UNIVERSITÉ DE MONTRÉAL

FINDING DIFFERENCES IN PRIVILEGE PROTECTION AND THEIR ORIGIN IN ROLE-BASED ACCESS CONTROL IMPLEMENTATIONS

MARC-ANDRÉ LAVERDIÈRE-PAPINEAU
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
AVRIL 2018

FINDING DIFFERENCES IN PRIVILEGE PROTECTION AND THEIR ORIGIN IN ROLE-BASED ACCESS CONTROL IMPLEMENTATIONS

DEDICATION

In memory of my father, Jean-Marie Laverdière. He was kept from seeing this accomplishment he dreamt about.

ACKNOWLEDGEMENTS

Get wisdom, get understanding; do not forget my words or turn away from them. Do not forsake wisdom, and she will protect you; love her, and she will watch over you. The beginning of wisdom is this: Get wisdom. Though it cost all you have, get understanding. Cherish her, and she will exalt you; embrace her, and she will honor you. She will give you a garland to grace your head and present you with a glorious crown. (Proverbs 4:5-9 [NIV])

I first thank and praise the Almighty. Counterintuitively, this thesis may be classified as a miracle. It is no stretch to say that this PhD became a possibility through prayer, and was completed through prayer. I can't count the number of times He empowered me to overcome the urge to give up along the way.

Coming back to Earth, I am reminded of the old proverb, "It takes a village to raise a baby." My studies have made me aware of the following variant: 'it takes a village to train a PhD."

I am very grateful for my family's constant support. My wife, Swapna, my daughter, Viviane-Nishika, and my mother, Lise Papineau, have been great cheerleaders. This is especially true of my wife, Swapna. It is a shame that her name will not appear anywhere on the diploma. I must also give an honourable mention to Viviane-Nishika. She dragged me away from the computer to play with her and give me well-needed infusions of joy.

My professor, Ettore Merlo, deserves many accolades. It is his involvement that allowed me to get unstuck and reach the finish line. While I have learned a lot in his graduate class, I learned a lot more about science and research through our interactions. The breadth of his knowledge (both in Software Engineering and beyond) never ceases to amaze me.

I also thank my labmates past and present for the friendship, learning, and many laughs over the years. I was able to achieve what I did because of the foundation built by Dominic Letarte, Dominic Gauthier and Thierry Lavoie. I must acknowledge the friendships of Théotime Menguy, Mathieu Mérineau and Nicolas Cloutier, who also made the lab a warm place to work in.

I am grateful for the professors at my *alma mater*, Concordia University, for laying a good foundation on which I was able to build. I especially want to note the mentorship I received from professors Mourad Debbabi, Peter Grogono and Terry Fancott.

Many friends and church members encouraged me and prayed along the way. While their contribution appears insignificant, it adds up! I especially want to acknowledge Tai Ngnoma and Karen Gold, who reviewed my writing for part of this thesis.

Last but definitely not least, I want to acknowledge and thank two instrumental persons in Tata Consultancy Services (TCS), my employer. Sitaram Chamarty has been my boss since 2008 and backed up my efforts to pursue a PhD. K Ananth Krishnan is TCS' Chief Technology Officer (CTO) and he personally signed off and followed up on this project. The fact that someone so senior gave attention to the efforts of someone so junior speaks volumes about his management style and approach to R&D. Both gave me unconditional support on this long journey, and allowed me to work full-time on my research project. I'll forever be grateful for the opportunity I have had.

RÉSUMÉ

Les applications Web sont très courantes, et ont des besoins de sécurité. L'un d'eux est le contrôle d'accès. Le contrôle d'accès s'assure que la politique de sécurité est respectée. Cette politique définit l'accès légitime aux données et aux opérations de l'application. Les applications Web utilisent régulièrement le contrôle d'accès à base de rôles (en anglais, « Role-Based Access Control » ou RBAC). Les politiques de sécurité RBAC permettent aux développeurs de définir des *rôles* et d'assigner des *utilisateurs* à ces rôles. De plus, l'assignation des *privilèges* d'accès se fait au niveau des rôles.

Les applications Web évoluent durant leur maintenance et des changements du code source peuvent affecter leur sécurité de manière inattendue. Pour éviter que ces changements engendrent des régressions et des vulnérabilités, les développeurs doivent revalider l'implémentation RBAC de leur application. Ces revalidations peuvent exiger des ressources considérables. De plus, la tâche est compliquée par l'éloignement possible entre le changement et son impact sur la sécurité (e.g. dans des procédures ou fichiers différents).

Pour s'attaquer à cette problématique, nous proposons des analyses statiques de programmes autour de la protection garantie des privilèges. Nous générons automatiquement des modèles de protection des privilèges. Pour ce faire, nous utilisons l'analyse de flux par traversement de patron (en anglais, « Pattern Traversal Flow Analysis » ou PTFA) à partir du code source de l'application. En comparant les modèles PTFA de différentes versions, nous déterminons les impacts des changements de code sur la protection des privilèges. Nous appelons ces impacts de sécurité des *différences de protection garantie* (en anglais, « Definite Protection Difference » ou DPD). En plus de trouver les DPD entre deux versions, nous établissons une classification des différences reposant sur la théorie des ensembles.

De plus, nous calculons des contre-exemples explicatifs. Ces contre-exemples sont des chemins dans un modèle PTFA démontrant la différence de protection entre deux versions. Pour faciliter leur compréhension, nous les simplifions à l'aide de transformations de graphes.

Nous calculons également les changements affectant la protection (en anglais, « Protection-Impacting Changes » ou PIC) à l'aide des différences entre les modèles PTFA et de la rejoignabilité des graphes. Finalement, nous identifions un sur-ensemble des causes racines des DPD en renversant ces changements.

À l'aide d'une étude d'ensemble de 147 paires de versions du logiciel WordPress, nous découvrons que 56% des paires de versions n'ont pas de DPD. Dans les 44% des paires de versions restantes, nous trouvons que seulement 0.30% du code source est affecté par les DPD en moyenne. Dans cette population, nous avons également découvert que les catégories de DPD les plus présentes sont les gains complets ($\approx$ 41%), les pertes complètes ($\approx$ 18%) et les substitutions ($\approx$ 20%). De plus, nous découvrons que les contre-exemples sont généralement courts, avec une longueur médiane de 88 états. L'exemple médian est contenu dans un fichier, et traverse une seule frontière de fonction. Nous avons observé les PIC avec une étude d'ensemble de deux applications Web. Cette étude couvre respectivement 220 et 192 paires de versions pour les logiciels WordPress et MediaWiki. Les PIC sont présents dans 87/220 (40%) et 42/192 (22%) des paires de versions. Dans ces paires de versions, la médiane relative des PIC est d'environ 27% et 14% des changements de code, respectivement.

De plus, nous observons que les causes racines de DPD sont toutes identifiées pour 87% à 93% des paires de versions.

L'identification automatique des DPD, leur classification, les caractéristiques de leurs contre-exemples explicatifs et de leurs PIC peuvent aider les développeurs à revalider l'implémentation RBAC de leurs applications. De plus, ces contributions peuvent faciliter les examens de sécurité, la vérification et validation d'application, les tests et la réparation des failles.

Notre approche peut être intégrée dans les processus de développement des logiciels. En effet, elle pourrait servir comme système d'alarme protégeant contre les implémentations RBAC fautives. De plus, nous envisageons une intégration de nos outils dans les environnements de développement intégrés. Cette intégration permettrait aux développeurs d'analyser interactivement leurs logiciels au besoin.

ABSTRACT

Web applications are commonplace, and have security needs. One of these is *access control*. Access control enforces a security policy that allows and restricts access to information and operations. Web applications often use Role-Based Access Control (RBAC) to restrict operations and protect security-sensitive information and resources. RBAC allows developers to assign users to various *roles*, and assign *privileges* to the roles.

Web applications undergo maintenance and evolution. Their security may be affected by source code changes between releases. Because these changes may impact security in unexpected ways, developers need to revalidate their RBAC implementation to prevent regressions and vulnerabilities. This may be resource-intensive. This task is complicated by the fact that the code change and its security impact may be distant (e.g. in different functions or files).

To address this issue, we propose static program analyses of definite privilege protection. We automatically generate privilege protection models from the source code using Pattern Traversal Flow Analysis (PTFA). Using differences between versions and PTFA models, we determine privilege-level security impacts of code changes using *definite protection differences (DPDs)* and apply a set-theoretic classification to them. We also compute explanatory counter-examples for DPDs in PTFA models. In addition, we shorten them using graph transformations in order to facilitate their understanding. We define *protection-impacting changes (PICs)*, changed code during evolution that impact privilege protection. We do so using graph reachability and differencing of two versions' PTFA models. We also identify a superset of source code changes that contain root causes of DPDs by reverting these changes.

We survey the distribution of DPDs and their classification over 147 release pairs of WordPress, spanning from 2.0 to 4.5.1. We found that code changes caused no DPDs in 82 (56%) release pairs. The remaining 65 (44%) release pairs are security-affected. For these release pairs, only 0.30% of code is affected by DPDs on average. We also found that the most common change categories are *complete gains* ($\approx$ 41%), *complete losses* ($\approx$ 18%) and *substitution* ($\approx$ 20%).

We also surveyed the same versions of WordPress for explanatory counter-examples of privilege protection losses. We computed over 14,000 explanatory counter-examples in 31 release pairs. Our results show that counter-examples are typically short and localized. The median example spans 88 statements, crosses a single function boundary, and is contained in the same file.

We surveyed two PHP web applications for PICs, WordPress and MediaWiki, over 220 and 192 release pairs, respectively. We found PICs in only respectively 87/220 (40%) and 42/192 (22%) release pairs. For these release pairs, median PICs are approximately 27% and 14% of the code changes.

We also found that all root causes of protection differences for a release pair were conservatively identified in 87% to 93% of examined pairs.

The automated identification and classification of DPDs, the characteristics of explanatory counter-examples, and PICs may help developers to focus their efforts more efficiently during security reviews, verification, validation, testing, and repairs.

Our approach could be integrated in secure software development process as an early warning system against invalid RBAC implementations. In addition, our approach could be used within Integrated Development Environments (IDE) and allow on-demand analyses.

TABLE OF CONTENTS

DEDICATION . . . iii

ACKNOWLEDGEMENTS . . . iv

RÉSUMÉ . . . vi

ABSTRACT . . . viii

TABLE OF CONTENTS . . . x

LIST OF TABLES . . . xv

LIST OF FIGURES . . . xvi

LIST OF SYMBOLS AND ABBREVIATIONS . . . xix

LIST OF APPENDICES . . . xx

CHAPTER 1 INTRODUCTION 1
- 1.1 Definitions and Concepts . . . 1
  - 1.1.1 Least Privilege . . . 1
  - 1.1.2 Access Control . . . 2
  - 1.1.3 Web Application Security . . . 5
  - 1.1.4 Static Program Analysis . . . 5
  - 1.1.5 Code Changes and Protection Differences . . . 8
  - 1.1.6 Root Cause Analysis . . . 10
- 1.2 Problem Statement . . . 10
- 1.3 Research Objectives . . . 11
- 1.4 Contributions . . . 12
- 1.5 Thesis Structure . . . 13

CHAPTER 2 CRITICAL LITERATURE REVIEW 14
- 2.1 Detection of SQL Injection and Cross-Site Scripting . . . 14
  - 2.1.1 Taint Analysis . . . 14
  - 2.1.2 String Analysis . . . 15
- 2.2 Slicing . . . 16

2.3 Evolution Classifications 19
2.4 RBAC Analysis 19
2.4.1 Analysis of RBAC Policies 19
2.4.2 Analysis of RBAC Implementations 20
2.4.3 Detection of Access Control Changes 21
2.5 Access Control Evolution Studies 21
2.6 Analyses Leveraging Program Differences 22
2.7 Change Impact Analysis 23
2.8 Root Cause Analysis 23

CHAPTER 3 RESEARCH PROJECT AND THESIS ORGANIZATION 24
3.1 Research Project 24
3.2 Terminology 25

CHAPTER 4 ARTICLE 1: CLASSIFICATION AND DISTRIBUTION OF RBAC PRIVILEGE PROTECTION CHANGES IN WORDPRESS EVOLUTION 27
4.1 Introduction 28
4.2 Background 28
4.3 Methodology 29
4.3.1 Privilege Protection Changes 29
4.3.2 Pattern Traversal Flow Analysis 30
4.3.3 Privilege Protection Change Categories 30
4.4 Results 32
4.4.1 Distribution of Privilege Protection in WordPress 32
4.4.2 Distribution of Privilege Protection Changes in WordPress 34
4.4.3 Distribution of Privilege Protection Change Classification 34
4.5 Discussion 37
4.6 Related Work 38
4.7 Threats to Validity 38
4.8 Conclusion and Future Work 39

