total number of VCL permission changes.

- Add Share Permissions Pattern: the most frequently occurring evolution pattern is to add "Share Permissions" for existing permissions set. This pattern is to allow users to share her/his virtual computing images (100%).

Due to our limited data of VCL ACP, we do not find evolution patterns that happened often. We could find more evolution patterns if we collect larger and longer historical data of VCL ACP. Moreover, share permissions and other permissions are independent.

Table 5.7. Permissions in VCL ACP

	Permissions	Description
Basic Permissions	• imageAdmin: allows users to do administrative tasks with images in image groups • imageCheckOut: allows users to make reservations for images in image groups	Allow reserve and use virtual computing images
Share Permissions	• serverCheckOut: allows users to make reservations through the Server Profiles and allow other users access to the reservations • serverProfileAdmin: allows users to manage the Server Profiles	Allow share one's reserved virtual computing image
Advanced Administrative Permissions	• groupAdmin - grants users access to the Manage Groups portion	Allow advanced resource controls

Table 5.8 shows permissions used in Snort ACP. In Snort ACP, we identified three evolution patterns:

- Add "fast_pattern" and "content" Pattern: the most frequently occurring evolution pattern is to add fast_pattern permission into an existing permissions set for allowing the rules to use the fast pattern matcher for filtering packets. This pattern happens 488 times (24%) out of the total number of Snort permission changes.

Table 5.8. Permissions use in Snort ACP

	Permissions	Description
Basic Snort Rule Permissions	• flow: permit rules to only apply to certain directions of the network traffic flow. • content: permit the rules to search for specific content in the packet. • classtype: categorize a rule as detecting one of attack types.	Access, open, and search directory
Optional Snort Rule Permissions	• fast_pattern: allows the rules to use the fast pattern matcher for monitoring packets • metadata: allows the users to write additional information such as a key-value-format • pcre: allows the rule to be written using perl compatible regular expressions. • The distance keyword allows the rule writer to specify distance that helps ignore parts of content.	Read, lock, and IO control for file contents

- Add "fast_pattern" and "metadata" Pattern: this pattern is to add "fast pattern" with "metadata" together. This pattern happens 228 times (11%). "metadata" allows the users to write additional information in the rule.
- Remove "distance" Pattern: this pattern is to remove "distance". This pattern happens 64 times (3%).

In summary, we can find evolution patterns covering a large number of evolved rules. We extracted five, one, and three evolution patterns from SELinux, Snort, and VCL ACPs. We observed that some of evolution patterns appear to occur more frequently. For example, we

found evolution patterns that happened for 568 and 488 times in SELinux and VCL ACPs, respectively. This result implies that policy authors tend to use common evolution patterns when they modify rules.

RQ4. How effective is our model at predicting the evolution of ACP change patterns?

Figure 5.2 and Figure 5.3 show our prediction results. We evaluated effectiveness of our model for prediction. Y-axis shows the probability of precision, recall, and F-measure values. X-axis indicates the number of the most likely states (i.e., predicted states) that we can choose.

Given a state BS ∪ RS, we expect that this state is likely to evolve into BS, BS ∪ RS ∪ RS. We first consider BS (indicating the highest probability) as a predicted state. If the given state evolves into BS, our prediction result is classified into "True Positive". Otherwise, our prediction result is classified into "True Negative".

For SELinux ACP, we observed that the prediction precision and recall values are roughly 0.8 (80%) and 0.7 (70%), respectively, using the most likely first four states. The recall value 0.7 means that seven out of ten evolution cases can be predicted (showing a list of predicted states). The precision value 0.8 means that eight out of ten evolution cases (that can be predicted) are correctly predicted.

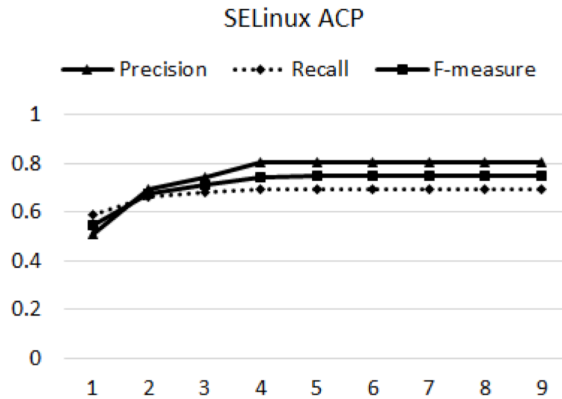

Figure 5.2. SELiux ACP evolution prediction precision, recall, and F-measure by choosing the most likely first 1, first 2, …, first 9 states based on ranking by SELinux ACP

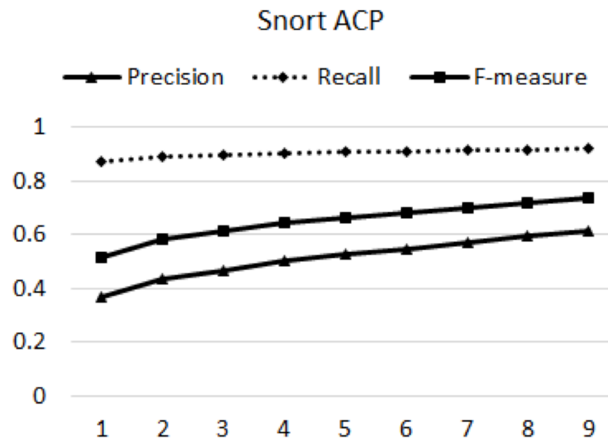

Figure 5.3. Snort ACP evolution prediction precision, recall, and F-measure by choosing the most likely first 1, first 2, …, first 9 states based on ranking

For Snort ACP, we observed that the prediction precision and recall values are roughly 0.6 (60%) and 0.8 (80%), respectively, using the most likely first four states. The recall value 0.8 means that eight out of ten evolution cases can be predicted (showing a list of predicted

states). The precision value 0.6 means that six out of ten evolution cases (that can be predicted) are correctly predicted. The prediction value for Snort ACP is lower than that of SELinux ACP by roughly 10%. However, the recall value for Snort ACP is higher than SELinux ACP by roughly 10%.

We do not consider VCL ACP for our experiments on prediction since VCL ACP has rarely occurring evolution patterns. Therefore, due to an insufficient number of evolution patterns, we have difficulty in prediction.

In summary, for SELinux ACP and Snort ACP, we observed that historical data of evolution patterns is effective at predicting ACP evolution. When we consider the most likely four states to move (for ACP evolution), we measured a precision of 50-80% and a recall of 70-90% showing high predictive power. This result implies that policy authors tend to use evolution patterns when they modify ACPs. Such information could further help improve tools by recommending frequent evolution patterns over infrequent evolution patterns to policy authors.

5.4.1 Threats to Validity and Limitations

We identified limitations within our study. However, we believe that these limitations could not invalidate our results. First, the systems and their ACPs in evolution studies may not be representative of the entire population. To make our results statistically significant, we collect ACP changes over a long timeframe (1-3 years) to yield statistically meaningful

results. Second, because ACPs are constantly evolving, some of the evolution patterns may be different for other versions of ACPs. Our study does not consider such effects that result from different versions and thus may suffer from this limitation.

5.5 **Chapter Summary**

We have conducted an empirical study of the evolution of centralized ACPs by providing observations about ACP evolution trends, practices, and their accompanying evolution patterns. We found evolution patterns characterizing changes of permissions of a subject (e.g., users or processes) with respect to an object (e.g., sensitive data).

We performed an empirical study by analyzing the ACP changes of three systems: SELinux, VCL, and Snort. We empirically observed growth trends of ACPs and corresponding systems in terms of the number of policy lines and lines of code (LOC), respectively. We found that the growth of our subject ACPs and systems increase linearly. We formalized evolution patterns characterizing policy changes. We observed that policy authors follow common evolution patterns that appear to occur more frequently. We built a prediction model based on the collected evolution patterns. Our evaluation results indicated that our model could predict evolution patterns in ACPs with a precision of 50-80%, a recall of 70-90% and an F-measure of 65-75%.

6 Systematic Structural Testing

We next present a systematic structural testing approach [26, 27] for security policies. Our approach analyzes security policies under test and generates test cases automatically to achieve high structural coverage. As the quality of protection provided by a security policy directly depends on the quality of its policy (i.e., configuration), ensuring the correctness of security policies is important and yet difficult. We present a systematic structural testing approach, which is effective in scenarios when usage patterns are difficult to infer from security policies under analysis to help detect faults. Instead of mined usage patterns, our approach is based on the concept of policy coverage, which helps test a policy's structural entities (i.e., rules, predicates, and clauses) to check whether each entity is specified correctly.

6.1 Introduction

A firewall is typically placed at the point of entry between a private network and the outside Internet such that all network traffic has to pass through it. In a distributed system, messages are encapsulated into packets, which often pass through multiple access points in a network and firewalls are responsible for filtering, monitoring, and securing such packets [38]. Corruption or misconfiguration in firewalls may cause that the firewalls fail to filter malicious packets properly and affect the performance and security of a distributed system.

Correctly specifying firewall policies is a critical and yet challenging task for building reliable firewalls [59] with three factors. First, the rules in a firewall policy are logically

entangled because of the conflicts among rules and the resulting order sensitivity. Second, a firewall policy may consist of a large number of rules. A firewall on the Internet may consist of hundreds or even a few thousands of rules. Third, an enterprise firewall policy often consists of legacy rules that are written by different operators, at different times, and for different reasons, which make maintaining firewall policies even more difficult.

In this chapter, we propose firewall policy testing based on the concept of firewall policy coverage, which helps test a firewall policy's structural entities (i.e., rules, predicates, and clauses) to check whether each entity is specified correctly. In firewall policy testing, test inputs and outputs are packets and their evaluated decisions (against the firewall policy under test), respectively. Given test packets and the policy under test, when evaluating packets against the policy, our coverage measurement tool measures firewall policy coverage —- which entities of the policy are involved (called "covered") in the evaluation. Moreover, our systematic firewall policy testing helps detect faults with the test packets, which often do not follow some configuration mistake patterns (e.g., anomalies [4, 39] and configuration errors [59]). Intuitively, policy testers shall generate test packets to achieve high structural coverage, which helps investigate a large portion of policy entities for fault detection.

As manual test-packet generation is tedious, we have developed an automated packet-generation tool (that can generate packets) for four packet-generation techniques: the random packet generation technique, the one based on local constraint solving (considering individual rules locally in a policy), the one based on global constraint solving (considering

multiple rules globally in a policy), and the one based on boundary values. As generated packets are often large and manual inspection of packet- decision pairs is tedious, we have developed an automated packet reduction tool to reduce the number of packets while keeping the same level of structural coverage.