CHAPTER 5 ARTICLE 2: COMPUTING COUNTER-EXAMPLES FOR PRIVILEGE PROTECTION LOSSES USING SECURITY MODELS 41
5.1 Introduction 42
5.2 Methodology 46
5.2.1 Pattern Traversal Flow Analysis 46

5.2.2 Privilege Protection Loss . . . 47
5.2.3 Computing Counter-Examples . . . 48
5.2.4 Release Pair Selection . . . 48
5.3 Results . . . 55
5.4 Discussion . . . 58
5.5 Related Work . . . 59
5.6 Threats to Validity . . . 61
5.7 Conclusion and Future Work . . . 62

CHAPTER 6 ARTICLE 3: DETECTION OF PROTECTION-IMPACTING CHANGES DURING SOFTWARE EVOLUTION 63
6.1 Introduction . . . 64
6.2 Background . . . 66
6.2.1 Pattern Traversal Flow Analysis . . . 66
6.2.2 Definite Protection Differences . . . 67
6.3 Method . . . 68
6.3.1 Added and Deleted Edges . . . 68
6.3.2 Appropriately Protected Paths . . . 69
6.3.3 Definition of Protection-Impacting Changes . . . 71
6.4 Example . . . 72
6.5 Experiment Design . . . 73
6.5.1 Approach . . . 73
6.5.2 Data Collection and Processing . . . 74
6.5.3 Measures . . . 74
6.6 Experimental Results . . . 75
6.7 Discussion . . . 79
6.8 Related Work . . . 80
6.9 Threats to Validity . . . 82
6.10 Conclusion and Future Work . . . 83

CHAPTER 7 ARTICLE 4: RBAC PROTECTION-IMPACTING CHANGES : A CASE STUDY OF PHP APPLICATION EVOLUTION 85
7.1 Introduction . . . 85
7.2 Background . . . 90
7.2.1 Access Control Methods . . . 90
7.2.2 DPD Root Cause Analysis . . . 91
7.2.3 Control Flow Graphs . . . 92

7.2.4 Definite Privilege Protection . . . 92
7.2.5 Pattern Traversal Flow Analysis . . . 93
7.2.6 Definite Protection Differences . . . 95
7.3 Method . . . 97
7.3.1 Reachable PTFA Change Detector . . . 98
7.3.2 Reachable Edges Finder . . . 100
7.3.3 Formal Definition of Protection-Impacting Changes . . . 105
7.4 Experiment Design . . . 105
7.4.1 RQ1: Reduction of Code Changes to Evaluate . . . 105
7.4.2 RQ2 : PICs containing all DPD root causes . . . 112
7.5 Experimental Results . . . 115
7.5.1 RQ1 : Reduction of Code Changes to Evaluate . . . 115
7.5.2 RQ2 : PICs containing all DPD root causes . . . 123
7.6 Discussion . . . 123
7.6.1 RQ1: Reduction of Code Changes to Evaluate . . . 124
7.6.2 RQ2 : PICs containing all DPD root causes . . . 125
7.6.3 Further Discussions . . . 125
7.7 Related Work . . . 129
7.8 Threats to Validity . . . 131
7.9 Conclusion and Future Work . . . 132

CHAPTER 8 GENERAL DISCUSSION 135
8.1 Discussion on the Research Objectives . . . 135
8.2 Vulnerability Oracles . . . 137
8.3 Secure Software Development Processes . . . 137
8.4 Limitations of the Proposed Solution . . . 138
8.4.1 Precision . . . 138
8.4.2 Program Size and Change Size Metrics . . . 139
8.4.3 Privilege Protection and Code Analyzed . . . 140
8.4.4 Complex Code Patterns . . . 140
8.5 Examples . . . 141
8.6 Implementation Notes . . . 145
8.6.1 The PHP Front-End . . . 145
8.6.2 The ACMA PTFA Analyzer . . . 146
8.6.3 Line Differences . . . 147
8.7 Language Support . . . 147

CHAPTER 9 CONCLUSION AND RECOMMENDATIONS 148
9.1 Summary of Contributions 148
9.2 Advancement to Knowledge 150
9.3 Recommendations for Developers 150
9.4 Future Research 151

REFERENCES 153

APPENDICES 171

LIST OF TABLES

Table 3.1 Correspondence Between Terms 26
Table 5.1 Privilege Protection Loss Examples Examples Summary Statistics . . 56
Table 5.2 Distribution of Function Boundaries Crossed in Privilege Protection Loss Examples 58
Table 5.3 Distribution of Number of Files in Privilege Protection Loss Example 58
Table 6.1 Protection-Impacting Changes Per Release Pair 77
Table 7.1 Survey Summary 115
Table 7.2 Protection-Impacting Changes Per Release Pair (WordPress) 116
Table 7.2 Protection-Impacting Changes Per Release Pair (WordPress – Cont.) 117
Table 7.2 Protection-Impacting Changes Per Release Pair (WordPress – Cont.) 118
Table 7.3 Protection-Impacting Changes Per Release Pair (MediaWiki) 118
Table 7.3 Protection-Impacting Changes Per Release Pair (MediaWiki – Cont.) 119
Table 7.4 Results of Reverting Definite Protection Differences 124
Table E.1 Rules for Reverting Files with Protection-Impacting Changes 182

LIST OF FIGURES

Figure 1.1 Sample Access Control Matrix 2
Figure 1.2 Sample RBAC Access Control Matrix and Group Memberships . . . 3
Figure 1.3 Sample RBAC use in WordPress 5
Figure 1.4 Steps Involved in Static Program Analysis 6
Figure 1.5 Examples Showing the Importance of Analysis Sensitivity 7
Figure 1.6 Reformatted Excerpt of the Patch for CVE-2009-2854 9
Figure 1.7 Reformatted Patch for CVE-2015-5623 9
Figure 2.1 Example of Slicing and Privilege Protection 18
Figure 4.1 Classification Categories Illustrated 31
Figure 4.2 Definitions for the Classification Categories 31
Figure 4.3 Total Vertices with Non-Empty Privilege Protection Set vs Program Size per version 33
Figure 4.4 Evolution of Definitely Protected Vertices' Privilege Protection Set Cardinalities 33
Figure 4.5 Distribution of Total Privilege Protection Changes per Release Pair . 34
Figure 4.6 Violin Plot of Classified Privilege Protection Changes Per Non-Empty Release Pair 35
Figure 4.7 Classified Vertices for All Release Pairs 36
Figure 5.1 Simplified PTFA Model for our Running Example 50
Figure 5.2 Privilege Protection Loss in Our Running Example 51
Figure 5.3 Reversed PTFA Model for our Running Example 52
Figure 5.4 Reversed PTFA Model for Listing 5.2 53
Figure 5.5 WordPress Release Tree Excerpt. 54
Figure 5.6 Cumulative Distribution of Counter-Examples per Release Pair . . . 55
Figure 5.7 Distribution of Privilege Protection Loss Example Lengths 56
Figure 5.8 Cumulative Distribution of Privilege Protection Loss Example Lengths 57
Figure 5.9 Privilege Protection Loss Example Length vs Program Size 57
Figure 6.1 Example CFG and Corresponding PTFA Model 67
Figure 6.2 Example Code, CFG and PTFA Model 72
Figure 6.3 Distribution of Protection-Impacting Lines of Code per Release Pair for security-affected Release Pairs 78
Figure 6.4 Distribution of Protection-Impacting Lines of Code per Release Pair for security-affected Release Pairs 79

Figure 7.1 Reformatted Vulnerability Correction by Adding Privilege Checks . . 88
Figure 7.2 Excerpt of Vulnerability Correction by Moving Privilege Check . . . 89
Figure 7.3 Example CFG and Corresponding Reachable PTFA Model 95
Figure 7.4 Examples of gain-affected and loss-affected CFG Vertices 96
Figure 7.5 Code Change That is Gain-Impacting and Loss-Impacting 97
Figure 7.6 Processing Steps to Determine Protection-Impacting Changes 98
Figure 7.7 Example Paths to definite protection differences 101
Figure 7.8 Example Paths to definite protection differences 102
Figure 7.9 Reachable PTFA Models Corresponding to Figure 7.8 104
Figure 7.10 Processing Pipeline 107
Figure 7.11 Release Tree Excerpt for MediaWiki 108
Figure 7.12 Example Illustrating the Need to Use Heuristics When Projecting Protection-Impacting-Changes to Lines of Code 111
Figure 7.13 Workflow for Reversing Protection-Impacting Changes 113
Figure 7.14 Algorithm for Individual Reversal 114
Figure 7.15 Distribution of Protection-Impacting Lines of Code per Release Pair for Security-Impacted Release Pairs 120
Figure 7.16 Distribution of Protection-Impacting Lines of Code per Release Pair for Security-Impacted Release Pairs 122
Figure 7.17 Example of definite protection difference Caused by Interprocedural Edge Resolution 126
Figure 7.18 Decision Support System for Future Research 128
Figure 8.1 Example of Privilege Checks Which Do Not Confer Definite Privilege Protection 141
Figure 8.2 Reformatted Patch for CVE-2015-5623 142
Figure 8.3 Reformatted Security Patch for CVE-2006-6016 143
Figure 8.4 Reformatted Security Patch for CVE-2007-1893 144
Figure 8.5 Contributed Software Ecosystem 146
Figure A.1 Example of Protection-Impacting Changes 171
Figure A.2 Kinds of Changed Edges 172
Figure A.3 Simplified CFGs of the Example 173
Figure A.4 Simplified PTFA Models of Our Example 174
Figure A.5 Simplified PTFA Models of Our Example 175
Figure C.1 *vertexMap* Construction Algorithm 178
Figure D.1 Release Tree Construction Algorithm 180
Figure F.1 Backwards Reachability Algorithm 184

Figure F.2 Backwards Reachability Algorithm (continued) 185
Figure F.3 Definitions for *bReachable* 186

LIST OF SYMBOLS AND ABBREVIATIONS

ABAC	Attribute-Based Access Control
ACL	Access Control List
ACM	Access Control Matrix
API	Application Programming Interface
ASR	Automated Software Repair
AST	Abstract Syntax Tree
BDD	Binary Decision Diagram
CFG	Control Flow Graph
CIA	Confidentiality, Integrity, Availability
CSRF	Cross-Site Request Forgery
DPD	Definite Protection Difference
IBAC	Identity-Based Access Control
IDE	Interprocedural Distributed Environment Transformer
IDE	Integrated Development Environment
IFDS	Interprocedural Finite Distributed Subset
LOC	Lines of Code
NBAC	Authentication-Based Access Control
NCLOC	Non-Comment Lines of Code
NIST	National Institute of Standards and Technology
PDG	Program Dependence Graph
PIC	Protection-Impacting Change
PLOC	Physical Lines of Code
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PPC	Privilege Protection Change
PTFA	Pattern-Traversal Flow Analysis
RBAC	Role-Based Access Control
RCA	Root Cause Analysis
SOD	Separation of Duties
SQLi	SQL Injection
V&V	Verification and Validation
XSS	Cross-Site Scripting
ZBAC	authoriZation-Based Access Control

LIST OF APPENDICES

Appendix A Example 171
Appendix B Obtaining Source Code for the Case Study 177
Appendix C VertexMap Construction Algorithm 178
Appendix D Release Tree Construction Algorithm 179
Appendix E Revert Rules 181
Appendix F Backwards Reachability Algorithm 183

CHAPTER 1 INTRODUCTION

Web applications are commonly used in a wide range of industries. Their many uses include email, shopping, filing taxes, and interacting with friends and family. Because of the sensitive nature of the data in these applications, they have high security needs. These needs are generally grouped along three axes: confidentiality, integrity and availability – this is called the CIA triad [32].

Access control (also known as authorization) is a method that ensures confidentiality and integrity. It enforces a policy restricting the operations that users may perform. A widely used form of access control is Role-Based Access Control (RBAC). RBAC is an approach to access control that groups the users in groups. Application developers then allow operations based on group memberships.