We have conducted an experiment on a set of real firewall policies with mutation testing [2], which is a specific form of fault injection that creates faulty versions of a policy by making small syntactic and semantic changes. We generate packet sets (for each policy) with the packet generation techniques. Our experimental results show that a packet set with higher structural coverage (including rule, predicate, and clause coverage) often achieves higher fault-detection capability (i.e., detecting more injected faults), which is measured through the number of "killed mutants" (i.e., detected faults). On the comparison of packet sets and their reduced packet sets, our experimental results also show that a reduced packet set achieves similar fault-detection capability with the original packet set.

6.2 **Example**

In firewall testing, exhaustive testing (i.e., executing all possible test packets) is time consuming and inefficient. Instead of exhaustive testing, we focus on testing to cover only specific entities (i.e., a predicate tested to be false or true) based on a set of defined coverage criteria.

6.2.1 Definition

Treating the firewall policy under test as program code (i.e., IF-THEN-ELSE statements), we apply structural coverage criteria similar to the ones defined by Ammann et al [2]. In this dissertation, a test suite is a set of packet-decision pairs to check whether a packet is evaluated to its corresponding expected decision. Table 6.1 summarizes the notations used in this chapter.

To measure cyclomatic complexity of our structural coverage criteria, we use McCabe's cyclomatic complexity [74] that is a quality metric that measures the number of linearly independent paths through the program. If all decisions of predicates are binary, McCabe cyclomatic complexity (i.e., McCabe number) is $v(G) = N_P + 1$ where N_P is the number of binary predicates.

We next define rule, predicate, and clause coverage criteria as follows.

Definition 1: Rule Coverage Criterion (*RCC*) for a test suite requires that for each rule r in a policy, the evaluation of the test packets in the test suite needs to match r (i.e., make a Boolean expression in r's predicate p to be evaluated to true) at least once, respectively. Because all decisions of the predicates of the rules are binary, McCabe cyclomatic complexity is $v(G) = N_P + 1$ where N_P is the number of the predicates. The cyclomatic complexity of satisfying RCC is N_P because we remove a case where all predicates are evaluated to false from V(G) based on the definition of *RCC*.

Table 6.1. Summary of notations

P	a set of predicates of the rules in a policy
C	a set of clauses of the predicates in a policy
r_i	a rule in a firewall policy
p_i	a predicate in a rule ri
c_i	an ith clause in a predicate
F_i	a field (e.g., IP address)
D_i	domain of field F_i (e.g., [0, 232 − 1] for the IP address)
S_i	a subset of domain D_i (e.g., [2,5])
$Cp_i(c_j)$	a constraint of a clause c_j in a predicate p_i
$C(p_{i)}$	a constraint of a predicate p_i

In other words, RCC requires that for each predicate p, p is evaluated to true at least once. Figure 6.1 shows example firewall rules where only two fields F_1 and F_2 are used.

For example, given two test packets, k_1 (3, 5) and k_2 (6, 10) over two fields F_1 and F_2, both predicates p_1 and p_2 (of r_1 and r_2, respectively) are evaluated to true. These two test packets achieve RCC. More specifically, k_1 evaluates p_1 to true, causing r_1's decision to be returned without further evaluating p_2. k_2 evaluates p_1 to false and next evaluates p_2 to true. Note that when a packet finds the first-matching rule r (i.e., evaluating a predicate to true), policy evaluation stops and returns r's decision as a final decision.

Definition 2: Predicate Coverage Criterion (PCC) for a test suite requires that for each predicate p of the rules in a policy, the evaluation of the test packets in the test suite needs to

make a Boolean expression in p to be evaluated to true and false at least once, respectively. Because all decisions of the predicates are binary, McCabe cyclomatic complexity is v(G) = $N_P + 1$ where N_P is the number of the predicates. Therefore, the cyclomatic complexity of satisfying PCC is $N_P + 1$.

To achieve *PCC*, in addition to k_1 and k_2, we require one more packet such as k_3 (6, 11) that evaluates p_2 to false. Figure 6.2 illustrates these three test packets that evaluate all combinations of true and false (of p_1 and p_2). N/A represents a not-applicable predicate or rule during packet evaluation. For k_1, we mark N/A in p_2's evaluation because k_1's decision is determined without further evaluating p_2.

Covering every predicate in a firewall requires at most $2n$ test packets, where n is the number of rules. However, the minimal number of test packets (for *PCC*) could be less than $2n$ because a single test packet can satisfy multiple true or false branches of predicates. As *RCC* and *PCC* do not require each clause to be covered, we then define clause coverage criterion (*CCC*), which specifically targets at covering each clause in a predicate.

Definition 3: Clause Coverage Criterion (*CCC*) for a test suite requires that for each clause c of the predicates in a policy, the evaluation of the test packets in the test suite needs to make a Boolean expression in c to be evaluated to true and false at least once, respectively. Because all decisions of the clauses are binary, McCabe cyclomatic complexity is v(G) = N_C

$$r_1 : F_1 \in [2, 5] \wedge F_2 \in [5, 10] \rightarrow accept$$
$$r_2 : F_1 \in [6, 7] \wedge F_2 \in [5, 10] \rightarrow discard$$

Figure 6.1. Example firewall rules.

+1 where N_C is the number of the clauses. Therefore, cyclomatic complexity of satisfying *CCC* is N_C +1.

In *CCC*, each clause is required to be evaluated to true and false at least once independently from other clauses. Consider that p_1 includes two clauses c_1 and c_2 (with regards to F_1 and F_2, respectively). Note that the boolean value of p_1 is equal to $c_1 \wedge c_2$. Figure 6.3 illustrates four test packets that evaluate all combinations of true and false (of c_1 and c_2) and the corresponding boolean value of p_1. There are several ways to cover clauses in p_1: (1) select k_2 and k_3 or (2) select k_1 and k_4. However, instead of the first selection, the second selection has an advantage to increase the coverage in terms of *RCC* and *PCC*.

6.2.2 Structural Coverage

We have developed three structural coverage measurements that monitor whether rules, predicates, or clauses are covered when evaluating packets against the policy under test. For each structural coverage criterion, we define coverage measurements as follows.

packet	p_1	p_2	matching rule
k_1: (3, 5)	True	N/A	r_1
k_2: (6, 10)	False	True	r_2
k_3: (6, 11)	False	False	N/A

Figure 6.2. Sample packets for all combinations of true and false values of predicates p_1 and p_2.

packet	c_1	c_2	$p_1 = (c_1 \wedge c_2)$
k_1: (3, 5)	True	True	True
k_2: (6, 10)	False	True	False
k_3: (3, 11)	True	False	False
k_4: (6, 11)	False	False	False

Figure 6.3. Sample packets for all combinations of true and false values of clauses c_1 and c_2.

Rule coverage measurements. For the rule coverage criterion, rule coverage is the percentage of the number of covered rules (i.e., a Boolean expression in a predicate being evaluated to true) in a policy.

Predicate coverage measurements. For the predicate coverage criterion, predicate coverage is the percentage of the number of covered predicates (i.e., a Boolean expression in a predicate being evaluated to true or false).

Clause coverage measurements. For the clause coverage criterion, clause coverage is the percentage of the number of covered true or false values of clauses (i.e., a Boolean expression in a clause being evaluated to true or false) .

6.2.3 Structural Coverage and Fault Detection

Policy testers may generate and select a test suite to achieve a certain (high) level of coverage. However, our main objective, through testing, is to detect faults in the firewall policy while reaching a certain level of coverage. Coverage analysis helps investigate a larger portion of entities for fault detection using a test suite that achieves higher structural coverage.

Consider that a fault in entities (i.e., rules, predicate, or clause) may cause to output incorrect decisions when evaluating some packets. A fault in a rule's decision (e.g., using accept by mistake instead of discard) is discovered if and only if the rule is covered and the derived decision is verified. A test suite with high rule coverage may detect such faults easily and increase our confidence on the correctness of the policy against such faults. Similarly, a test suite with high predicate/clause coverage may have a high chance to detect faults in predicates/clauses. Therefore, we are interested in covering each entity at least once to exercise a wide range of the policy's behavior.

6.3 Approach

This section presents our framework for testing firewall policies. Figure 6.4 shows the overview of framework of our approach. Our framework includes three phases: test packet generation, test reduction, and fault detection. In the test packet generation phase, our test packet generation component analyzes a firewall policy and generates test packets to cover entities (e.g., predicates and clauses) in the policy. We propose four different test packet

generation techniques. In the test reduction phase, the test reduction component reduces the number of packets based on coverage criteria by including only packets that help increase policy coverage measurement during evaluation. In the fault detection phase, the policy authors manually inspect whether the actual decisions (i.e., evaluated decisions of the generated packet against the firewall policy) are consistent with expected decisions. If the authors find any inconsistent decisions, the authors determine that they detect a fault in the policy.

6.3.1 Test Packet Generation

As manually generating packets for testing policies is tedious, we have developed four techniques to automatically generate packets for the policy under test. The objective is to generate packets for achieving high structural coverage. This section describes four packet generation techniques (developed in our approach): the random packet generation technique, the packet generation technique based on local constraint solving, the packet generation technique based on global constraint solving, and the packet generation technique based on boundary values. The key difference between the second and third techniques is the scope (i.e., local or global) of constraints used in the packet generation. While the second and third techniques generate packets based on random values within values solved by each constraint solving, the fourth one generates test packets based on boundary values within values solved by local constraint solving.

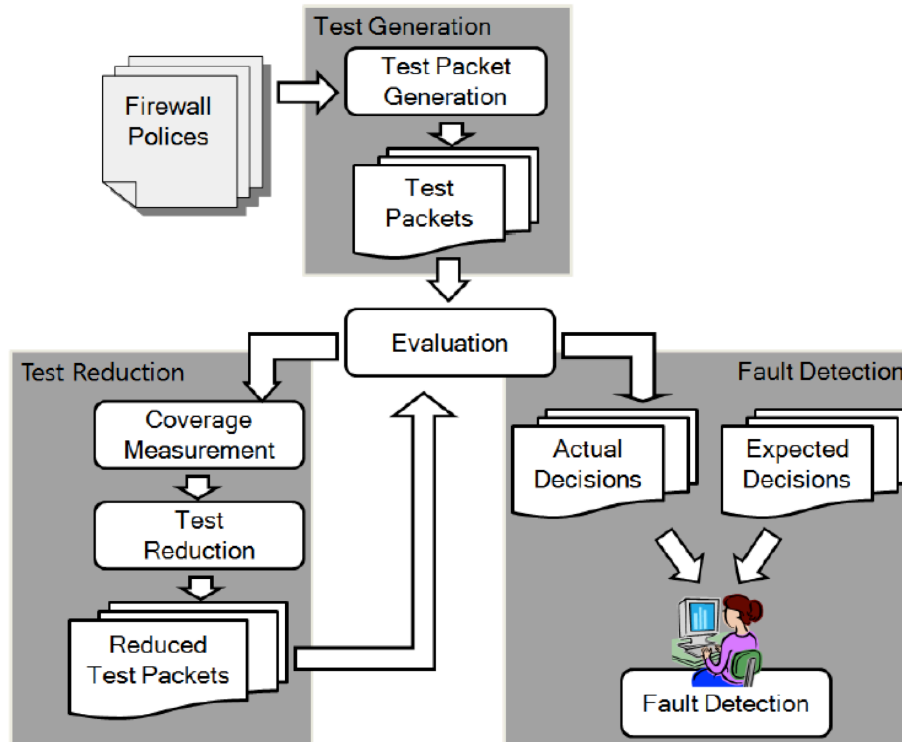

Figure 6.4. Framework overview.

In this section, p and (p) denote a predicate and its constraint, respectively. To evaluate p to be true (false), a packet should satisfy the constraint (p) $(\neg C(p))$ (for the true (false) branch of p). (p) is represented in the form of $C_p(c_1) \wedge .... \wedge C(c_n)$, where $C_p(c_1)$, ..., $C_p(c_n)$ are the constraints of the clause c_1, ..., c_n in p, respectively.

1) Random Packet Generation Technique: The random packet generation technique is straightforward. A packet k is in the form of $(k_1, ..., k_n)$, where k_1, ..., k_n are numeric values over fields (such as source addresses), whose domains are denoted by D_1, ..., D_n). Given the

domains of the policy under test, the generator for the technique automatically generates a packet k by randomly selecting k_1, ..., k_n (within the domain D_1, ..., D_n, respectively). While the technique does not require the policy itself in test generation and can quickly generate a large number of test packets, the technique often lacks effectiveness to achieve high structural coverage with the generated packets. Due to randomness, the number of the entities (i.e., predicates or clauses) being covered is often small in comparison to the total number of the entities in the policy under test.

2) Packet Generation Technique based on Local Constraint Solving: In general, packet generation should focus on generating packets to cover those entities (i.e., predicates and clauses) that have not been covered previously. Different from the preceding technique, the technique based on local constraint solving statically analyzes the entities in an individual rule and generates packets to evaluate the constraints (i.e., conditions) of the entities to be true or false. The technique takes into account local constraints (given by a rule) without considering the impact of other rules in the policy.

More specifically, the generator constructs constraints (p) and $\neg C(p)$ (for both true and false branches of p) for each rule. The generator generates a packet based on the concrete values to satisfy each constraint. As the generator generates packets based on satisfying constraints in predicates, the generated packets may not be effective in covering each clause (to be true and false). To target at covering many clauses, the generator constructs combinations of $C(c_i)$ and $\neg C_p(c_i)$. For example, combinations $C(c_1)$ $\wedge$ .... $\wedge$ $C(c_n)$ (for true

branches of all clauses) and $\neg C_p(c_1) \wedge .... \wedge \neg C(c_n)$ (for false branches of all clauses) can be considered.

There are two major limitations of the technique. First, the generated packets may fail to cover target entities due to overlapping predicates (i.e., predicates that can be satisfied by the same packet) across multiple rules. As shown in Figure 2.2, a packet k (whose destination IP address is 192.168.0.0 and protocol type is UDP) satisfies the predicates of both r_1 and r_3 but fails to be evaluated against r_3, which can be k's potential target entities. Second, the technique cannot determine whether a structural entity could be covered in advance. Some rules may be completely shadowed by other rules and never evaluated. In such cases, there is no criterion to decide whether to generate additional packets (based on other more capable solutions to solve the same constraints) or stop testing.

3) Packet Generation Technique based on Global Constraint Solving: To better generate packets to cover target entities, our generator (based on global constraint solving) analyzes the policy under test and generates packets by solving global constraints (collected from the policy). The motivation of global constraint solving is to take into account the influence of overlapping predicates across rules. Covering entities in a rule requires that the predicates of all the preceding rules should be evaluated to false. To find such entities, we define rule reachability as follows.

Definition 4: Rule reachability of a packet k to reach a rule r_i in a policy requires that k evaluates r_i's preceding rules' predicates to false and reaches the rule.

We may generate packets to reach and evaluate all the reachable rules in the policy. To cover entities in a rule r_i, we explore a (path) constraint $Path(r_i)$ that represents rule r_i reachability. $Path(r_i)$ is additionally used upon the preceding technique to cover target entities by taking into account the impact of overlapping predicates in the preceding rules.

More specifically, $Path(r_i)$ is represented as the form of $\neg C(p_1) \wedge .... \wedge \neg(p_{i-1})$ where $C(p_1)$, ..., $C(p_{i-1})$ are the predicate constraints in the preceding rules r_1, ..., r_{i-1}. Given the path constraint $Path(r_i)$, to cover the predicate p_i in r_i, the generator constructs two constraints $Path(r_i) \wedge C(p_i)$ (for the true branch of p_i after reaching r_i) and $Path(r_i) \wedge \neg C(p_i)$ (for the false branch of pi after reaching ri). As the generator generates packets based on solutions of constraints $Path(r_i) \wedge C(p_i)$ and $Path(r_i) \wedge \neg C(p_i)$, the packets reach r_i and exercise r_i's true and false branches, respectively.

Given the constraints, the generator generates packets based on solutions for the collected constraints. This technique is useful to generate packets with high structural coverage by taking into account the impact of the preceding rules of a target rule. However, this technique requires higher analysis time (e.g., constraint-solving cost) than the two preceding techniques.

Algorithm 1. Packet Generation Technique Based On Boundary Values.

Input: Firewall policy P where n is the number of rules, $r_1, r_2, ..., r_n$. $C_p(c_1), ..., C_p(c_m)$ where each $C_p(c_j)$ is the jth clause constraint of constraints C_p of a rule. $D_1, D_2, ..., D_m$ where D_j is a domain of the jth field.

Output: A set of packets.

$output = \langle\rangle$;
for $i := 1$ *to* n **do**
  C_p = constraints of r_i;
  for $j := 1$ *to* m **do**
    k_j = **MinValue** $(C_p(c_j))$;
  $k = (k_1, k_2, ..., k_j)$;
  $output = output \cup k$;
  for $j := 1$ *to* m **do**
    l = **MinValue** $(C_p(c_j))$;
    if $l \leq MinValue(D_j)$ **then**
      $k_j = l$;
    else
      $k_j = l - 1$;
  $k = (k_1, k_2, ..., k_j)$;
  $output = output \cup k$;
  for $j := 1$ *to* m **do**
    k_j = **MaxValue** $(C_p(c_j))$;
  $k = (k_1, k_2, ..., k_j)$;
  $output = output \cup k$;
  for $j := 1$ *to* m **do**
    r = **MaxValue** $(C_p(c_j))$;
    if $r \geq MaxValue(D_j)$ **then**
      $k_j = r$;
    else
      $k_j = r + 1$;
  $k = (k_1, k_2, ..., k_j)$;
  $output = output \cup k$;
return *output*

4) Packet Generation Technique based on Boundary Values: To better generate packets to detect a fault in a firewall policy, our generator (based on boundary values) analyzes the policy under test and generates packets based on boundary values by solving local constraints (collected from the policy). The generated packets include boundary values, which are on the

range boundaries (i.e., the smallest value and the largest value) of each field. Intuitively, when a fault is injected to a firewall policy, the policy could reveal a faulty policy behavior.

The technique selects boundary values instead of random values from values satisfying rule constraints. Boundary values are the values around the smallest and largest values of a clause in a rule. For example, Figure 6.1 has a rule r_1 that includes two fields $F_1 \in [2, 5]$ and $F_2 \in [5, 10]$. For the smallest value 2 of F_1, boundary values are 1 and 2 that evaluate F_1 to be false and true, respectively. For the largest value 5 of F_1, boundary values are 5 and 6 that evaluate F_1 to be true and false, respectively. Similarly, we can select boundary values 4, 5, 10, and 11 for $F_2 \in [5, 10]$. Given boundary values of F_1 and F_2, we can generate four packets (1, 4), (2, 5), (5, 10), and (6, 11) to cover true and false branches of clauses in r_1.

More specifically, the generator generates packets based on boundary values to cover true and false branches of clauses in a rule r_i. Algorithm 1 presents our technique to generate packets based on boundary values. $C(c_j)$ is the jth clause constraint in a rule. In the algorithm, Lines 4-7 present that the generator generates a packet based on the smallest boundary values S to satisfy positive constraints (i.e., $C(c_1) \wedge .... \wedge C(c_n)$ for true branches of all clauses) of each rule. Lines 8-15 present that the generator generates a packet based on boundary values (next to S) to satisfy negative constraints (i.e., $\neg C(c_1) \wedge .... \wedge \neg C(c_n)$ for false branches of all clauses) of each rule. Lines 16-19 present that the generator generates a packet based on the largest boundary values L to satisfy positive constraints of each rule.

Lines 20-27 present that the generator generates a packet based on boundary values (next to L) to satisfy negative constraints of each rule.

The generator generates packets based on boundary values within solutions for the collected constraints. This technique generates packets with high structural coverage (that can be achieved based on local constraint solving) and fault detection with using boundary values instead of any other values feasible to cover a target entity.

However, the packets generated based on boundary values of a rule's constraints could not reach the rule due to the impact of overlapping predicates in the preceding rules. To further generate packets based on correctly identified boundary values, we leverage an existing technique [36] to remove redundant overlapping predicates of firewall policies. In addition, this redundancy removal technique helps reduce the number of generated packets based on boundary values when redundant rules are removed and the number of rules is decreased.

6.3.2 Test Reduction

Manual inspection of a test suite (which is a set of packet-decision pairs) is time-consuming and tedious. Therefore, we should reduce the size of the test suite for inspection without incurring substantial loss in fault-detection capability. Since structural coverage is an important factor for reflecting fault detection capability, we can reduce the size of the test suite while keeping its coverage level.

Given a packet set, we evaluate each packet set against the policy. We use a greedy algorithm that removes a packet from the packet set if and only if evaluating the packet does not increase any of the coverage metrics that are achieved by previously evaluated packets in the packet set.

6.3.3 Measuring Fault-Detection Capability

Fault detection is a focus of any testing process. We aim to investigate the relationship between firewall policy structural coverage achieved by a packet set and the packet set's fault-detection capability. We adopt mutation testing [2] to measure the fault-detection capability of the packet set.

In policy mutation testing, we inject a fault into the original policy and thereby create a mutant (faulty version). Injected faults can be of various types including simple mistakes (e.g., incorrect decision in a rule) and complex configuration errors involving multiple rules. The intuition behind mutation testing is that if a policy contains a fault, there will usually be a set of mutants that can be detected (killed) only by a test packet that also detects that fault.

When different decisions are produced by the evaluations of the same test packet on the original policy and its mutant, the test packet is adequate to detect the fault in the mutant and we say that the mutant is "killed". When various mutants are used, fault-detection capability of a test suite is measured through the mutant-killing ratio, which is the number of mutants killed by the test suite divided by the total number of mutants.