Despite their best efforts, developers may implement RBAC improperly. When they do so, their application may suffer from access control vulnerabilities. According to a recent industry survey, broken access control is the fifth most important security issue in web applications [146]. The impact of improper access control varies from one application to the next, but it can be considerable. For instance, a vulnerable application may allow unauthorized information disclosure (e.g. unauthenticated users accessing sensitive data, accessing another user's account). In another example, a vulnerable application may allow non-administrator users to perform administrative tasks.

The OWASP Foundation, a leading industry body[1], observes that "access control weaknesses are common due to the lack of automated detection, and lack of effective functional testing by application developers" [146].To address this issue, we present an automated method to detect variations in RBAC protection between versions. When variations are detected, our approach computes additional information to help developers determine if the differences are harmful, and understand why the difference occurred.

1.1 Definitions and Concepts

1.1.1 Least Privilege

A well-designed web application should use access control to enforce the principle of least privilege. This principle is intended to lower the risk of accidental or malicious damage. It

[1]OWASP stands for the Open Web Application Security Project

states that “every program and every user of the system should operate using the least set of privileges necessary to complete the job.” [158]. In other words, this principle means that code should execute with *only* the privileges specified in the security policy.

1.1.2 Access Control

A way to enforce the principle of least privilege is using access control. An access control policy defines what operations *subjects* can perform on which *objects*. In a web application, the subjects are all active entities and objects are the entities that need protection. Subjects include users, processes, and remote services. Objects include files, database rows and sensitive operations. The access control mechanism is responsible for enforcing this policy.

We introduce widely used access control approaches: Identity-Based Access Control (IBAC), Role-Based Access Control (RBAC), Attribute-Based Access Control (ABAC), and authoriZation-Based Access Control (ZBAC). Afterwards, we describe access control implementations in software systems.

Identity-Based Access Control

Identity-Based Access Control [25, 91] is a form of access control specified using the subjects’ and objects’ identity. The access control policies can be specified using an access control matrix (ACM) [24]. For every entry in the ACM, the subject has the specified rights over the object. In Figure 1.1 we give an example ACM, where o_i refers to an object and s_j refers to a subject.

	o_1	o_2		o_n
s_1		read, edit		read
s_2	read, print		$\dots$	
s_3	read, print			
			$\dots$	
s_n	read	read		read, print, edit

Figure 1.1 Sample Access Control Matrix

Since maintaining and storing the ACM for large systems may be problematic, it is possible to project the ACM on either axis.

The projection by column is called access control lists (ACLs) [23]. For each object o in the system, the object is stored with a set of 2-tuples (s, r), where s is a subject and r is a set of rights that s has over o. A subject with no such tuple has no rights over o.

The projection by row is called *capabilities* [23]. For each subject s in the system, there is a set of 2-tuples (o, r), where o is an object and r is the set of rights that s has over o. The use of capabilities requires tamper-proof mechanisms (such as cryptography), since the user could otherwise alter their capability list.

Role-Based Access Control

In the example above (Figure 1.1), subjects s_2 and s_3 have the same rights. This kind of duplication is one of the factors that gave rise to Role-Based Access Control. RBAC users are assigned to groups and permissions are assigned to groups. Thus, users obtain privileges indirectly, through their group memberships. [26, 91, 160]. Using RBAC, our example of Figure 1.1 could thus become as in Figure 1.2. We show the group memberships in Figure 1.2a and the updated ACM in Figure 1.2b. Please note that group membership is an N-to-N association, meaning that a subject may belong to many groups.

User	**Groups**
s_1	g_1
s_2	g_2
s_3	g_3
...	
s_n	g_m

(a) Group Memberships

	o_1	o_2		o_n
g_1		read, edit		read
g_2	read, print		...	
...				
g_m	read	read		read, print, edit

(b) ACM

Figure 1.2 Sample RBAC Access Control Matrix and Group Memberships

Many variants of RBAC have been proposed over the years, and a US government agency, the National Institute of Standards and Technology (NIST), proposed a standard with four variants of RBAC: flat, hierarchical, constrained, and symmetric [160]. Flat RBAC follows the textbook definition of RBAC. Symmetric RBAC allows constrained RBAC, but it includes management tools for reviewing assignments of permissions to roles. We now describe the remaining two RBAC variants.

Hierarchical RBAC enables the organization of groups using a partial order. In this ordering, the greater roles have all the privileges of their lesser roles, in addition to their own privileges.

Constrained RBAC enforces *separation of duties (SOD).* SOD is intended to lower the risk of fraud and accidents by preventing users from gathering too much power. When SOD is required to perform an action, multiple users with different roles must perform the action. For instance, users may not belong to two or more mutually exclusive groups.

Attribute-Based Access Control

An alternative family of access control is ABAC [75, 91]. In this approach, there are no roles with statically assigned privileges. Instead, the access control policy specifies allowed access using the subject's and object's attributes as well as environment conditions. In essence, ABAC policies are first-order logic expressions over attributes and environment conditions. These attributes are assigned to the subject and the objects by the ABAC engine.

Authorization-Based Access Control

A more recent proposal is authoriZation-Based Access Control (ZBAC) [91]. ZBAC is designed for distributed and service-based systems and is akin to capability-based models. Users must authenticate with an authentication service. The latter will provide the user with one or more authorizations. When using a service, the user submits the appropriate authorization with their request. The service only needs to verify the authorization's validity to make an access decision.

Access Control Implementation

Generally speaking, access control mechanisms are built using a Policy Decision Point (PDP) and multiple Policy Enforcement Points (PEPs) [75]. PEPs act as the gatekeepers of sensitive operations. A client attempting a privileged operation interacts with a PEP. The PEP then requests access on behalf of the client to the PDP. Upon receiving this request, the PDP consults the access control policy and either grants or denies the request. If the request is granted, the PEP allows the operation to proceed.

Software developers sometimes implement RBAC differently than we described earlier. For instance, they sometimes allow both users and groups as subjects in the ACM (e.g. Microsoft Windows ACLs). Also, they sometimes define privileges which apply on all objects.

In software, PEPs are typically conditional statements. They gather the context information required by the PDP and execute a call to a stereotypical Application Programming Interface (API). We show an example PEP from WordPress in Figure 1.3. The call to `current_user_can` interacts with the PDP, requesting if the current user holds the `edit_posts` privilege. This API returns a Boolean indicating if the execution may proceed or not. In this case, if the policy disallows the action, it stops the execution of the procedure and returns to the calling context. Please note that, in this example, the privilege is queried with no relation to objects in the system. However, WordPress also allows to request access in relation to objects.

```
if(! current_user_can( 'edit_posts' )){
  return;
}
$last_post_id =(int)get_user_option('dashboard_quick_press_last_post_id');
//...
```

Figure 1.3 Sample RBAC use in WordPress

1.1.3 Web Application Security

As we alluded to earlier, access control is only one dimension of security. Other areas of web application security include authentication, session management, input handling, data protection, error handling, logging, use of encryption, server configuration, etc [144].

The OWASP Foundation publishes guidance documents and security tools for web application developers. One of their flagship project is the OWASP Top Ten, a list of the ten most prevalent web application security issues [146]. They also publish a comprehensive web application security verification standard [144] and a testing guide [143].

Some of the most dangerous web application vulnerabilities are Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF) and SQL Injection (SQLi). These vulnerabilities occur when user input is not sufficiently validated. This user input can then inject malicious scripts into a web page or modify queries to the database.

1.1.4 Static Program Analysis

Static analysis, broadly speaking, is a family of program analyses that do not require the execution of the program. Program analyses can be used to determine many program properties, and they are an integral part of compiler operations. They are essential for many optimizations [6].

Some program analyses are dynamic, meaning that they gather information while running the program. An example of dynamic analysis is to gather runtime invariants [50]. Some hybrid program analyses exist, which combine both static and dynamic analyses. An example of hybrid analysis is recording the uses of reflection during program execution and then using this information to improve the precision of a static analysis [29].

Static program analyses rely on mathematical representations of the program (trees, graphs), which we show in Figure 1.4. Each representation can be transformed into the other using formal rules. Since source code files are character strings, syntactic analysis is initially required to create an abstract syntax tree (AST). Some analyses only need the AST to operate. More powerful analyses will convert the AST into graph form. An important graph for static analyses is the control flow graph (CFG). Unless stated otherwise, CFGs are *intraprocedural*, meaning that each procedure has its own self-contained CFG. More powerful analyses require an *interprocedural* CFG, which is an intraprocedural CFG with an explicit entry point and additional edges for function calls and returns. These additional edges are determined by the call graph construction algorithm. The level of precision of the call graph thus has a major impact on interprocedural analyses.

The static analysis used throughout this thesis expands on interprocedural CFGs. It is called Pattern-Traversal Flow Analysis (PTFA), and we describe it further in Section 1.1.4.

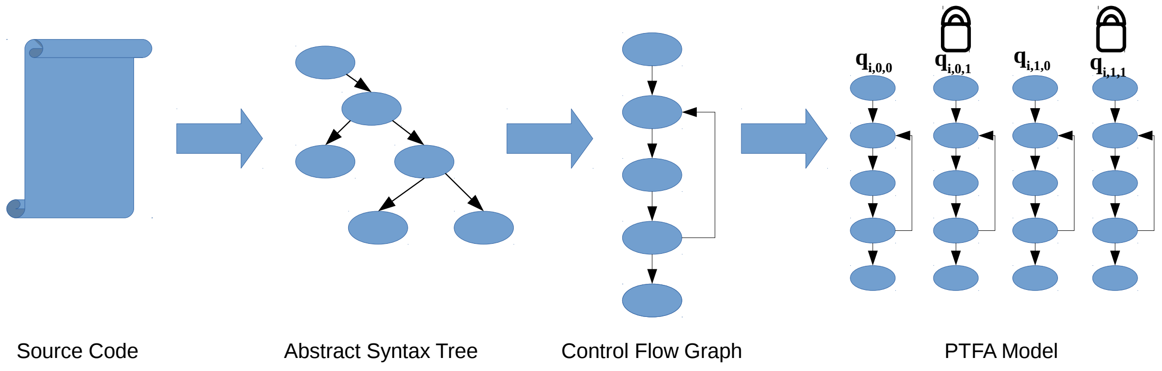


Figure 1.4 Steps Involved in Static Program Analysis

Static analyses have different of levels of sensitivity. Higher sensitivity implies more precise results, but often at the cost of higher analysis time.

Flow sensitivity means that the analysis takes into consideration the statements' order of execution. For instance, consider the code in Figure 1.5a. A flow sensitive analysis would conclude that `$a` = 1, meaning that the code in the "then" branch of the `if` statement is dead code. A flow-insensitive analysis would be unable to determine that this is the case, as it would conclude that `$a` can either be zero or one.

Context sensitivity means that an interprocedural analysis distinguishes between calling contexts [5]. A context insensitive analysis, on the other hand, merges all calling contexts. We illustrate the impact of context sensitivity in Figure 1.5b. Function `foo` has different behavior whether the product of its parameters is positive or negative. Function foo is called using both positive and negative values. A context-sensitive analysis would conservatively conclude that both branches are executable, since it conservatively assumes that the function is called with a mix of positive and negative arguments. However, a context-sensitive analysis would conclude that `$c` is always positive, and that the else branch in `foo` is dead code.

```
$a = 0;
$b = 1;
$a = $b
if ($a == 0){ ... }
```

(a) Flow Sensitivity

```
function foo($a, $b){
  $c = $a * $c;
  if ($c >= 0) //...
  else //...
}
foo(2, 4);
foo(-3, -7);
```

(b) Context Sensitivity

Figure 1.5 Examples Showing the Importance of Analysis Sensitivity

Many frameworks for general interprocedural analysis have been proposed. The traditional bit vector framework can be extended for interprocedural analyses [92]. It is also possible to use other representations, such as Binary Decision Diagrams (BDDs) [114]. In addition, one may translate the interprocedural CFG into Datalog relations and use a Datalog engine [5]. To simplify this process, one may also use a domain-specific query language [114, 124]. It is also possible to use graph reachability in either the Interprocedural Finite Distributed Subset (IFDS) [154] or Interprocedural Distributed Environment Transformer (IDE) [157] frameworks. An IFDS analysis creates an interprocedural graph connecting different program facts. An IDE analysis is a generalization of IFDS, which allows arbitrary transformations of the input domain. However, these analyses are limited to analyses with distributive flow functions. A more general approach uses value contexts [147].

Software Model Checking and PTFA

Model checking, in general terms, is a method used to verify properties over models. These properties are often expressed in temporal logic [86]. For instance, model checking could verify that "processing step s_1 always leads to step s_2". Whenever the property is violated, a model checker normally outputs a *counter-example*, which is a series of steps in the model that lead to the violation of the property.

Model checking can operate on various kinds of models, including models of programs. In this case, it is called *software model checking*. It typically employs static analysis [86].

We use PTFA, a domain-specific Boolean model checking approach. PTFA is designed to validate simple Boolean properties efficiently, but is not designed to validate complex properties. In contrast, traditional model checking approaches are capable of validating arbitrary properties, at the expense of scalability. PTFA models are built from the CFG using graph rewriting rules. For each vertex in the CFG, a PTFA model has up to four reachable states. Each of these states represents a combination of property satisfaction in the local context and in the calling context. In Figure 1.4, we showed a PTFA model with four columns. All states in the same column represent the same property satisfaction. Columns of states that satisfy the property are represented with a padlock icon.

PTFA models can be simplified into *reachable* PTFA models, where only edges and states reachable from the start of the program remain.

The analysis of the Boolean property in PTFA models is flow sensitive. In addition, it has a variant of context sensitivity, which we call *predicate context sensitivity*. Nonetheless, PTFA models are identical to those that would be constructed by a fully context sensitive analysis.

PTFA is suited for analyzing RBAC privilege protection. In this case, it validates the property *has privilege p been verified before executing this statement?* Using PTFA, we obtain *definite privilege protection*, which relates to privilege protection on all execution paths. Specifically, a statement s is definitely protected by privilege p if and only if it is protected by p on all program paths leading to s.

1.1.5 Code Changes and Protection Differences

The term *code change* has a variety of meanings in software engineering. We use this term to mean any addition, suppression or modification to the source code. It has no relationship to version control. We are also concerned with security changes, specifically changes in privilege protection. These are not the same as code changes. To reduce confusion on these terms, we use *protection differences* instead.

We see an example of a code change that causes protection differences in Figure 1.6. This is an excerpt of the patch for a vulnerability affecting WordPress before 2.8.3 and is documented by CVE-2009-2854. The patch adds privilege checks at the beginning of 12 scripts, making their computations definitely protected.

We see another example in Figure 1.7. This is the patch to a vulnerability documented by CVE-2015-5623. In this example, developers added a privilege check for privilege `edit_posts` to one file (line 5). They improved the error handling when the user lacks the `edit_posts` privilege in another (lines 12–16). Finally, they added application-specific logic in a third (line 24). Developers re-validating their application may like to easily tell these kinds of changes apart.

```
  <?php
  /**
  * Edit category form for inclusion in administration panels.
  *
  * @package WordPress
  * @subpackage Administration
  */

+ if ( !current_user_can('manage_categories') )
+   wp_die(__('...'));
```

Figure 1.6 Reformatted Excerpt of the Patch for CVE-2009-2854. Represented in Unified Diff Format. This Change Makes the Computations in The File Definitely Protected with Regards to Privilege `manage_categories`.

```
   // In wp-admin/includes/dashboard.php
   wp_network_dashboard_right_now() {
   function wp_dashboard_quick_press( $error_msg = false ) {
   global $post_ID;
+   if ( ! current_user_can( 'edit_posts' ) ) {
+      return;
+   }
   // ...
   //In wp-admin/post.php
   if ( ! wp_verify_nonce( $nonce, 'add-post' ) )
     $error_msg = __( '...' );
-  if ( ! current_user_can( 'edit_posts' ) )
-     $error_msg = __( '...' );
+  if ( ! current_user_can( 'edit_posts' ) ) {
+     exit;
+  }
   //...
   //In wp-includes/capabilities.php
   case 'edit_post':
   case 'edit_page':
   $post = get_post( $args[0] );
-  if ( empty( $post ) )
+  if ( empty( $post ) ) {
+     $caps[] = 'do_not_allow';
      break;
+  }
   //...
```

Figure 1.7 Reformatted Patch for CVE-2015-5623. Represented in Unified Diff Format. Some Changes in The Patch Make Code Definitely Protected With Regards to Privilege `edit_posts`. One Change Has No Impact on Definite Protection.

1.1.6 Root Cause Analysis

A root cause is “the fundamental reason for the occurrence of a problem” [44]. The link between the root cause and the problem is not always direct, as there may be a causal chain between them. Identifying the root cause thus requires uncovering every step in the causal chain. Root cause analysis may be difficult, but it is beneficial, since properly addressing the root cause of an issue prevents it from reoccurring.

We are interested in the software-level root causes of differences in privilege protection. When a program implements its RBAC policy as privilege checks throughout the code, differences in privilege protection are caused by code changes. However, in practice, only some of the changes affect privilege protection. As we will see, it is possible to identify a set of code changes which contains the root causes of protection differences.

1.2 Problem Statement

Many web applications implement their RBAC policy directly in the source code. To know what changed in the RBAC policy between two versions, we must examine code changes.

Code changes occur during an application’s maintenance and evolution. Software maintenance can be corrective, adaptive, perfective and preventive [83]. We do not have estimates of the maintenance effort dedicated to security in general, nor for RBAC policy maintenance and validation. However, from industry data, we know that corrective and adaptive maintenance represents 30% to 50% of the maintenance effort [46, 109], with security-oriented maintenance activities belonging to the remaining 50% to 70% [38].

Non-security maintenance activities are both functional and non-functional. They include bug-fixing, refactoring, and design changes. Such changes may cause intended [34] and unintended security flaws [97]. Because code changes add, remove and modify execution paths, security flaws can be introduced in different parts of the application than what was modified.

Besides insider threats, RBAC maintenance activities include vulnerability correction, and updating the code to conform to the RBAC policy. To do so, developers may add, move, remove and modify privilege checks. These changes may be related to new features, but sometimes are due to a reorganization of the RBAC policy.

In addition, maintainers abandon, rename, split and merge privileges. An abandoned privilege is a privilege that does not protect any part of the code. A rename means that only the name of a privilege is changed, while the code it protects is identical. A privilege split means that the set of code locations is partitioned in two or more sets, with each set protected by a different privilege. A privilege merge is the reverse of a split. The code protected by two or more privileges becomes protected by one privilege.

Because of the risk of vulnerabilities introduced during maintenance, developers should revalidate their RBAC implementation regularly [97]. After completing maintenance activities, developers are likely to ask *is the protection different?* Answering this question is non-trivial and time-consuming, since it is difficult to manually determine which security properties hold at every point in the program. When privilege protection differs, developers are likely to ask *why is it different?* Answering this question is also challenging for the same reasons.

1.3 Research Objectives

Our thesis is:

Thesis
Using static program analysis, we can automatically (1) identify differences in definite privilege protection between two versions of an application, (2) classify these differences in explanatory categories, (3) compute explanatory counter-examples justifying the differences, (4) determine which code changes are responsible for these differences and (5) conservatively identify all root causes for these differences in most cases.

The overarching research objective in this thesis is to facilitate the revalidation of the RBAC policy's implementation.

Specifically, we cover the following five specific research objectives.

- **RO1:** *Determine which parts of the program have different definite privilege protection, compared to the last version.*
- **RO2:** *Determine how the protection differs.*
- **RO3:** *Given a change in definite protection, justify the outcome.*
- **RO4:** *Determine which code changes are responsible for protection differences.*
- **RO5:** *Investigate the relationship between the code changes identified in* ***RO4*** *and root causes of protection differences.*

1.4 Contributions

Our major contributions are the following:

- We define Definite Protection Differences (DPDs), which is code common to two versions that have different definite privilege protection.
- We define a set-theoretic classification of definite protection differences (DPDs).
- We define explanatory counter-examples for definite protection changes, which are application paths between the start of the program and the security-affected code.
- We propose graph transformations on PTFA models to make explanatory counter-examples easier to understand.
- We define protection-impacting changes (PICs), which are code changes responsible for DPDs.
- We investigate whether PICs contain the root causes of DPDs by reverting them.
- We develop processing steps and implemented programs to compute DPDs, explanatory counter-examples and PICs.
- We experimentally evaluate these contributions with surveys on popular open source applications. Our smallest survey consists of 147 release pairs of WordPress. Our largest survey covers 212 release pairs of WordPress and 192 release pairs of MediaWiki.

Our findings are also telling:

- Most release pairs have no DPDs. When present, less than one percent of the program is security-affected.
- The most common categories of protection changes are complete gain, complete loss and substitution.
- Explanatory counter-examples are fairly short and local, with a median length of less than 100 states.
- Protection-impacting changes are typically less than a third of the code changes.
- Protection-impacting changes contain all root causes nearly 90% of the time

We additionally envision the impact of our approaches on software development processes.

1.5 Thesis Structure

This thesis is organized as follows. In Chapter 2, we survey the related literature. In Chapter 3, we describe our research project and how the articles connect together. In Chapters 4 to 7, we include the articles as they were published or, in the case of one unpublished article, as it was submitted. In Chapter 8, we discuss our findings at greater length and bring concluding remarks in Chapter 9. Please note that the article included in Chapter 7 was submitted with appendices. In accordance with the thesis writing regulations of Polytechnique Montréal, they have been moved to this thesis's appendices.

CHAPTER 2 CRITICAL LITERATURE REVIEW

We now introduce and discuss the literature related to vulnerability detection in web applications (taint and string analysis). We then do the same for classifications of software evolution. Then, we describe methods for the analysis and comparison of RBAC policies and implementations. Finally, we present RBAC evolution studies and non-security analyses that rely on program differences.

2.1 Detection of SQL Injection and Cross-Site Scripting

We introduce major contributions to detect XSS, CSRF and SQLi vulnerabilities in web applications. This introduction is purposefully incomplete. Inquisitive readers may wish to read recent surveys on vulnerability detection [62, 118].

2.1.1 Taint Analysis

Taint analysis determines if data from a *source* can reach a *sink*, which results in an exploitable vulnerability. The vulnerability is avoided by use of a *sanitizer*, a function which transforms the data into a form that will not create any vulnerabilities. Taint analysis requires an *a priori* knowledge of which APIs constitute sources, sinks, and sanitizers, as well as which sanitizers are appropriate for which sinks. Furthermore, taint analysis assumes that sanitizers are perfect, which is not always the case in practice [72]. Taint analysis can be done using static, dynamic and hybrid approaches.

Static Taint Analysis

Static taint analysis approaches often use classic flow analysis [21, 78, 88, 89], slicing [171, 175] and IFDS [15, 176] approaches.

It is also possible to find taint violations using a Datalog engine [123, 124], using the Object Constraint Language (OCL) [12] and information flow type systems [77]. In addition, it is possible to use symbolic execution to lower false positives [16].

Dynamic and Hybrid Taint Analysis

Dynamic taint analysis approaches can induce significant overhead. To reduce this overhead, they rely on various methods and static analyses to determine where to inject their instrumentation [22, 37, 40, 43, 67, 133].

A family of hybrid approaches embed runtime monitors. For some approaches, this monitor will halt program execution or execute recovery actions if a tainted flow reaches a sink [95, 122, 132]. In other approaches, the monitor compares the structure of the executed query against its expected structure [68, 127]. Other hybrid approaches generate test cases intended to trigger the monitor [79, 164].

Another group of hybrid taint analyses use symbolic analysis [129, 130]. These approaches inject instrumentation into the program that records relevant events, which are later analyzed using symbolic analysis.

Taint analysis is designed to track data flow between arbitrary program points. The predicate it handles is "variable v at location l contains data that originates from a taint source". There are two major differences between PTFA and taint analysis with regards to privilege protection. First, PTFA considers predicate satisfaction over execution paths, whereas taint analysis considers predicate satisfaction over propagation paths. Privilege protection is predominantly a control flow issue, meaning that PTFA is a more natural choice to analyze it. Second, PTFA can compute possible and definite predicate satisfaction, whereas taint analysis only considers possible predicate satisfaction. In other words, taint analysis only reports that there exists a tainted data propagation path. This is well-suited for finding some kinds of vulnerabilities, such as XSS, CSRF and SQLi, but not for privilege protection. In addition, taint analysis operates with a specification of which functions are taint sources and sinks. In contrast, applications often have no specification of their intended privilege protection, meaning that taint analysis would benefit little from an evolutionary approach.

Taint analysis bears a few similarities with the contributions in this thesis. Like taint analysis, explanatory counter-examples provide an interprocedural path of interest. Also taint analysis also identify the root causes of the vulnerability whenever it identifies all propagation paths between sources. These similarities are nonetheless superficial, for the reasons mentioned above. In addition, taint analysis does not take code changes in consideration.