Table 6.2 shows the chosen mutation operators for firewall policies and their descriptions. Mutation operators may change predicates, clauses, or decisions of a policy. We classify mutation operators into two groups: (1) rule-level mutation operators including RP, RPF, CRO, CRD, AR and RMR and (2) clause-level mutation operators including RCT, RCF, $CRSV$, $CREV$, $CRSO$, and $CREO$. The first group adds, removes, or modifies a rule in a policy. The number of generated mutants with each mutation operator is equal to the number of rules of the policy. The second group modifies a clause in a rule. The number of generated mutants with each mutation operator is equal to the number of clauses.

However, syntactic changes of firewall policies cannot guarantee semantic changes of the firewall policies. In other words, the mutant generator for each mutation operator may generate semantically equivalent mutants that are mutants with the same behaviors as the original policy; any test packet cannot kill an equivalent mutant. In order to guarantee semantic changes of firewall policies after fault injection, we leverage an existing change-impact analysis tool [34] on firewall policies to determine whether the modifications incur any semantic changes. Given two policies p_1 (an original policy) and p_2 (its corresponding mutant), change-impact analysis is to analyze what would be different policy behaviors between p_1 and p_2.

6.3.4 Implementation

Our implementation (written in Java) includes four components: packet generation, packet evaluation, packet reduction, and mutation generation. In the packet generation component,

Table 6.2. Mutation operators for policy mutation testing

Name	Description
Rule Predicate True (RPT)	A rule is applied to all packets by modifying every clause range to "*".
Rule Predicate False (RPF)	A rule is never applied to any packet by modifying every clause range to an invalid range (e.g., [10, 5]).
Rule Clause True (RCT)	A clause ci is applied to the field value f_{vi} of all packets by modifying the clause range to "*".
Rule Clause False (RCF)	A clause ci is never applied to the field value f_{vi} of all packets by modifying the clause range to an invalid range (e.g., [10, 5]).
Change Range Start point Value ($CRSV$)	The range in a clause is changed by modifying the start point value randomly.
Change Range End point Value ($CREV$)	The range in a clause is changed by modifying the end point value randomly.
Change Range Start point Operator ($CRSO$)	The range in a clause is changed by increasing the start point value by one.
Change Range End point Operator ($CREO$)	The range in a clause is changed by decreasing the end point value by one.
Change Rule Order (CRO)	Rule order is changed by exchanging the locations of two adjacent rules.
Change Rule Decision (CRD)	A rule's decision is inverted (i.e., accept to discard or discard to accept).
Add Rule (AR)	add a randomly generated rule in a policy.
Remove Rule (RMR)	remove the rule in a policy.

for packet generation based on local constraint solving, our packet generator selects random values (that satisfy a given constraint) for each field value of a test packet. For packet generation based on global constraint solving, we leveraged a theorem prover called Z3 [62]. The component statically analyzes and finds concrete solutions (i.e., numeric values), each of which is transformed to a test packet. If no solution exists, Z3 outputs unsolvable. For packet generation based on boundary values, our packet generator selects boundary values (that

satisfy a given constraint) for each field value. In order to remove redundancy, we leverage an existing tool [36] to detect redundancy in a firewall policy.

In the packet evaluation component, we developed a generic firewall evaluation engine to simulate evaluating packets against the policy under test. The engine parses and stores rules as a List. When evaluating a packet, the engine searches for the first-applicable rule and outputs the rule's decision. The engine also automatically compares the evaluated decisions (on the policy and the mutated policies) and log "killed" mutant information if the decisions are inconsistent.

In the packet reduction component, our packet reduction tool observes the details of covered entities and their covering packets as well as the details of uncovered entities when evaluating a packet set.

In the mutation generation component, our mutator automatically generates mutant policies by modifying the policy under test using the selected mutation operator.

6.4 **Evaluation**

We carried out our experiments on a laptop PC running Windows XP SP2 with 1G memory and dual 1.86GHz Intel Pentium processor. Our packet generation tool generates packet sets using the four techniques (random packet generation, packet generation based on local constraint solving, one based on global constraint solving, and one based on boundary

values). We use $Rand$, $Local$, and $Global$ to denote the packet sets generated by these first three techniques, respectively. We use $Bound_1$ and $Bound_2$ to denote the packet sets generated by the fourth technique on an original policy and its redundancy-removed policy, respectively. For each policy, we measured the structural coverage of each packet set and reduce the size of each packet set while keeping the same level of structural coverage. We use $Rand-$, $Local-$, $Global-$, $Bound_1-$, and $Bound_2-$ to denote the reduced packet sets, respectively.

The mutator generates mutants (using the defined mutation operators) by seeding faults in each policy (with one mutant including one seeded fault). For each policy and its mutants, the evaluation engine checked if a mutant is "killed" and measured mutant-killing ratios of each packet set (i.e., the number of mutants killed by the packet set divided by the total number of mutants). We compare our proposed four packet-generation techniques in terms of effectiveness to achieve structural coverage by the generated packet sets. In order to investigate the effect of structural coverage on fault-detection capability, we aim to demonstrate that packet sets with higher coverage can detect more faults than packet sets with lower coverage. We have also conducted the same experiment with reduced packet sets to further investigate whether this reduction significantly affects their fault-detection capability.

6.4.1 Instrumentation

We conducted experiments on 14 real-life firewall policies collected from a variety of sources. For the local and global constraint-solving packet-generation techniques, we first generated the following two constraints for each rule: (1) a constraint for evaluating every clause in the rule to true and (2) a constraint for evaluating clauses, each of which is within (but not equal to) its domain, to false and the remaining clauses (which subsume their domains) to true. Because many clauses in firewall policies subsume their domains (e.g., clauses with "*" marks in Figure 2.2) and these clauses cannot be evaluated to false, we evaluated such clauses to true in the second constraint as described earlier. The local constraint-solving packet-generation technique generated $n \times 2$ packets. The global constraint-solving packet-generation technique conjuncts the path constraint for a target rule with its two preceding constraints to form a new constraint for solving. If the new constraint is found to be infeasible (due to the impact of the path condition), this technique cannot generate packets to satisfy such constraints and may include fewer than $n \times 2$ packets. The packet-generation technique based on boundary values generated at most $n \times 4$ packets. The technique removes duplicate packets to reduce the number of packets. Moreover, the technique removes redundancy to help reduce the number of rules, which reflects the number of packets.

6.4.2 Comparison of Structural Coverage

Table 6.3 shows the basic statistics of each firewall policy. Columns 1-3 show subject names, numbers of rules (denoted by "#RL"), and generated mutants for each firewall policy

Table 6.3. Experimental results on firewall policies.

Policy	#RL	#MT	# Packets					# Reduced packets				
			Rand	Local	Global	Bound$_1$	Bound$_2$	Rand−	Local−	Global−	Bound$_1$−	Bound$_2$−
1 Firewall1	3	51	6	6	6	10	10	1	3	3	3	3
2 BACKUP	5	26	10	10	9	12	6	1	5	4	6	3
3 LAN-OUT	28	280	56	56	43	104	29	1	17	17	18	12
4 MAILOUT	18	206	36	36	26	64	22	1	10	9	12	9
5 MAIL	26	378	52	52	44	96	62	2	18	19	22	21
6 MAIL2	26	338	52	52	39	96	42	2	14	14	15	14
7 MAIL3	27	360	54	54	41	100	46	3	15	15	16	15
8 MAIL4	28	494	56	56	53	101	87	3	24	27	29	28
9 NEWS-OUT	14	185	28	28	25	48	34	2	11	11	13	12
10 NS3-OUT	17	179	34	34	29	56	22	1	11	13	14	10
11 RCPRO	23	233	46	46	34	77	30	3	14	11	15	11
12 RCPRO1	6	44	12	12	11	16	10	1	6	5	7	5
13 SSHOUT	16	152	32	32	23	54	18	1	8	7	9	8
14 WANIN	24	313	48	48	40	58	42	2	19	17	18	14
Average	18.6	231.3	37.2	37.2	30.2	63.7	32.8	1.7	12.	12.2	14.0	11.7

(denoted by "#MT"). Column group "# Packets" shows the size of the generated packet sets $Rand$ $Local$, $Global$, $Bound_1$, and $Bound_2$, respectively for each packet generation technique. Columns 9-13 show the size of their reduced packet sets (denoted by $Rand-$, $Local-$, $Global-$, $Bound_1-$, and $Bound_2$-), respectively. Note that the last row shows the average.

$Global$, $Bound_1$, and $Bound_2$ can achieve 100% of rule, predicate, and clause coverage when global constraints can be feasible to be solved. Firewall policies may include rules,

predicates, and clauses that are infeasible to reach. We consider only feasible constraints when we measure structural coverage.

We observe that $Global$ may contain fewer packets than $Rand$ and $Local$. The reason is that when solving a global constraint, the constraint can be infeasible to be solved and a constraint solver returns a decision of unsolvable — no packets are generated based on the decision. We observe that $Bound_2$ contains fewer packets than $Bound_1$ since the number of rules in the policy under test is reduced after redundancy removal.

Figure 6.5, Figure 6.6, and Figure 6.7 show the rule, predicate, clause coverage metrics, respectively, of each policy achieved by $Rand$, $Local$, $Global$, $Bound_1$, and $Bound_2$. We observe that $Rand$ achieved the lowest structural coverage. The reason is that randomly generated field values in generated packets have a low chance of satisfying constraints for a rule, predicate, or clause.

We observe that $Global$ achieves higher rule/predicate coverage than other packet sets. This observation is consistent with our expectation described in Section 6.3. On average, $Global$ is approximately 5% (2%) and 86% (35%) higher than $Local$ and $Rand$ in terms of rule (predicate) coverage. $Bound_1$ achieves similar rule/predicate coverage with $Global$. $Bound_2$ achieves lower coverage than $Bound_1$ because packets (generated based on a redundancy-removed policy) are not suitable to achieve high structural coverage for its original policy due to structure change after redundancy removal.

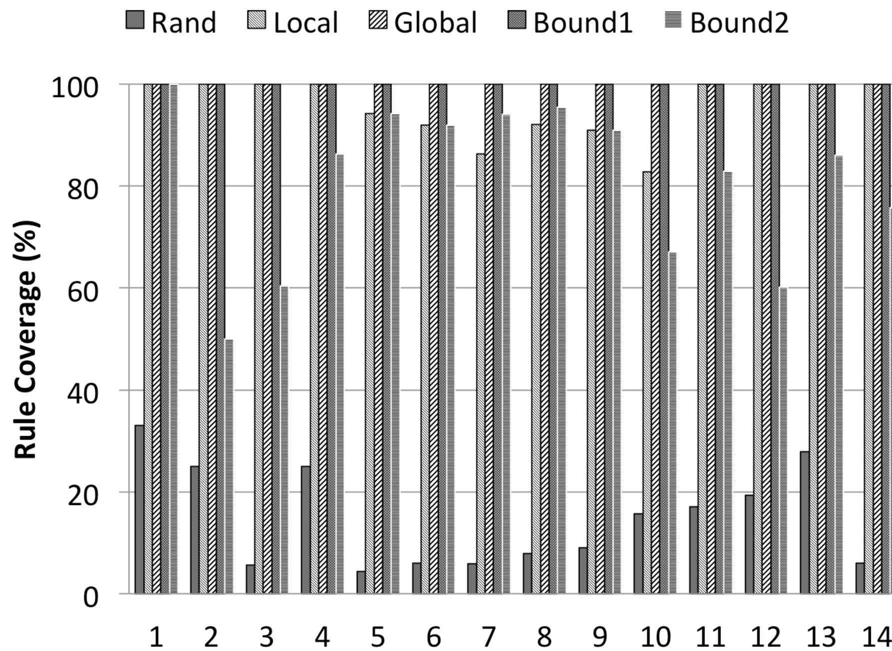

Figure 6.5. Rule coverage achieved by each packet set.