2.1.2 String Analysis

String analysis determines what values a string may contain. It does so by building a formal model that summarizes value constraints and transformation operations in the program. To determine the presence of a vulnerability, the analysis engines compare these models with models of acceptable input.

Many string analyses use finite state *transducers* [128]. Transducers are finite state automata that also generate strings, a model suitable for string transformation functions. Some string analysis approaches compute operations on a transducer using SMT solvers [56, 71, 72]. SMT-based approaches have some undecidability constraints. However, the typical string operations in JavaScript and PHP are in a decidable form, or can be automatically converted into one [56].

Grammar-based string analysis [41, 181] constructs a context-free grammar from the program's flow graphs. This grammar construction may take advantage of transducers [181]. Recent advances aim at making tree automata and transducers more general and succinct using symbolic alphabets [45]. In addition, recent advances take into consideration the output context to analyze legacy applications [170].

Other applications of string analysis include sanitizer generation and placement [121, 161, 189, 190] ; program output verification [159] and the detection of parameter tampering vulnerabilities [27]. String analysis also allows the detection of validation inconsistencies between clients and servers [10] and their repair [11].

String analysis is complementary to PTFA to determine privilege protection. This is because developers sometimes use string operations (e.g. concatenation) and conditional expressions to determine the value of the privilege to verify. We describe these practices in section 8.4.4. String analysis, when used to detect vulnerabilities such as XSS, CSRF and SQLi, operates with a specification of acceptable input for sensitive functions. As such, it would benefit little from an evolutionary analysis.

Like to taint analysis, string analysis is somewhat similar to our contributions, as it may identify the root cause of a vulnerability and indicate the propagation path to it. Nonetheless, string analysis is very different from the contributions in this thesis due to its lack of applicability for privilege protection and the fact that it does not consider code changes.

2.2 Slicing

Program slicing [183, 184] extracts a subset of a program, called *program slice*, that encompasses control and data flow related to a statement of interest (often called *seed*). The computed slice preserves the program's behaviour at the seed's point with respect to the slicing criterion. Finding a minimal slice is unsolvable in the general case.

Backward slicing computes a slice from the start of the program to the seed, whereas forward slicing computes all statements that have a control or data dependency from the seed. Backward program slices may be executable in some approaches. A slice is executable when the extracted subprogram could be executed as-is and obtain the same behavior, as far as the seed is concerned. Not all slicing approaches compute executable slices. This property is useful for model checking, as slices can be analyzed in lieu of the original program, providing more scalability [94].

One static interprocedural slicing approach operates on the System Dependence Graph (SDG), which connects multiple Program Dependence Graphs (PDGs) with additional edges [73]. The construction of the SDG is quadratic with regards to the number of predicates and assignments in procedures and the number of procedures. The computation of the slice itself is linear with regards to the size of the SDG.

Slicing variants have been proposed in order to obtain smaller slices. Thin slicing finds a subset of a traditional slice by retaining statements of greater interest to the user. These are *producer statements*, which "help compute and copy a value to the seed" [169]. Thin slices are not executable, but they have between 3.3 and 9.4 times fewer statements than a traditional slice [169]. Fine slicing [1] allows a user to specify non-contiguous seeds. It also relies on user input to ignore unwanted control and data dependencies. It uses semantic restoration to ensure that fine slices are executable, which requires oracle-based semantics to supply missing values. The immediate application of fine slicing is the *Extract Computation* refactoring. An issue with thin slicing is that properties held in the program may not hold on the sliced program. Value slicing addresses this issue [94]. It eliminates statements that only relate to the seed's reachability, while keeping statements influencing the values of the variables in the verified property. This addition over thin slicing ensures that properties verified on the sliced program hold on the original program and offer scalability gains over traditional backward slicing.

The original formulation of slicing called for static analysis, but dynamic [2] and hybrid variants have been proposed [174]. Program slicing has multiple uses, including debugging, program understanding [169], refactoring [1], and taint analysis [175].

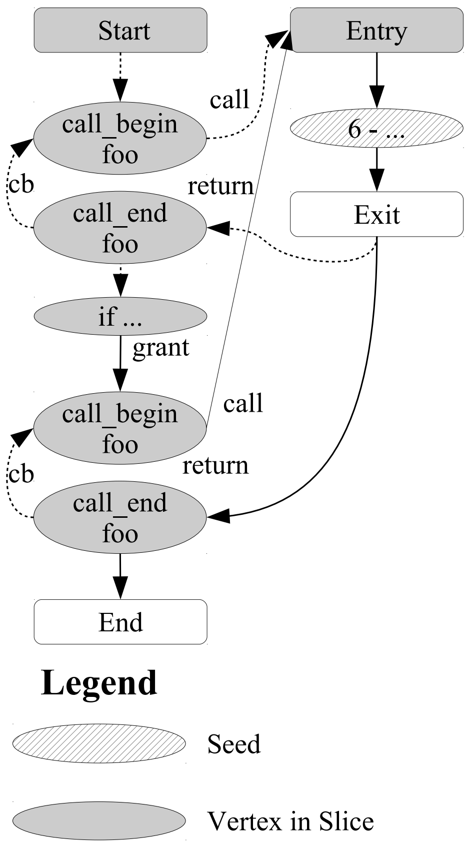
Despite its wide range of applications, slicing, *per se*, would not be appropriate to determine privilege protection due to slicing's limitations and its complexity.

Inherently, slicing does not take property satisfaction in consideration. For instance, consider the example in Figure 2.1a and imagine a backwards slice of a seed statement in function `foo`. The slice in Figure 2.1b includes a privilege verification routine. However, its presence informs that, at best, there may be a protected path between the start of the program and

```
function foo(){
// assumes privilege p1 held
}

foo();
if (!current_user_can('p1'))
  die();
foo();
```

(a) Example of Possibly Protected Code



(b) Corresponding Simplified CFG and Slice

Figure 2.1 Example of Slicing and Privilege Protection

the seed. The slice does not tell us, *per se*, whether the relationship between the verification routine is a control or a data dependency. In addition, it cannot offer any guarantees about the satisfaction of a property on all paths leading to the seed statement. In other words, backward slicing would provide insufficient information to determine definite protection. This is consistent with our example, where the code in function `foo` is protected only on one path. Nonetheless, slicing would provide a reduced graph on which another privilege protection analysis could be performed.

The use of forward slicing does not resolve the issue. Suppose that one would compute forward slices from privilege verification routines. This information, again, would give us possible privilege protection at best, as we would have no guarantees about unprotected paths missing from the slice.

Finally, slicing on an SDG is a quadratic operation when higher precision is sought, and slices tend to be large in practice, which prompted thinner slicing variants. Even if slicing would be adapted to determine privilege protection, multiple slices would need to be computed, as the slice is dependent on the seed statement.

In comparison, PTFA considers property satisfaction and provides both possible and definite protection information for all points of the program. PTFA analysis operates in linear time thanks to the merging of contexts that have the same property satisfaction. Nonetheless, it does so with no loss of precision in comparison with a fully context sensitive analysis.

2.3 Evolution Classifications

Software evolution classifications provide a common vocabulary and a method to group similar program changes together.

However, general software evolution classifications (e.g. [33, 38]) are not suited for the classification of RBAC evolution [134]. This is because they do not consider security maintenance, or only do so minimally. Security-related classifications and taxonomies (e.g. [97, 179]) are also inappropriate for RBAC evolution, since they consider evolution minimally. This is also the case for classifications related to access control (e.g. [3, 90]).

In comparison, our classification is built for differences in privilege protection. It therefore integrates the dimension of access control with software evolution.

2.4 RBAC Analysis

Both RBAC policies and implementations can be analyzed. In general, this comparison requires a specification of what is an inappropriate protection. These specifications are often lacking. As such, RBAC analyses often rely on inferred policies, inconsistency in implementation, or contradictory requirements. These analyses, however, do not inform the user of *what* changed, but that there are issues in the current form of access control.

2.4.1 Analysis of RBAC Policies

Model checking can validate that an RBAC policy allows users to achieve their objectives. Model checking also allows to determine how an attacker could violate a formal policy specified in the RW language [191]. It is also possible to find anomalies and conflicts in eXtensible Access Control Markup Language (XACML) policies [74, 151]. General queries over XACML policies are possible using SMT [177] and Answer Set Programming [153].

However, these approaches have four limitations. First, they require a semi-formal or formal policy specification. Second, they do not take in consideration their use in the source code (e.g. parts of the policy may not be used in practice). Thus, these approaches do not consider how the policy changes affect the privilege protection in the code. Third, applications may have hidden logic in conjunction with the RBAC policy [108]. Finally, these approaches are limited in their ability to compare policies.

2.4.2 Analysis of RBAC Implementations

To address the shortcomings of RBAC policy analysis, one may extract an RBAC policy from the application. For instance, it is possible to reverse engineer a SecureUML model of an application [7, 59, 163], which can then be analyzed independently [8].

In a similar vein, Le et al. [106] propose a hybrid analysis to infer the access control policy. They use dynamic security testing tools and record the application's behaviour. Using machine learning, they build access control rules for resources. Some rules are problematic and require review by the developers.

In general, an issue with reverse-engineering policies is that they should be validated by developers. This may be a resource-intensive tasks for large applications. Developers reverse-engineered policies could use the formal verification tools mentioned above. They would suffer from the same limitations, except for the fact that the policy reflects what is implemented.

Many approaches skip this intermediary step and analyze code-level RBAC implementation directly. These analyses typically do not determine differences in privilege protection.

RoleCast [167] uses heuristics to detect parts of the program that control security-sensitive statements in Java programs. RoleCast partially connects the security issue to its cause, by providing an enriched calling context. However, this calling context contains less information than an execution path. In addition, RoleCast depends on inferences, whereas PTFA relies on *a priori* knowledge of the code patterns used for privilege checks.

Fix Me Up [168] is an extension of RoleCast. It statically finds missing access-control checks. It does so using automatically-generated access control templates. Fix Me Up relies on slicing, which has an higher complexity than PTFA.

CanCheck [28] is a hybrid analysis tool that examines the source code for inappropriate implementations of the RBAC policy. It is designed for applications running on Ruby on Rails and using a specific authorization library. This approach extracts access control models dynamically, but verifies authorization properties statically using a first-order logic theorem prover. As with other dynamic analyses CanCheck depends on the code coverage in the analyzed executions. In addition, this approach may not scale to large applications, as the theorem prover may not terminate. In comparison, PTFA's complexity is linear with regards to the number of CFG edges.

Alalfi et al. [9] and Gauthier et al. [58] presented approaches based on clone analysis to identify similar code that does not have the same privilege protection or patterns in interactions indicative of security access violations. However, the approaches were used for clones within the same version.

The approaches mentioned above work within one version of the application. As such, they would not detect regressions (i.e. DPDs). In addition, they could not identify the code change that caused a DPD nor guarantee that it conservatively found its root causes.

2.4.3 Detection of Access Control Changes

Various approaches have been proposed to compare policies: multi-terminal BDDs [55, 119], Description Logic [93], Answer Set Programming [4], first-order temporal logic [48], and rewrite systems [31, 85]. Many of these contributions have limited policy comparison, focusing on equivalence [4, 48, 85]. Thus, the tool could only tell the user that the policy changed, but not how it changed. Most of these approaches operate on XACML policies. In contrast, PTFA analyzes source code, and the methods we introduce in this thesis tell us *how* privilege protection differs, which code change introduced the differences, and if the identified changes contain all root causes.

2.5 Access Control Evolution Studies

Wei et al. [182] studied how permissions and their use evolved in the Android operating system. They found that permissions are added and deleted, following new functionalities offered by the device. They also found that Android apps are becoming increasingly over-privileged. They also found that there is no tendency towards finer-grained permissions.

Hwang et al. [80] studied the SELinux access control policies for the NCSU Virtual Computing Lab. They found that growth was linear and that the policies were changing frequently.

Letarte et al. [112] also studied access control evolution using PTFA. Their survey was over 31 phpBB releases, and only handled a binary distinction between administrator and unprivileged users. This approach was a precursor to the contributions in this thesis. Whereas their access control model was binary, our approach handles a plurality of privileges. While they focused on protection for database access in PHP applications, we considered privilege protection for all parts of the program. We also introduced classifications of privilege differences, two ways to explain them, and a method to determine if the root cause was identified.

Han et al. [69] examined the evolution of the default MediaWiki RBAC configuration. They found that privileges were hierarchically organized using formal concept analysis.

Except for the survey by Letarte et al. [112], these RBAC evolution studies did not find differences in privilege protection between versions. In addition, none of these contributions generate explanatory counter-example, identify code changes responsible for DPDs and guaranteed that root causes have been identified.

2.6 Analyses Leveraging Program Differences

We mention a few of the many approaches that rely on program differences between versions.

Bugginings [166] identifies the cause of a bug in the context of software evolution by differencing dependence graphs. It computes program dependences from a bug fix version and the preceding version. Then, it identifies a *bug region* by comparing these dependences. Their analysis then examines the dependence graphs of each version in reverse chronological order until it finds the version in which the bug region appears. Their approach relies on CFG differencing.

Hydrogen [107] uses multiversion interprocedural CFGs to determine if changes fix bugs, and if changes break the code of other versions. This CFG variant is built using an initial CFG and by incrementally adding control flow changes of successive versions. This representation allows to determine changes of program properties between program versions, as well as allowing analyses that operate simultaneously on many program versions.

PatchAdvisor [139] infers the impact of a patch on a system. It compares CFGs of an unpatched and a patched version of the software. Then, PatchAdvisor uses a hybrid analysis to determine which paths are executed during the program's execution. PatchAdvisor then reports whether an execution trace intersects a modified part of the CFG, or if an execution path executes near a modified part of the CFG. In addition, traces allow to rank results based on how often parts of the program are executed.

These approaches have mostly focused on bugs, and, to the best of the author's knowledge, none has targeted RBAC implementations. In addition, they rely on a bug database, which makes them unable to find and fix unknown bugs. In comparison, DPDs allows developers to identify unknown security bugs. The other analyses support finding the root causes and is suitable to automated repair.

2.7 Change Impact Analysis

We briefly describe change impact analysis, which bears similarities with our approach. Change impact analysis aims at estimating as accurately as possible the impact of a change. In the context of software systems, the scope of the analysis is often regarding code changes, but it may encompass documentation [116] and other project files (e.g. graphics or multimedia files) [84]. Change impact analysis determines an *impact set*. The *starting impact set* [14] is the set of objects that initially appear affected by a change. The *estimated impact set* [14] is the set of objects that would be affected by the change. The *actual impact set* [14] is the set of objects actually modified as the result of performing the change. The granularity of these sets varies depending on the approach [116]. Because change impact analyses are often used for estimation, the *impact set* normally refers to the estimated impact set. The size of this impact set is a common metric (e.g. [140]).

Our approach is similar, but works in reverse - given an impact on definite privilege protection, we determine which paths and code changes explain it. To the best of my knowledge, no change impact analysis method targeting privilege protection has been proposed.

2.8 Root Cause Analysis

As we mentioned earlier, it is possible to perform root cause analysis on software artifacts.

AutoPaG [120] is a system that automatically identifies bug root causes and automatically patches them. Its analysis targets buffer overflows and similar out-of-bound vulnerabilities using data flow analysis.

Thung et al. [173] identify bug root causes. They use a combination of machine learning and program analysis. Their approach relies on AST differencing, which identifies some changes as essential. They intersect essential changes with line-level differences. Machine learning is then used to filter out changes that are unlikely to be part of a root cause. They also use a static analysis on data dependencies. In comparison, PTFA uses a conservative static analysis instead of machine learning.

Schneider identifies root causes of failures and defects in Simulink models [162]. Her approach uses a backwards search to find all paths leading to the defect. This method also determines weights to determine the probability that each path found contains a root cause.

To the best of my knowledge, no RCA method targeting privilege protection differences was proposed prior to the article included in Chapter 7.

CHAPTER 3 RESEARCH PROJECT AND THESIS ORGANIZATION

The articles presented in this thesis belong to one research project. We elaborate on their link to this project and to each other. We also show how the articles are connected to the research goals we presented in Chapter 1.

3.1 Research Project

The research project in this thesis is intended to facilitate the revalidation of the RBAC policy's implementation. The approaches and tools we present revolve around DPDs and additional information we obtain from formal security models. This information would assist developers to determine which protection differences are likely to be harmful, and which code changes are responsible for these differences. The articles in this thesis belong to the Research Objectives stated in Section 1.3. Please note that this project omits my earlier work on static taint analysis of web service compositions [99].

In Chapter 4, we include our article titled "Classification and Distribution of RBAC Privilege Protection Changes in Wordpress Evolution", which was published in PST 2017 [103].

We address RO1 and define definite protection differences (DPDs). DPDs occur whenever code that is common to two versions has a different definite privilege protection in each version. We also address RO2 and propose a set-theoretic classification of DPDs. In addition, we survey the occurrence of DPDs in 147 release pairs of WordPress, a content management system. We also present the distribution of DPD categories in this system. The identification and classification of DPDs supports our project's objective for two reasons. First, identifying which parts of the code have changed protection focuses reviewers' attention where it matters. Second, it may help developers determine if the change is problematic or not, since a statement's definite protection may change for more than one privilege at time. Determining DPDs is an essential first step for the other analyses in this thesis.

In Chapter 5, we include our article titled "Computing Counter-Examples for Privilege Protection Losses Using Security Models," with minor corrections [101]. It was published in SANER 2017.

We address RO3 for one kind of DPDs, *losses*. Definite protection losses occur whenever a statement shared between two versions was definitely protected for a privilege in the previous version, but is no longer protected for the same privilege in the newer version. To help developers understand why the loss occurs, we compute examples. They are unprotected paths in the formal model from the start of the program to loss-affected statements. Because shorter paths are likely to be easier to understand, we use graph transformations to remove interprocedural subpaths that have no effect on definite privilege protection. We also survey privilege protection losses in 147 release pairs of WordPress and calculate counter-examples for them. We also report the length and locality of these counter-examples. Protection loss counter-examples support our project's objective by showing why some DPDs occur.

In Chapter 6, we include our article titled "Detection of Protection-Impacting Changes During Software Evolution," with minor corrections [100]. It was published in SANER 2018.

We address RO4 by defining protection-impacting changes (PICs). PICs are changed code between versions that may have caused DPDs. PICs are computed using graph differences and graph reachability in security models. We also survey 210 release pairs of WordPress for protection-impacting changes and report on their distribution. Calculating protection-impacting changes helps the revalidation of RBAC policy implementation by focusing review efforts. Using PICs, developers avoid wasting effort reviewing code changes with no effect on privilege protection.

In Chapter 7, we present an extension article over Chapter 6, with minor corrections [102]. This article is titled "RBAC Protection-Impacting Changes : a Case Study of PHP Application Evolution" and was submitted to IEEE Transactions on Software Engineering.

This extension addresses RO5. We revert PICs and determine that the root causes of DPDs are contained in the set of protection-impacting changes (PICs). We also report on a survey of PICs in 210 release pairs of WordPress and 192 release pairs of MediaWiki. MediaWiki is another content management system. We also survey the reversal of PICs in these systems' security-affected release pairs. Determining that protection-impacting changes contain the root causes of DPDs helps the revalidation of RBAC policy implementation by focusing review efforts. Whenever this is the case, developers are certain that evaluating PICs is sufficient to understand the reason for all DPDs between two versions.

3.2 Terminology

The terms we used evolved because we felt that they would confuse the reader or did not convey well our contributions. We provide a correspondence between terms in Table 3.1.

Table 3.1 Correspondence Between Terms

Newer Term	**Equivalent to**	**Found in**
Pattern-Traversal Flow Analysis (PTFA)	Security Pattern Traversal	Related work
Definite Privilege Difference DPD	Privilege Protection Changes	Chapters 4 and 5
Security-affected release pair	Security-impacted release pairs	Chapter 4
Loss-affected	Privilege protection loss	Chapter 5

CHAPTER 4 ARTICLE 1: CLASSIFICATION AND DISTRIBUTION OF RBAC PRIVILEGE PROTECTION CHANGES IN WORDPRESS EVOLUTION

Marc-André Laverdière and Ettore Merlo

To appear in the Proceedings of the 15th International Conference on Privacy, Security and Trust (PST 2017), 2017

Abstract

Role-Based Access Control (RBAC) is commonly used in web applications to protect information and restrict operations. Their security may be affected by source code changes between releases in unexpected ways. To prevent regression and vulnerabilities, developers need to validate them prior to each release, which may be a major undertaking. We automatically and statically determine privilege-level security impacts of code changes using *privilege protection changes* and apply a set-theoretic classification to them. To do so, we analyze code and determine the security privilege protection models of Web applications written in PHP using Pattern Traversal Flow Analysis (PTFA). We present the distribution of both privilege protection changes and their classification over 147 release pairs of WordPress, spanning from 2.0 to 4.5.1. We found that code changes had no impact on privilege protection in the 82 (56%) release pairs. The remaining 65 (44%) release pairs are affected by privilege protection changes. For the latter release pairs, only 0.30% of code is affected by privilege protection changes. We also found that the most common change categories are *complete gains* (40.81%), *complete losses* (17.99%) and *substitution* (20.10%). The automated identification and classification of *privilege protection changes* may help developers to more efficiently focus their effort during security reviews, verification, validation, testing, and repairs.

Keywords

Security Change Classification, Role Based Access Control, Static Analysis, Software Evolution, Software Maintenance

4.1 Introduction

Many web applications require *access control*, which is ensuring that only the right user(s) can execute specific code. Access control vulnerabilities present in the OWASP Top 10 2013 include in the *Insecure Direct Object References* and *Missing Function Level Access Control* categories [145].

A statement's *privilege protection* set is the set of privileges that are definitely *true* prior to its execution. Given a statement s common to two releases (r_a and r_b), a *privilege protection change* [101] occurs when the privilege protection for s differs between r_a and r_b.

In Listing 4.1, we see an example of a code change causing privilege protection changes for the statements at line 5 onwards. As a check for privilege `edit_posts` was added, their privilege protection set went from $\emptyset$ to $\{edit\_posts\}$. Please note that this information is carried interprocedurally along that execution path (e.g. for the execution `get_user_option`).

```
1 + if(! current_user_can( 'edit_posts' )){
2 +         return;
3 + }
4 +
5 $last_post_id = (int) get_user_option( '
    dashboard_quick_press_last_post_id' );
```

Listing 4.1 Excerpt of a Security Change in WordPress 4.2.2 vs 4.2.3 – the marker `+` on the left hand side refers to added code.

A vulnerability like the one solved by Listing 4.1 stems in part from the challenge of knowing which privileges have been verified prior to the execution of a statement. Security reviews are meant to catch these issues, but they are difficult and expensive to perform.

We propose to facilitate reviews by automatically computing and classifying *privilege protection changes* from the source code. In this paper, we define a set-theoretic classification of privilege protection changes and report on a longitudinal case study of privilege protection and their changes. We studied 147 release pairs (from 2.0 to 4.5.1) of WordPress. WordPress is a popular PHP web-based content management system that uses RBAC. Measuring only the PHP code, this application's size ranges from 35,708 to 301,605 physical lines of code (PLOC).

4.2 Background

A commonly used access control approach is role-based access control (RBAC), where *privileges* are assigned to one or more *roles* as necessary. RBAC systems are often architected with Policy Decision (PDP) and Enforcement (PEP) Points. PEPs request a policy decision from the PDP and enforce it, whereas the PDP decides to either grant or deny requests by interpreting the policy.

PEPs are typically scattered across the code base and communicate with the PDP before allowing execution of sensitive code. The PEP asks the PDP to verify if the user holds a given privilege, sometimes in relation to an object. The APIs to do so vary, but tend to be consistent within the same application.

Some maintenance activities will target security. For instance, developers change which privilege protects a piece of code. They may add a privilege check where none was, or move privilege checks to cover only some code sections instead of the whole script (or vice-versa). Developers also add, remove and rename roles and privileges. Some of these changes may be related to functional changes (e.g. new features added), but sometimes reflect a reorganization of the application's security policy at a high level.

Counterintuitively, code changes with no apparent link to security may cause privilege protection changes. Some of these changes may be due to interprocedural side effects, since code protected by a privilege check often calls other code and carries its protection along that execution path.

Code changes during maintenance should only have a planned and desirable impact on security. To achieve this, maintainers may leverage techniques such as testing (e.g. penetration testing [13] and generated security tests [135]), manual verification (e.g. inspections [51, 148]) and validation methods (formal and semi-formal) [152].

4.3 Methodology

4.3.1 Privilege Protection Changes

Privilege protection is a *definite* static analysis, determining properties that are *true* on all execution paths prior to a statement. This is in contrast with *possible* analyses, which determine properties that are *true* in some execution paths, but not necessarily in all paths reaching a statement.