We also observe that for clause coverage, $Global$ achieves approximately similar (sometimes less) coverage with $Local$. As illustrated earlier, $Global$ may include fewer packets based on the constructed constraints. When a constraint is found to be infeasible, we did not take into account other clause constraint combinations, which may be feasible to solve for covering some of uncovered clauses. Instead, $Local$, $Bound_1$, and $Bound_2$ may cover some (but not all) target clauses among such uncovered clauses.

Furthermore, as our subjects have only a few or no overlapping predicates across rules, the packet-generation technique based on local constraint solving could generate a packet set

with almost the highest structural coverage. If predicates are more complex, we expect that $Global$ shall perform better than $Local$.

6.4.3 Comparison of Fault-Detection Capability

To find correlation between each structural coverage and mutation-killing ratios, we classify mutation operations into two categories, rule-level and clause-level mutation operators (explained in Section 6.3.3).

Figure 6.8 shows the average mutant killing ratios for all operators by policies. We observe that the mutant killing ratios are similar over the generated packet sets and their reduced packet set. For $Rand$, $Local$, and $Global$, the largest ratio difference between the generated packet sets and their reduced packet set is less than 2%. $Rand$ and $Rand-$ show the lowest mutant-killing ratios. As $Rand$ contains a relatively large number of packets and the lowest mutant-killing ratios, we observe that the size of a packet set is not highly correlated with fault-detection capability. We also observe that $Bound_1$ ($Bound_1$-) achieves the highest mutant-killing ratios among the generated packet sets (the reduced packet sets). While $Local$, $Global$, and $Bound_1$ achieve similar structural coverage, $Bound_1$ achieves the highest mutant-killing ratios. This result is expected as the evaluation of these packet sets can involve more structural entities and boundary values than the other packet sets.

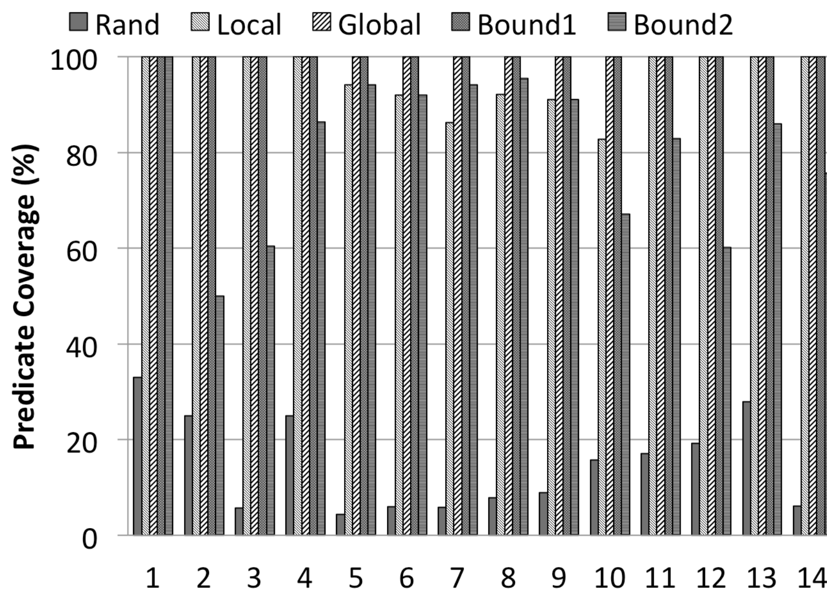

Figure 6.6. Predicate coverage achieved by each packet set.

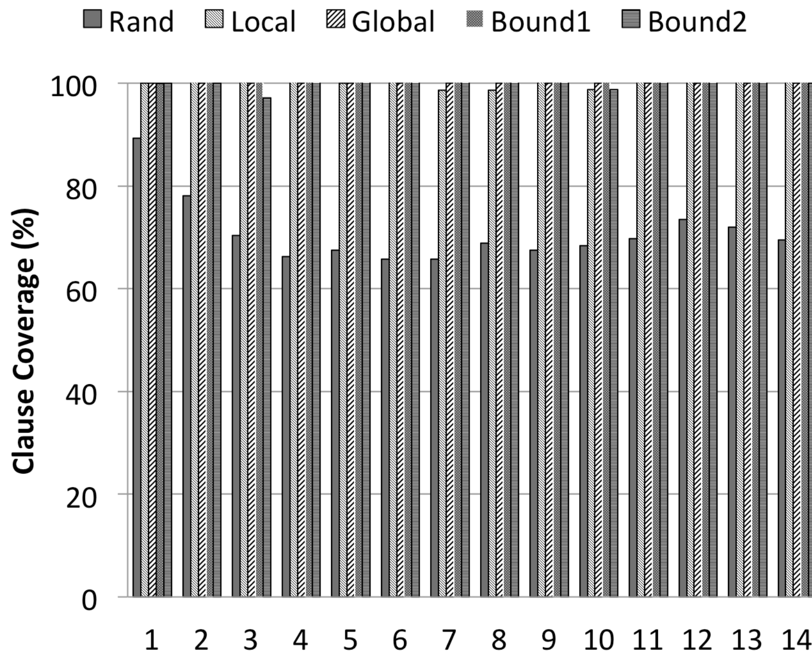

Figure 6.7. Clause coverage achieved by each packet set.

We also observe that, in Figure 6.8 for most cases, mutant killing ratios are below 60%. The reason for such low mutant killing ratios is that a policy can include various types of faults denoted in Table 6.2 and our test packet generation could not find all possible changed behaviors of a given policy. For a CRO-mutated policy, two rules swap locations. In order to detect such a fault, packets should match intersections of two packets. However, our test packet generation does not consider such intersections for test packet generation and cannot easily detect such a fault.

We next present more details about mutants being killed. Figure 6.9 shows the average mutant killing ratios for all policies by operators. For rule-level mutation operators, we observe that $Global$, $Global-$, $Bound_1$, and $Bound_1-$ achieve highest mutant-killing ratios. The reason is that the highest rule/predicate coverage achieved by $Global$, $Global-$, $Bound_1$, and $Bound_1-$ helps exercise more rules and detects faults in rules.

In Figure 6.9, we observe that our generated packet sets cannot detect any faults in the policies with AR faults. AR simulates a forgotten rule in a given policy. The reason for such low mutant-killing ratios is that our test packet generation is based on a set of rules in a given policy and does not have any information of a forgotten rule to help detect its fault. Moreover, randomly generating a packet for fault detection is not trivial as well due to a large domain of a firewall policy representation.

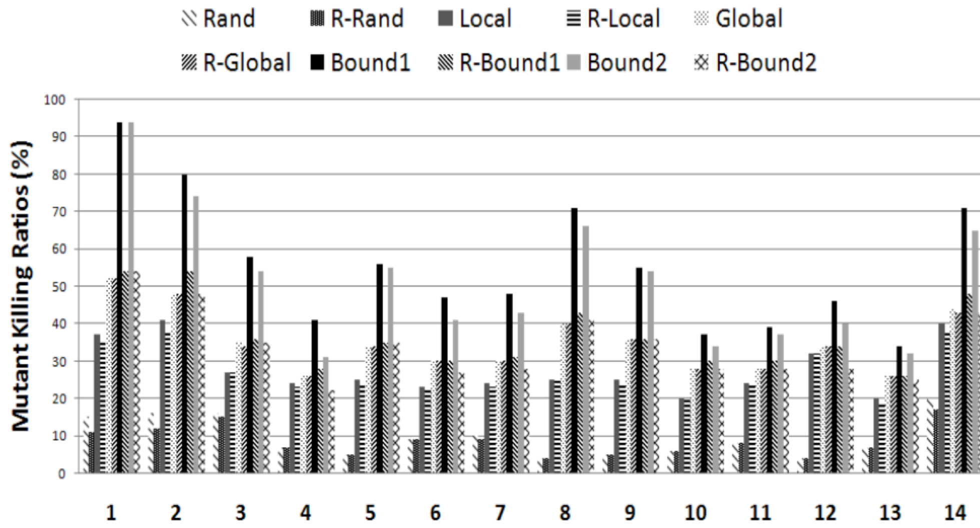

Figure 6.8. Mutant-killing ratios for all operators by subjects.

For example, an IP address field in a rule includes a subset of the IP address domain (i.e., [0, 28 − 1]), which is huge. There is a very low possibility that a randomly generated IP address field value in a packet could detect such a fault. In other words, in order to detect a fault in a rule, a packet matches not only an IP address field in the rule. The packet is required to match other fields in the rule as well. A randomly generated packet may match some of fields, especially when a field is a subset of a relatively small domain (e.g., Boolean). However, matching all of the fields in the rule with a randomly generated packet is not trivial.

Among clause-level mutation operations, $Bound_1$ and $Bound_1$- achieves the highest mutant-killing ratios over RCT, RCF, $CREV$, and $CREO$ mutated policies. As $Bound1$ and

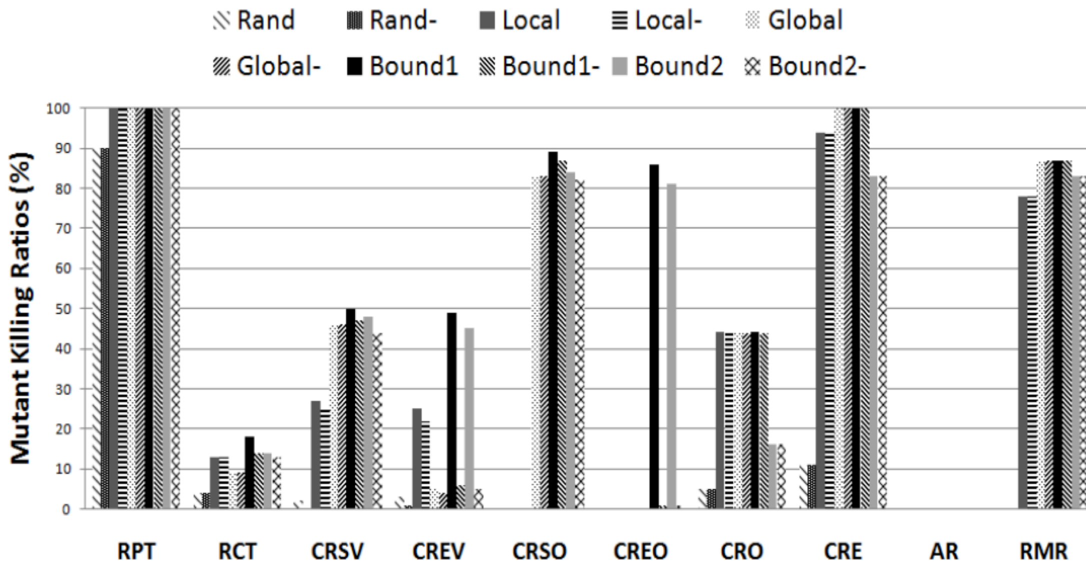

Figure 6.9. Mutant-killing ratios for all subjects by operators.