Formally, we define privilege protection as a function over an interprocedural control flow graph (CFG).

Each release $r \in R$ has an CFG composed of vertices $V_r \subseteq V$ and edges $E_r \subseteq E$. All releases share the set of privileges $Privs$, though not all releases check all privileges.

Privilege protection is defined by function $PrivProt(v) : V_r \mapsto \mathcal{P}(Privs)$ (Equation 4.1), which returns all the privileges that are definitely protecting (*DefProt*) the vertex v in release r.

A privilege protection change occurs when the privilege protection for corresponding CFG vertices differs between two releases (Equation 4.2). The relationship between corresponding vertices is in the partial bijective function $vertexMap : V \times R \times R \mapsto V$. When comparing releases r_a with r_b, the CFG vertex v_a belongs to release r_a and v_b is its corresponding vertex in r_b's CFG.

$$PrivProt(v) \doteq \{priv \in Privs \mid DefProt(v, priv)\} \tag{4.1}$$

$$PrivProtChg(v_a, r_a, r_b) \doteq PrivProt(vertexMap(v_a, r_a, r_b)) \neq PrivProt(v_a) \tag{4.2}$$

4.3.2 Pattern Traversal Flow Analysis

Pattern-Traversal Flow Analysis (PTFA) [57, 61, 111] is a domain-specific model-checking approach for privilege protection. A PTFA engine analyzes the application's source automatically and statically for application-specific code patterns for privilege verification security checks. It generates security-context-sensitive security models, meaning that the security context is distinguished in interprocedural analysis.

States in the security model are either protected or unprotected, the latter being the default. A protected state associated to a vertex $v \in V$ represent the existence of a path from the beginning of the CFG to v, in which the privilege *priv* has been granted. $DefProt(v, priv)$ thus holds true when only protected states are reachable for v.

4.3.3 Privilege Protection Change Categories

Our set-theoretic classification distinguishes between the addition and removal of privileges. For the sake of simplicity, we use the following symbols in the descriptions below. When comparing releases r_a with r_b, the vertex v_a belongs to release r_a and v_b is its corresponding vertex in r_b such that $v_b = vertexMap(v_a, r_a, r_b)$. Due to space constraints, we define the aliases $Prot_a := PrivProt(v_a)$ and $Prot_b := PrivProt(v_b)$. We also define the predicate $A \bigtriangleup_r B$, which is true if and only if the sets A and B are non-empty and have a non-empty overlap.

We describe each privilege protection category below. We also provide an illustration of all the categories in Figure 4.1 and formal definitions in Figure 4.2.

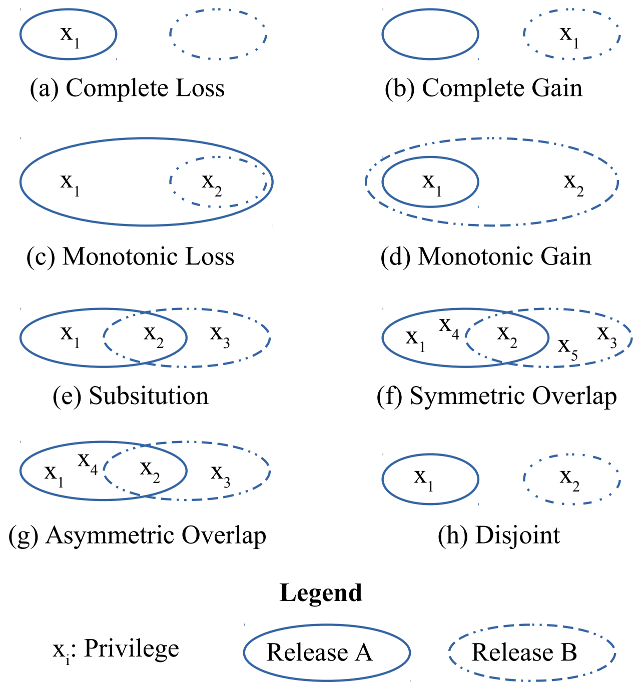


Figure 4.1 Classification Categories Illustrated

$$
\begin{aligned}
A \triangle_r B &\doteq A \cap B \neq \emptyset \;\wedge\; A \neq \emptyset \;\wedge\; B \neq \emptyset \\
CompleteLoss(v_a, r_a, r_b) &\doteq Prot_a \neq \emptyset \wedge Prot_b = \emptyset \qquad CompleteGain(v_a, r_a, r_b) \doteq Prot_a = \emptyset \wedge Prot_b \neq \emptyset \\
MonotonicLoss(v_a, r_a, r_b) &\doteq Prot_a \cap Prot_b = Prot_b \;\wedge\; Prot_b \neq \emptyset \\
MonotonicGain(v_a, r_a, r_b) &\doteq Prot_a \cap Prot_b = Prot_a \;\wedge\; Prot_a \neq \emptyset \\
Substitution(v_a, r_a, r_b) &\doteq |Prot_b \setminus Prot_a| = 1 \;\wedge\; |Prot_a \setminus Prot_b| = 1 \\
AsymmOverlap(v_a, r_a, r_b) &\doteq Prot_a \triangle_r Prot_b \;\wedge\; |Prot_a \setminus Prot_b| \neq |Prot_b \setminus Prot_a| \\
SymmOverlap(v_a, r_a, r_b) &\doteq Prot_a \triangle_r Prot_b \;\wedge\; |Prot_a \setminus Prot_b| > 1 \;\wedge |Prot_a \setminus Prot_b| = |Prot_b \setminus Prot_a| \\
Disjoint(v_a, r_a, r_b) &\doteq \neg Substitution(v_a, r_a, r_b) \;\wedge Prot_a \cap Prot_b = \emptyset \;\wedge\; Prot_a \neq \emptyset \wedge Prot_b \neq \emptyset
\end{aligned}
$$

Figure 4.2 Definitions for the Classification Categories

A **Complete Loss** (Figure 4.1a) may indicate a vulnerability and occurs when $Prot_b$ is empty and $Prot_a$ is not.

A **Complete Gain** (Figure 4.1b) may indicate a vulnerability correction and occurs when $Prot_a$ is empty and $Prot_b$ is not.

A **Monotonic Loss** (Figure 4.1c) may indicate a privilege reorganization (e.g. merged privileges) and occurs when $Prot_b$ is a strict subset of $Prot_a$ and both are non-empty.

A **Monotonic Gain** (Figure 4.1d) may indicate a privilege reorganization (e.g. split privilege) and occurs when $Prot_a$ is a strict subset of $Prot_b$ and both are non-empty.

A **Substitution** (Figure 4.1e) may indicate an association between the privileges (e.g. renaming) and occurs when $Prot_a$ and $Prot_b$ each have one privilege in addition of those they have in common.

An **Asymmetric Overlap** (Figure 4.1g) might indicate multiple changes happening simultaneously. It occurs when $Prot_a$ and $Prot_b$ have at least one privilege in common and the releases' specific privileges are of different cardinality.

A **Symmetric Overlap** change (Figure 4.1f) may indicate a non-obvious correspondence between the changed privileges. It is similar to substitutions, but for greater cardinalities.

A **Disjoint** (Figure 4.1h) changes may indicate major changes that warrant inspection. It occurs when $Prot_a$ and $Prot_b$ have nothing in common and are not a substitution.

4.4 Results

Our analysis, implemented in Java, typically occurs in less than 7 minutes per release pair on a system powered by an Intel i7-3770 CPU @ 3.40GHz and using the Oracle Java VM version 1.8.0_66 configured to use up to 8Gb of RAM. We used GNU `diff` to determine $vertexMap$ and the same release pair determination as previously [101].

4.4.1 Distribution of Privilege Protection in WordPress

Our first research question is **RQ1:** *How is the cardinality of privilege protection sets in WordPress distributed?*

In Figure 4.3, we compare the total number of definitely protected statements (i.e. statements which have non-empty privilege protection sets) with the program size measured in CFG vertices. When compared logarithmically, the curves appear to grow together. Total definitely protected statements and log program size is correlated at 96.77%.

In Figure 4.4, we represent the number of occurrences of privilege protection sets per version, per set cardinality. Singleton sets vastly dominate the other cardinalities. The largest cardinality is four, and is found in a few versions.

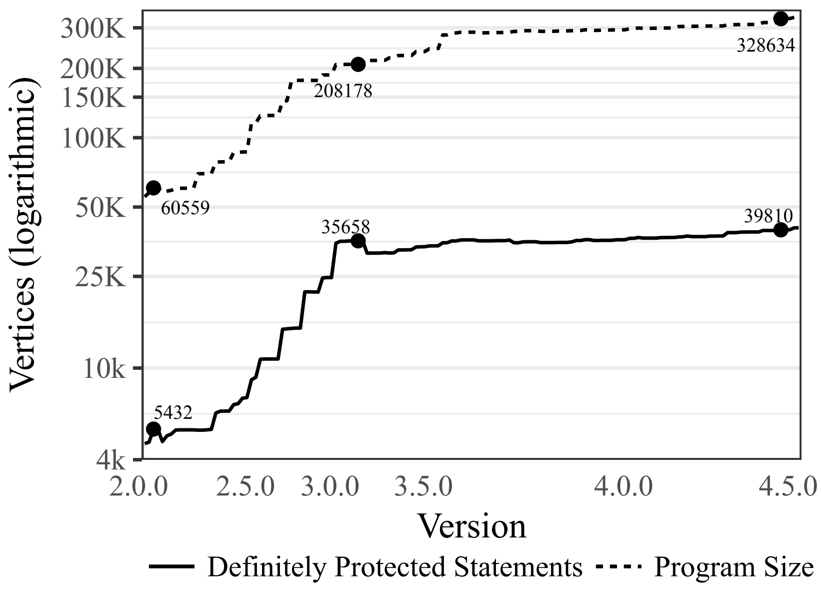


Figure 4.3 Total Vertices with Non-Empty Privilege Protection Set vs Program Size per version - definite protection grows logarithmically to program size.

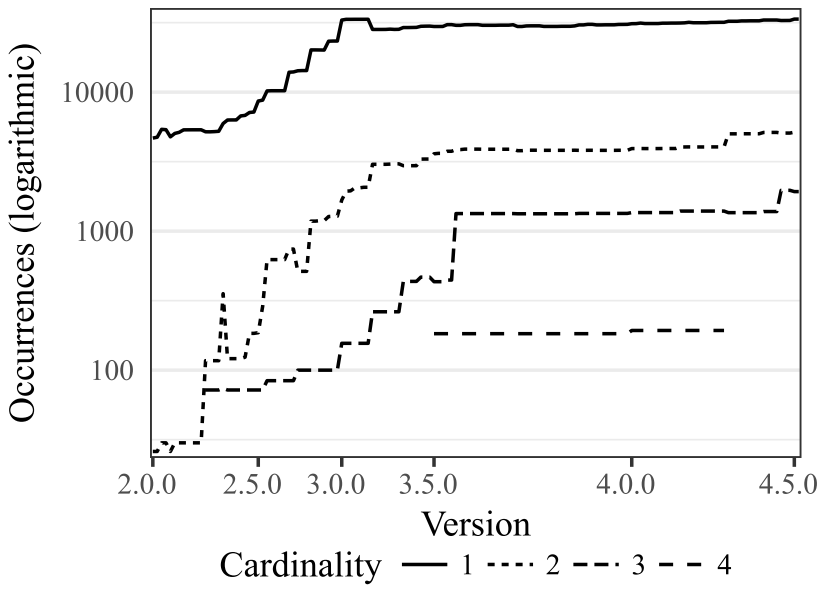


Figure 4.4 Evolution of Definitely Protected Vertices' Privilege Protection Set Cardinalities - singleton sets dominate throughout.

4.4.2 Distribution of Privilege Protection Changes in WordPress

We now consider **RQ2** : *What is the distribution of privilege protection changes in WordPress? How big are those changes in relation to the program size?*

Over all release pairs, we found 33,687 privilege protection changes. We observe, in Figure 4.5, that only 65 (44%) release pairs are security-impacted. The bulk of release pairs have less than 500 privilege protection changes, with only eight (5%) exhibiting more than a thousand.

We determined the proportion of privilege protection changes by comparing the number of privilege protection changes over the number of CFG vertices in the whole program.