$Bound1$- evaluate more clauses to true or false, the packet sets are more effective to detect faults in a larger portion of clauses in the policy. $Bound_1$ ($Bound_1$-) and $Bound_2$ ($Bound_2$-) detect more faults in $CRSV$ and $CRSO$ mutated policies. The reason is that a packet in $Bound1$ ($Bound_1$-) and $Bound2$ ($Bound_2$-) are based boundary values in the constraint. Therefore, $Bound_1$ ($Bound_1$-) and $Bound_2$ ($Bound_2$–) are effective to detect faults caused by the change of the boundary value of a clause over other packet sets.

6.5 Limitation

Our approach is based on the concept of policy coverage, which helps test a policy's structural entities (i.e., rules, predicates, and clauses) to check whether each entity is specified correctly. Our approach has the following limitation: our approach is designed for

firewall policies that follow first-match semantic based on the order of rules. In this semantic, each packet is evaluated to the decision of the first rule that the packet matches. For example, if a firewall policy includes a rule that matches a packet, structural entities in subsequent rules of the rule are not covered (i.e., evaluated to either true or false). Therefore, our approach may not be suitable for firewall policies that follow semantics other than first-match semantic for conflict resolution.

6.6 Chapter Summary

We presented a systematic structural testing approach for firewall policies. We defined three types of structural coverage for firewall policies: rule, predicate, and clause coverage criteria. Among the four proposed packet generation techniques, the global constraint solving technique often generated packet sets to achieve the highest structural coverage. Generally, our experimental results showed that a packet set with higher structural coverage has higher fault-detection capability (i.e., detecting more injected faults). Our experimental results showed that a reduced packet set (maintaining the same level of structural coverage with the corresponding original packet set) maintains similar fault-detection capability with the original set.

7 Automated Regression Test Selection for Regression Testing for Security Policy Evolution

We present a safe-test-selection approach [29] for regression testing of security policies. Among given initial test cases in access control systems under test, our approach selects and executes only test cases that could expose different policy behaviors across multiple versions of security policies.

7.1 Introduction

With the change of security requirements, developers may modify policies. After the modification, policy authors should verify the given system to determine that this modification is correct and does not introduce unexpected behaviors (i.e., regression faults). Consider that the system's original policy P is replaced with a modified policy P'. The system may exhibit different system behaviors affected by different policy behaviors (i.e., given a request, its evaluated decisions against P and P', respectively, are different) caused by the policy changes. Such different system behaviors are "dangerous" portions where regression faults could be exposed.

Given existing test cases for P, a naive strategy of regression testing is to rerun all existing system test cases. However, rerunning these test cases could be costly and time-consuming, especially for large-scale systems. Instead of this strategy, developers can use regression-test selection before execution of test cases. This regression-test selection selects and executes

only test cases that may expose different behaviors across multiple versions of the policies. This regression-test selection may require substantial cost to select and execute such system test cases. If the cost of regression-test selection and selected test execution is smaller than rerunning all of the initial system test cases, regression-test selection helps reduce overall cost in validating whether the modification is correct. Safety is an important aspect in regression-test selection. A safe approach of regression-test selection selects only test case that may reveal a fault in a modified program [48].

In this chapter, we present a safe approach of regression-test selection to select a superset of fault-revealing test cases, i.e., test cases that reveal faults due to the policy modification.

Our approach includes three regression-test selection techniques: the first one based mutation analysis, the second one based on coverage analysis, and the third one based on recorded request evaluation. The first two techniques establish correlation (i.e., of rules and test cases.

The first technique selects a rule r_i in P and creates P's mutant $M(r_i)$ by changing r_i's decision. This technique selects test cases that reveal different policy behaviors by executing test cases on program code interacting with P and $M(r_i)$, respectively. Our rationale is that, if a test case is correlated with r_i, the test case may reveal different system behaviors affected by modification of r_i in P. However, this technique requires at least 2×n executions of each

test case to find all correlations between test cases and rules where n is the number of rules in *P*.

The second technique uses coverage analysis to establish correlations between test cases and rules by monitoring which rules are evaluated (i.e., covered) for requests issued from program code. Compared with the first technique, this technique substantially reduces cost during the correlation process because it requires execution of each test case once.

The third technique captures requests issued from program code while executing test cases. This technique evaluates these requests against *P* and *P´*, respectively. This technique then selects only test cases that issue requests evaluated to different decisions.

7.2 Example

Figure 7.1 shows an example policy specified in XACML. Due to space limit, we describe only one rule in the policy in a simplified XACML format. Lines 3-12 describe a rule that borrower is permitted to borroweractivity (e.g., borrowing books) book in working days.

7.3 Regression Test Selection Approach

As manual selection of test cases for regression testing is tedious and error-prone, we have developed three techniques to automate selection of test cases for security policy evolution. Consider that program code interacts with a PDP loaded with a policy P. Let P´ denote P's modified policy. Let S_P denote program code interacting with P. For regression-test selection,

```
1 <Policy PolicyId="Library Policy" RuleCombAlgId="Permit-overrides">
2  <Target/>
3    <Rule RuleId="1" Effect="Permit">
4      <Target>
5        <Subjects><Subject> BORROWER </Subject></Subjects>
6        <Resources><Resource> BOOK </Resource></Resources>
7        <Actions><Action> BORROWERACTIVITY </Action></Actions>
8      </Target>
9     <Condition>
10       <AttributeValue> WORKINGDAYS </AttributeValue>
11      </Condition>
12    </Rule>
...
35 </policy>
```

Figure 7.1. An example policy specified in XACML.

our goal is to select T´⊆ T where T is an existing test suite and T´ reveals different system behaviors due to the modification between P and P´.

7.3.1 Test Selection based on Mutation Analysis

Our first technique establishes correlation between rules and test cases based on mutation analysis before regression-test selection.

Correlation between rules and test cases. For rule r_i in P, we create P's rule-decision-change (RDC) mutant $M(r_i)$ by changing r_i's decision (e.g., Permit to Deny). Figure 7.2 illustrates an example mutant by changing the decision of the first rule in Figure 7.1. The technique next executes T on SP and $S_M(r_i)$, respectively, and monitors evaluated decisions. If the two decisions are different for $t \in T$, the technique establishes correlation between r_i and t.

Regression-test selection. This step selects test cases correlated with rules that are involved with syntactic changes between P and P′. In particular, this technique analyzes syntactic difference, *SDiff*, between P and P′ (e.g., a rule's decisions or locations are changed) and identifies rules that are involved in the syntactic difference.

The drawback of this technique is that it requires the correlation step, which could be costly in terms of execution time. This technique executes T for 2×n times where n is the number of rules in P. Moreover, if the policy is modified, the correlation step should be done again for the changed rules. As this regression-test selection is based on *SDiff*, this technique may select rules that may not be involved with actual policy behavior changes (i.e., semantic policy changes).

7.3.2 Test Selection based on Coverage Analysis

To reduce the cost of the correlation step in the preceding technique, our second technique correlates only rules that can be evaluated (i.e., covered) by test cases.

Correlation between rules and test cases. Our technique executes test cases T on S_P and monitors which rules are evaluated for requests issued from the execution of test case $t \in T$. Our technique establishes correlation between a rule r_i and $t_i \in T$ if and only if r_i is evaluated for requests issued from PEPs while executing t_i.

```
1 <Policy PolicyId="Library Policy" RuleCombAlgId="Permit-overrides">
2  <Target/>
3    <Rule RuleId="1" Effect="Deny">
4      <Target>
5        <Subjects><Subject> BORROWER </Subject></Subjects>
6        <Resources><Resource> BOOK </Resource></Resources>
7        <Actions><Action> BORROWERACTIVITY </Action></Actions>
8      </Target>
9     <Condition>
10        <AttributeValue> WORKINGDAYS </AttributeValue>
11      </Condition>
12    </Rule>
...
35 </policy>
```

Figure 7.2. An example mutant policy by changing the first rule's decision (i.e., effect).

Regression-test selection. We use the same selection step in the preceding technique. An important benefit of this technique is to reduce cost in terms of execution of test cases. This technique requires executing T only once. Similar to the preceding technique, this technique finds the modified rules based on *SDiff* between P and P′, which may not be involved with actual policy behavior changes.

7.3.3 Test Selection based on Recorded Request Evaluation

To reduce correlation cost in the preceding techniques, we develop a technique that does not require correlation between test cases and rules. The third technique executes T on S_P. The technique captures and records requests R_rs issued from PEPs while executing T on S_P. For test selection, our technique evaluates R_rs against P and P′. Our technique selects test case $t \in T$ that issues requests engendering different decisions for P and P′.

This technique requires the execution of T only once. Moreover, this technique is useful especially when polices are not available, but only evaluated decisions are available. As different decisions are reflected by actual policy behavior changes (i.e., semantic changes) between P and P´, this technique can select fault revealing test cases more effectively.

7.3.4 Safe Test-Selection Techniques

A test-selection algorithm is safe if the algorithm includes the set of every fault-revealing test case that would reveal faults in a modified version. In our work, the first test-selection technique is safe when a policy uses the first-applicable algorithm. If the policy uses other combining algorithms, we use an approach [35] to convert the policy to its corresponding policy using the first-applicable algorithm. The second and third techniques are safe for any policies specified in XACML. Due to space limit, proof of safety of our three techniques is presented on our project website.

7.4 **Evaluation**

We conducted experiments for evaluating our proposed techniques of regression-test selection. We carried out our experiments on a PC, running Windows 7 with Intel Core i5, 2410 Mhz processor, and 4 GB of RAM. As experimental subjects, we collected three Java programs [45] each interacting with policies written in XACML. The Library Management System (LMS) provides web services to borrow/return/manage books in a library. The Virtual Meeting System (VMS) provides web conference services to organize online meetings. The Auction Sale Management System (ASMS) provides web services to manage

online auction. These three subjects include 29, 10, and 91 security test cases, which target at testing security checks and policies. The test cases cover 100%, 12%, and 83% of 42, 106, and 129 rules from policies in LMS, VMS, and ASMS, respectively.

Instrumentation. We implemented a regression simulator, which injects any number of policy changes based on three predefined regression types. RMR (Rule Removal) removes a randomly selected rule. RDC (Rule Decision Change) changes the decision of a randomly selected rule. RA (Rule Addition) adds a new rule consisting of attributes randomly selected among attributes collected from P. Combination of the three regression types can incur various policy changes.

For our experiments, the regression simulator injects 5, 10, 15, 20, and 25 policy changes, respectively. Our experiments are repeated 12 times to avoid the impact of randomness of policy changes. We measure effectiveness and efficiency of our three techniques by measuring test-reduction percentage, the number of fault-revealing test cases, and elapsed time.

Research questions. We intend to address the following research questions:

• RQ1: How high percentage of test cases (from an existing test suite) is reduced by our test-selection techniques?

• RQ2: How high percentage of selected test cases can reveal regression faults?

• RQ3: How much time our techniques take to conduct test selection?

Results. To answer RQ1, we measure test-reduction percentage (%TR), which is the number of selected test cases divided by the number of existing security test cases. Table 7.1 shows the number of selected test cases on average for each technique. "Regression - m" denotes a group of modified policies where m is the number of policy changes on P. "#S_{MC}", denotes the number of selected test cases on average by our two test-selection techniques, one based on mutation analysis (TS_M) and one based on coverage analysis (TS_C). "#S_R" denotes the number of selected test cases on average by our technique based on recorded request evaluation (TS_R). We observe that TS_R selected a fewer number of test cases than the other two techniques. The reason is that, while TS_M and TS_C select test cases based on syntactic difference, TS_R selects test cases based on actual policy behavior changes (i.e., semantic policy changes). As illustrated in Section 7.3, syntactic difference may not result in actual policy behavior changes.

Figure 7.3 shows the results of test-reduction percentage for our three subjects with modified policies. LMS1 (LMS2), VMS1 (VMS2), and ASMS1 (ASMS2) show test-reduction percentages for our three subjects, respectively, using TS_M and TS_C (TS_R). We observe that our techniques achieve 42%~97%of test reduction for our subjects with 5~25

Table 7.1. The number of selected test cases on average for each policy group.

Subject	Regression-5		Regression-10		Regression-15		Regression-20		Regression-25	
	#S_{MC}	#S_R	#S_{MC}	#S_R	#S_{MC}	#S_R	#S_{MC}	#S_R	#S_{MC}	#S_R
LMS	4.7	4.5	11.0	9.5	12.9	10.2	14.8	13.8	16.8	14.6
VMS	0.1	0.1	0.4	0.2	1.2	0.8	1.6	1.2	1.8	1.1
ASMS	6.6	5.9	10.9	10.0	16.4	14.8	21.3	19.3	22.4	17.2
Avg.	3.8	3.5	7.4	6.6	10.2	8.6	12.6	11.4	13.7	11.0

policy changes. Such test reduction reduces a substantial cost in terms of test-execution time for regression testing.

To answer RQ2, we show the percentage of selected test cases that reveal regression faults. Detection of regression faults is dependent on the quality of test oracles in test cases. The test cases for our three subjects include test oracles, which check correctness of decisions evaluated for all the requests issued from PEPs. Therefore, selected test cases by TS_R would all detect regression faults (caused by semantic policy changes). On average, the percentages of selected test cases that reveal regression faults are 87%, 87%, and 100% for our three techniques TS_M, TS_C, and TS_R, respectively.

To answer RQ3, we compare efficiency by measuring elapsed time of conducting test selection. Table 7.2 shows the evaluation results. For TS_M and TS_C, the results show the elapsed time of correlation ("Cor") and test selection ("Sel"), respectively. For TS_R, the results show the elapsed time of request recording ("Col") and test selection ("Sel"). We observe that correlation (11,714 milliseconds on average) of TS_C takes substantially less time

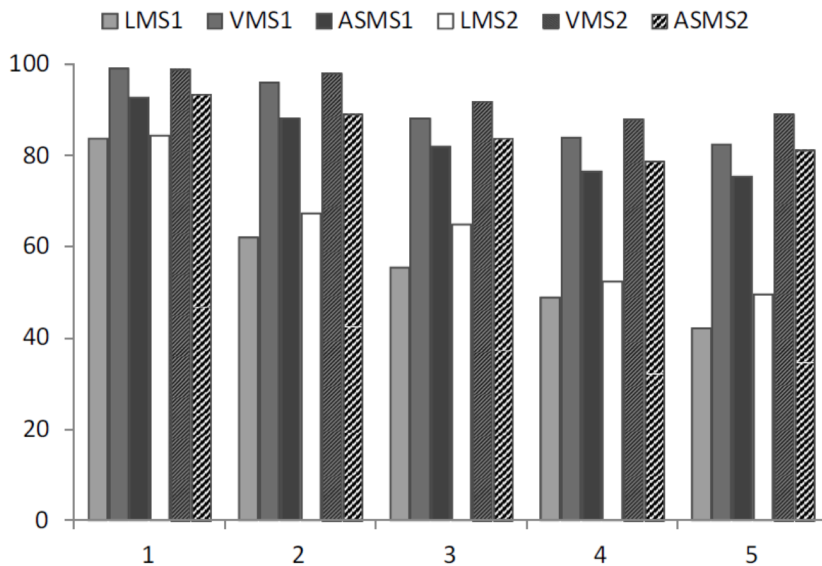

Figure 7.3. LMS1 (LMS2), VMS1 (VMS2), and ASMS1 (ASMS2) show test-reduction percentages for our subjects with modified policies, respectively, using TS_M and TS_C (TS_R). Y-axis denotes the percentage of test reduction. X-axis denotes the number of policy changes on our subjects.

than correlation (69,505 milliseconds on average) of TS_M. The reason is that TS_C executes the existing test cases only once but TS_M executes the existing test cases for 2×n times where n is the number of rules in a policy under test. For total elapsed time by each technique, we observe that the total elapsed time of TS_R is 43 and 8 times faster than that of TS_M and TS_C, respectively.

Threats to validity. The threats to external validity primarily include the degree to which the subject programs, the policies, and regression model are representative of true practice. These threats could be reduced by further experimentation on a wider type of policy-based software systems and a larger number of policies. The threats to internal validity are

Table 7.2. Elapsed time (millisecond) for each test-selection technique.

Subject	TS_M		TS_C		TS_R	
	Cor	Sel	Cor	Sel	Col	Sel
LMS	70,496	4	5,214	4	2,096	2
VMS	19,771	1	7,506	1	1,873	2
ASMS	118,248	11	22,423	11	1,064	21
Average	69,505	5	11,714	5	1,678	8

instrumentation effects that can bias our results such as faults in the PDP, and faults in our implementation.

7.5 Chapter Summary

Our approach could be practical and effective to select test cases for policy-based software systems interacting not only with XACML policies but also with policies specified by other policy specification languages (e.g., EPAL). We make two key contributions. First, we proposed three automatic test-selection techniques in the context of policy evolution. Second, we conducted experiments to assess the effectiveness and efficiency of our three test-selection techniques.

8 Conclusions and Future Work

In this chapter, we present conclusions and future work.

8.1 Conclusions

Faults (i.e., misconfigurations) in security policies may result in tragic consequences such as disallowing an authorized user to access her/his resources and allowing malicious users to access critical resources. Therefore, to improve the quality of security policies in terms of policy correctness, policy authors must conduct rigorous testing and verification.

In this dissertation, we proposed approaches that improve the quality of security policies.

We first present approaches that mine patterns from security policies. Our approaches collected common patterns such that anomalies of those patterns are inspected to determine whether these anomalies expose faults. In addition, we conducted an initial empirical study of policy evolution to answer questions such as how security policies evolve. We conducted two studies:

- Our first study mines patterns with regards to the correlations of attributes showed. Given these mined patterns, our approach has more than 30% higher fault-detection capability than that of the previous related approach, which mines properties based on a classification algorithm. Our results showed that we could reduce the number of

anomalies (that could detect potential faults) for inspection while maintaining the similar level of fault detection capabilities.

- In our second study, we empirically observed that ACPs continue to increase and some of evolution patterns appear to occur more frequently than other evolution patterns. This observation could help build a prediction model for future policy changes with a precision of 50-80%, a recall of 70-90% and an F-measure of 65-75%.

We next developed a systematic structural testing approach based on well-established testing techniques in software engineering. We defined three types of structural coverage for firewall policies: rule, predicate, and clause coverage criteria. We developed automated test-packet generation for achieving high structural coverage. In general, our evaluation results showed that a packet set with higher structural coverage has higher fault-detection capability (i.e., detecting more injected faults).

We developed a test-selection approach, which selects only system test case that may reveal regression faults caused by policy changes. We showed show that our test-selection approach reduces a substantial number of system test cases efficiently.

8.2 **Future Work**

While we introduced several approaches that improve the quality of security policies by helping detecting faults, we do not claim that our approaches could detect every fault in security policies. Future research could improve the effectiveness of our approaches to help

detect other types of faults. For example, our mining approach (in Chapter 5) may detect suspicious activities of policy authors based on frequently used permissions. If policy authors specify rarely used permissions (e.g., “unlink”) before frequently used permissions (e.g., “read”), our approach may give alerts to policy authors for inspection. To reduce false-positives, we could consider cases such that policy authors may use rarely used permissions, which may not be faulty to reduce human efforts. In addition, we consider patterns (in Chapter 4) based on resource, subject-resource, or action-resource attribute values. We could consider other types of patterns for improving fault-detection capabilities.

Second, our approach could be practical and effective to detect real faults in security policies. Future work could empirically investigate how our approaches are effective in terms of fault-detection capability. Real faults may consist of one or several simple faults as described in our evaluation, and may cause a policy’s behaviors to deviate from the policy’s normal behaviors. Detecting real faults often depend on detecting such simple faults, which could be detected effectively by our proposed approaches. Empirical studies could demonstrate the effectiveness of our research approaches.

Third, our work (in Chapter 5) showed that while security by default could be the most secure setting, policy authors continue to change permissions of ACPs in practice. Future work could be a comprehensive study of regressions faults in security policies. Our approach (in Chapter 7) could aid policy authors avoid regression faults that could happen during ACP evolution. However, we do not know how and why policy authors make regression faults. If

we better answer these questions, we could improve our approaches to detect regression faults.

REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. *In Proc.* 20th International Conference on Very Large Data Bases, pages 487–499, 1994.

[2] P. Ammann, J. Offutt, and H. Huang. Coverage criteria for logical expressions. *In Proc.* International Symposium on Software Reliability Engineering, pages 99–107, 2003.

[3] A. Andress. Surviving security: how to integrate people, process, and technology. CRC Press, 2003.

[4] E. Al-Shaer and H. Hamed. Discovery of policy anomalies in distributed firewalls. *In Proc.* 2004 IEEE Conf. on Communications, pages 2605–2616, 2004.

[5] L. Bauer, L. Cranor, R. W. Reeder, M. K. Reiter, and K. Vaniea. Real life challenges in access-control management. *In Proc.* 2009: Conference on Human Factors in Computing Systems, pages 899–908, 2009.

[6] L. Bauer, S. Garriss, and M. K. Reiter. Detecting and resolving policy misconfigurations in access-control systems. *In Proc.* 13th ACM Symposium on Access control Models and Technologies, pages 185–194, 2008.

[7] P. E. Black, V. Okun, and Y. Yesha. Mutation operators for specifications. *In Proc.* 2000 IEEE International Conference on Automated Software Engineering, pages 81–88, 2000.

[8] C. Borgelt. Apriori - Association Rule Induction/Frequent Item Set Mining. http://www.borgelt.net/apriori.html/, 2009.

[9] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change: Research Articles. J. Softw. Maint. Evol., vol. 17, no. 5, pages 309–332, 2005.

[10] B. Caswell, and B. Jay. Snort 2.1 intrusion detection. Syngress, 2004.

[11] P. H. Chia, Y. Yamamoto, and N. Asokan. Is this app safe?: a large scale study on application permissions and risk signals. *In Proc.* 21st international conference on World Wide Web, pages 311–320, 2012.

[12] J. Chomicki, L. Jorge, and N. Shamim. Conflict resolution using logic programming. IEEE Transactions on Knowledge and Data Engineering, 15.1, pages 244-249, 2003.

[13] N. Damianou , N. Dulay , E. Lupu, and M. Sloman. The Ponder policy specification language. *In Proc.* International Workshop on Policies for Distributed Systems and Networks, pages 18–38, 2001.

[14] T. Das, B. Ranjita, and N. Prasad. Baaz: A System for Detecting Access Control Misconfigurations. *In Proc.* USENIX Security Symposium, pages 161-176, 2010.

[15] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: help for the practicing programmer. IEEE Computer, vol. 11, no. 4, pp. 34–41, 1978.

[16] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. *In Proc.* ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 102–112, 2000.

[17] A. El-Atawy, T. Samak, Z. Wali, E. Al-Shaer, F. Lin, C. Pham, and S. Li. An automated framework for validating firewall policy enforcement. *In Proc.* 2007 IEEE

International Workshop on Policies for Distributed Systems and Networks, pages 151–160, 2007.

[18] T. Erl, SOA Design Patterns, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2009.

[19] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. ACM Trans. Inf. Syst. Secur. 4, pages 224–274, 2001.

[20] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. *In Proc.* 27th International Conference on Software Engineering, pages 196–205, 2005.

[21] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. ACM Trans. Softw. Eng. Methodol., pages 184–208, 2001.

[22] D. Hoffman and K. Yoo. Blowtorch: a framework for firewall test automation. *In Proc.* 2005 IEEE/ACM international Conference on Automated Software Engineering, pages 96–103, 2005.

[23] H. Hu, and G. Ahn. Enabling verification and conformance testing for access control model. *In Proc.* 13th ACM symposium on Access control models and technologies, pages 195-204, 2008.

[24] V. Hu, D. F. Ferraiolo and D. R. Kuhn. Assessment of access control systems. US Department of Commerce, National Institute of Standards and Technology, 2006.

[25] G. Hughes and T. Bultan. Automated verification of access control policies. Technical Report 2004-22, Department of Computer Science, University of California, Santa Barbara, 2004.

[26] J. Hwang, T. Xie, F. Chen, and A. X Liu. Systematic structural testing of firewall policies. *In Proc.* 2008 IEEE Symposium on Reliable Distributed Systems, pages 105-114, 2008.

[27] J. Hwang, T. Xie, F. Chen, and A. X Liu. Systematic structural testing of firewall policies. Network and Service Management, IEEE Transactions on, 9(1), 1-11, 2012.

[28] J. Hwang, T. Xie, V. Hu, and M. Altunay. Mining likely properties of access control policies via association rule mining. *In Proc.* 2010 Data and Applications Security and Privacy XXIV, pages 193-208, 2010.

[29] J. Hwang, T. Xie, D. El Kateb, T. Mouelhi, and Y. Le Traon. Selection of regression system tests for security policy evolution. *In Proc.* 27th IEEE/ACM International Conference on Automated Software Engineering, pages 266-269, 2012.

[30] J. Jürjens and G. Wimmel. Specification-based testing of firewalls. *In Proc.* 2001 International Andrei Ershov Memorial Conference on Perspectives of System Informatics, pages 308–316, 2001.

[31] C. F. Kemerer and S. Slaughter. An empirical approach to studying software evolution. IEEE Transactions on Software Engineering, vol. 25, no. 4, pages 493–509, 1999.

[32] M. Koch, L. V. Mancini, and F. P. Presicce. On the specification and evolution of access control policies. *In Proc.* 6th ACM symposium on Access control models and technologies, pages 121–130, 2001.

[33] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski,. Metrics and laws of software evolution - the nineties view. *In Proc.* 4th International Symposium on Software Metrics, IEEE Computer Society, 1997, pages 20–32, 1997.

[34] A. X. Liu. Change-impact analysis of firewall policies. *In Proc.* 2007 European Symposium Research Computer Security, pages 155–170, 2007.

[35] A. X. Liu, F. Chen, J. Hwang, and T. Xie. XEngine: A fast and scalable XACML policy evaluation engine. *In Proc.* International Conference on Measurement and Modeling of Computer Systems, pages 265–276, 2008.

[36] A. X. Liu and M. G. Gouda. Complete redundancy detection in firewalls. *In Proc.* 2005 Annual IFIP Conference on Data and Applications Security, pages 196–209.

[37] A. X. Liu, M. G. Gouda, H. H. Ma, and A. H. Ngu. Non-intrusive testing of firewalls. *In Proc.* 2004 International Computer Engineering Conference, pages 196–201, 2004.

[38] S. W. Lodin and C. L. Schuba. Firewalls fend off invasions from the net, IEEE Spectrum, vol. 35, no. 2, pages 26–34, 1998.

[39] M. R. Lyu and L. K. Y. Lau. Firewall security: policies, testing and performance evaluation. *In Proc.* International Conference on Computer Systems and Applications, pages 116–121, 2000.

[40] E. Martin, J. Hwang, T. Xie, and V. Hu. Assessing quality of policy properties in verification of access control policies. *In Proc.* Annual Computer Security Applications Conference, pages 163–172, 2008

[41] E. Martin and T. Xie. A fault model and mutation testing of access control policies. *In Proc.* International Conference on World Wide Web, pages 667–676, 2007.

[42] E. Martin and T. Xie. Inferring access-control policy properties via machine learning. *In Proc.* 7th IEEE Workshop on Policies for Distributed Systems and Networks, pages 235–238, 2006.

[43] E. Martin, T. Xie, and T. Yu. Defining and measuring policy coverage in testing access control policies. *In Proc.* International Conference on Information and Communications Security, pages 139–158, 2006.

[44] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. *In Proc.* International Conference on Software Maintenance, pages 120-130, 2000.

[45] T. Mouelhi, Y. Le Traon, and B. Baudry. Transforming and selecting functional test cases for security policy testing. *In Proc.* 2nd International Conference on Software Testing, Verification, and Validation, pages 171–180, 2009.

[46] A. Pretschner, F. Sch¨utz, C. Schaefer, and T. Walter. Policy Evolution in Distributed Usage Control. Electron. Notes Theor. Comput. Sci., vol. 244, pages 109–123, 2009.

[47] M. Roesch. Snort: Lightweight Intrusion Detection for Networks. *In Proc.* Large Installation System Administration, vol. 99, pages 229-238, 1999.

[48] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. IEEE Trans. Softw. Eng., 22:529–551, 1996.

[49] P. Samarati and S. Vimercati. Access control: Policies, models, and mechanisms. Foundations of Security Analysis and Design. Springer Berlin Heidelberg, pages 137-196, 2001.

[50] R. S. Sandhu and E. J. Coyne. Role-based access control models. Computer 29.2, pages 38-47, 1996.

[51] H. Seltman. Experimental design and analysis. Pittsburgh, PA: Carnegie Mellon University, 2012.

[52] S. Smalley. Configuring the SELinux policy. NAI Labs Rep: 02-007, 2002.

[53] S. D. Stoller, P. Yang, C. amakrishnan, and M. I. Gofman. Efficient policy analysis for administrative role based access control. *In Proc.* 14th ACM Conference on Computer and Communications Security, pages 445–455, 2007.

[54] J. Trent. Managing access control complexity using metrices. *In Proc.* 6th ACM symposium on Access control models and technologies, pages 131-139, 2001

[55] D. Truog, J. Bernoff, T. Ritter, and H. Goldman. Centralize Access Control Now. Cambridge (MA): Forrester Research, 1999.

[56] J. Vaidya, A. Vijayalakshmi, and G. Qi. The role mining problem: finding a minimal descriptive set of roles. *In Proc.* 12th ACM symposium on Access control models and technologies, pages 175-184, 2007.

[57] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Permission evolution in the Android ecosystem. *In Proc.* 28th Annual Computer Security Applications Conference, pages 31–40, 2012.

[58] G. Wimmel and J. Jürjens. Specification-based test generation for security-critical systems using mutations. *In Proc.* 2002 International Conference on Formal Engineering Methods, pages 471–482, 2002.

[59] A. Wool. A quantitative study of firewall configuration errors, Computer, vol. 37, no. 6, pages 62–67, 2004.

[60] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra. FIREMAN: a toolkit for FIREwall Modeling and Analysis. *In Proc.* 2006 IEEE Symposium on Security and Privacy, pages 199–213, 2006.

[61] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. ACM Comput. Surv., vol. 29, no. 4, pages 366–427, 1997.

[62] Constraint Solver Z3, http://research.microsoft.com/projects/z3/, 2014

[63] Example Policies. http://www.cs.brown.edu/research/plt/software/margrave/versions/01-01/examples/college, 2014

[64] Koji, http://arm.koji.fedoraproject.org/koji/packageinfo?packageID=8645, 2013

[65] Virtual Computing Lab, http://vcl.apache.org/, 2013

[66] Margrave Policy Analysis Tool, http://www.margrave-tool.org/, 2014

[67] A. Oldehoeft, Foundations of a Security Policy for Use of the National Research and Educational Network, NIST special publication 4734, 1992.

[68] OASIS eXtensible Access Control Markup Language (XACML), http://www.oasis-open.org/committees/xacml/, 2013.

[69] Ponder2 Policy Analysis Tool, http://ponder2.net/, 2014

[70] Ponemon Institute, 2010 Access Governance Trends Survey

[71] Security and Privacy Controls for Federal Information Systems and Organizations NIST Special Publication 800-53, 2013

[72] SELinux Reference Policy Repository, https://git.fedorahosted.org/git/selinux-policy.git, 2013

[73] Snort Rules, http://www.snort.org/snort-rules/, 2013

[74] A. H. Watson, T. J. McCabe, and D. R. Wallace. Structured testing: A testing methodology using the cyclomatic complexity metric. NIST special Publication 500, no. 235, pages 1-114, 1996.