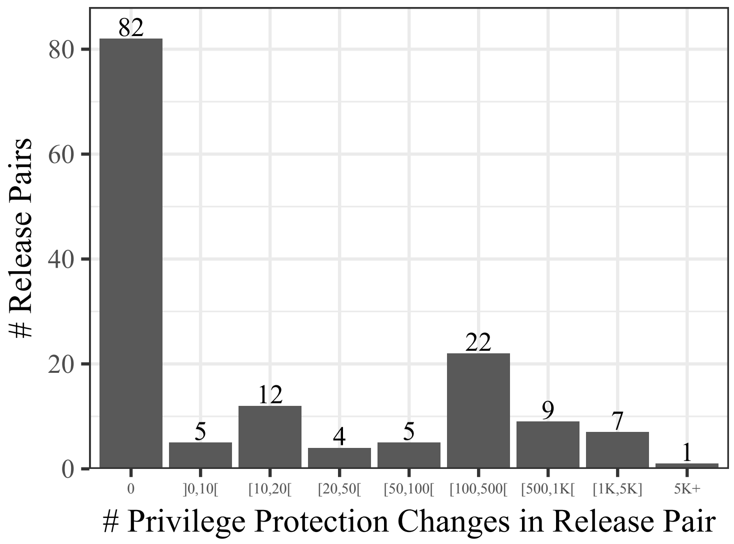


Figure 4.5 Distribution of Total Privilege Protection Changes per Release Pair

Globally, privilege protection changes is found in 0.10% of vertices (computed over all CFG vertices of all releases studied). For security-impacted release pairs, this ratio increases to 0.25%. The median is 182 (0.08%) privilege protection changes per release pair, while the average and standard deviation is 0.30% ± 0.58%, a small fraction of the program.

4.4.3 Distribution of Privilege Protection Change Classification

We now examine **RQ3** : *What is the distribution of privilege protection changes' classification in WordPress? Which classifications are the most common?*

Table 4.7a shows how many releases have which amount of privilege protection changes of each category. We found no instance of symmetric overlap. We observe that only Complete Gain and Substitution are present with ≥ 1000 privilege protection changes.

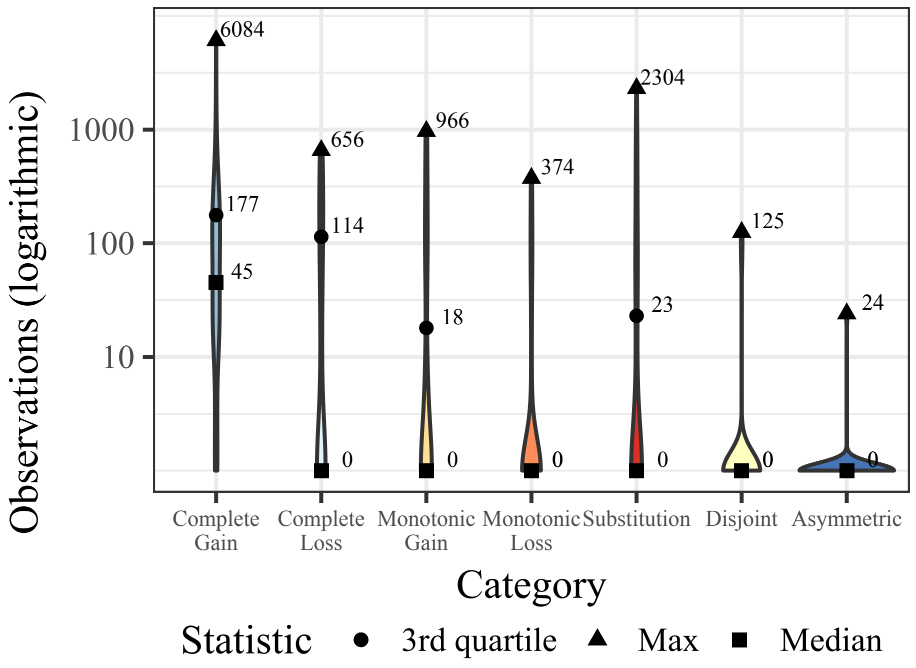


Figure 4.6 Violin Plot of Classified Privilege Protection Changes Per Non-Empty Release Pair - Some Categories Are Seldom Seen.

Table 4.7b shows the total number of vertices per category for all release pairs as well as its average per release pair. Figure 4.6 shows the distribution of classified vertices for security-impacted release pairs. The distributions are skewed towards zero, and the most common change category is complete gain, followed by substitution and complete loss.

Category	**Privilege Protection Changes**							
	0	1-10	11 -20	21-50	51-100	101-500	501-1000	> 1000
Complete Gain	86	7	12	10	8	20	2	2
Complete Loss	119	1	4	3	2	14	4	0
Monotonic Gain	120	10	1	0	6	6	4	0
Monotonic Loss	135	2	1	2	3	4	0	0
Substitution	124	4	2	3	5	4	4	1
Asymmetric	146	0	0	1	0	0	0	0
Symmetric	147	0	0	0	0	0	0	0
Disjoint	143	1	0	0	1	2	0	0

(a) Release Pairs Showing Privilege Protection Change, per Preponderance and Category

Category	**Average**	**Total**	**%**
Complete gain	93.51	13746	40.81%
Complete loss	41.22	6059	17.99%
Monotonic gain	36.57	5376	15.96%
Monotonic loss	9.52	1399	4.15%
Substitution	46.07	6772	20.10%
Asymmetric	0.16	24	0.07%
Symmetric	0	0	0%
Disjoint	2.11	311	0.92%
Total	229.16	33687	100.00%

(b) Summary Statistics

Figure 4.7 Classified Vertices for All Release Pairs – Complete Gain, Complete Loss and Substitution are the Dominant Categories.

4.5 Discussion

Our results show that code changes between releases have a small privilege-level security impact, as there are very few privilege protection changes in proportion to the program size (0.10%).

Identifying this small change may help focus validation efforts, because most of the program need not be evaluated for regressions at the privilege level. This is explained by the fact that an absence of privilege protection changes imply that the privilege protection for that code is just as valid for r_b as they were for r_a.

We observed a strong logarithmic correlation (96.77%) between the program size and the total definitely protected statements. This may indicate that developers purposefully confined privileged operations in a small part of the application. The logarithmic growth in definitely protected statements is markedly different from earlier results from Hwang et al. [80], which found a linear growth in SELinux polices of the NCSU Virtual Computing Lab.

Privilege protection changes classification may prioritize review efforts, as they may indicate the nature of the changes and their security impacts. For instance, a complete loss of privilege protection may be much more troublesome than a monotonic increase of privilege protection and developers may want to investigate the former first.

The additional information provided by our classification might further facilitate validation-style testing, since the dominant classification categories are very black-and-white – moving from unprotected to protected (or vice-versa) or entirely from one privilege to another. This kind of straightforward information may be easier to reason about.

We also reviewed all results for accuracy and violations of security assumptions – which privileges are required over which parts of the code. Since WordPress does not explicitly document its security assumptions, we built a plausible mental model of these security assumptions using code patterns (e.g. database and HTML writes), identifier names as well as file location in the project's hierarchy (e.g. code reserved for administrators).

During a code review, we used this mental model and the CFG to determine if the privilege protection change was accurate and if it could affect the system's security.

We found our classification to be 81.56% accurate and 70.01% of privilege protection changes are, in our opinion, both security assumption violations and confirmed results.

These results hint that our approach provide useful information for developers, although they are subject to several limitations, which we will discuss in Section 4.7.

4.6 Related Work

PTFA was used in evolution studies before. A first study by Letarte et al. studied privilege protection in a longitudinal study over 31 phpBB releases [112], though only handling a binary distinction between administrator and unprivileged users. Our earlier study [101] only considered privilege protection losses. This study is different, as we proposed a set-theoretic classification encompassing all possible privilege protection change and reported on their observed frequencies.

Hwang et al. [81] developed a semantic differencing approach of eXtensible Access Control Markup Language (XACML) policies for test selection. Their approach requires running the full test suite at least once, it also needs a stand-alone access control policy or pre-recorded PDP decisions. In comparison, our approach targets RBAC policies, is fully static and works for policies that are hardcoded and mixed with business logic in the source code.

General classifications of software evolution (e.g. [33, 38]) are not suited for RBAC classification evolution [134] as they scantly cover security maintenance.

Classifications and taxonomies for security (e.g. Wang and Wang [179]), vulnerabilities (e.g. Landwehr et al. [97]) and access control (e.g. Kane and Browne [90]) do not (or only minimally) integrate the dimension of evolution.

4.7 Threats to Validity

Threats to internal validity refer to confounding variables that may influence our results. Our results depend on the accuracy of the PTFA engine we used. Because PHP applications like WordPress rely on many dynamic features, the engine relies on sound but conservative approximations that may lead to spurious paths and thus spurious privilege protection changes.

Our results also depend on the differencing method used. We extracted line-level differences between releases. Since there may be many CFG vertices on the same line of code, we underestimate the unchanged CFG vertices. Thus, the rate of privilege protection changes may be higher than we report. However, we expect change categories to be distributed similarly as we found.

As our validation was done by non-experts, the accuracy of our results and the real rate of privilege protection changes that are violations of security assumptions may be very different than we found. Thus, it would be desirable to conduct a formal review of our results with experts.

Threats to external validity relate to the generalizability of our results. We did not have a vulnerability oracle for our study, though our approach may use one. Consequently, we cannot establish a relationship between vulnerabilities and the privilege protection change categories. To counter this issue, studies using a vulnerability oracle should be performed.

Another threat to generalizability is that our study is on a single open source content management system implemented in PHP. We may obtain different results when studying other systems written in PHP or other languages. While the PTFA engine we used only handles PHP at the moment, our approach itself is language-independent. To counter this issue, studies that include other systems should be performed.

Another threat to generalizability is that our study is on a single type of access control. While the PTFA engine we used only handles RBAC at the moment, it could be extended to handle a wide range of access control models, such as attribute-based access control and authorization-based access control [91]. It could also be extended to analyze applications that use technologies separating access control decisions from the business logic, such as XACML.

4.8 Conclusion and Future Work

We studied privilege protection impact of code changes using *privilege protection changes* and their classification. We examined their demography over 147 release pairs of WordPress. Our automated implementation need only a few minutes per release pair and has detected 33,687 privilege protection changes – 0.10% of the cumulative program size. Only 44% of the releases had any privilege protection changes. For security-impacted release pairs, only 0.30% of code is affected by security changes. We also saw that complete gain, complete loss and substitution were the most common change categories.

We manually verified our results for false positives and violations of our understanding of the developers' security assumptions and we estimate that 70.01% of privilege protection changes represent possible security violations.

We find that proportionally few privilege protection changes, if any, occur between releases. These changes mostly belong to simple categories, which are likely to be simple to reason about and prioritize. These factors are likely to significantly lower the effort required for security reviews.

In our future research, we would like to investigate of the distribution of privilege protection change categories corresponding to vulnerabilities, whenever a vulnerability oracle is available.

Further research may also include the assessment of the real impact of the proposed techniques to application security verification and validation in collaboration with developers. Such studies would measure the accuracy of our results, developers' validation efforts during an application evolution and assess the advantages of the proposed approach.

Acknowledgements

We would like to thank Mathieu Lavallée for his constructive feedback and guidance in the classification literature.

CHAPTER 5 ARTICLE 2: COMPUTING COUNTER-EXAMPLES FOR PRIVILEGE PROTECTION LOSSES USING SECURITY MODELS

Marc-André Laverdière and Ettore Merlo

Published in the Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution, and Reengineering (SANER 2017), 2017, pp. 240–249. Minor corrections added.

Abstract

Role-Based Access Control (RBAC) is commonly used in web applications to protect information and restrict operations. Code changes may affect the security of the application and need to be validated, in order to avoid security vulnerabilities, which is a major undertaking. A statement suffers from privilege protection loss in a release pair when it was definitely protected on all execution paths in the previous release and is now reachable by some execution paths with an inferior privilege protection. Because the code change and the resulting privilege protection loss may be distant (e.g. in different functions or files), developers may find it difficult to diagnose and correct the issue. We use Pattern Traversal Flow Analysis (PTFA) to statically analyze code-derived formal models. Our analysis automatically computes counter-examples of definite protection properties and privilege protection losses. We computed privilege protections and their changes for 147 release pairs of WordPress. We computed counter-examples for a total of 14,116 privilege protection losses we found spread in 31 release pairs. We present the distribution of counter-examples' lengths, as well as their spread across function and file boundaries. Our results show that counter-examples are typically short and localized. The median example spans 88 statements, crosses a single function boundary, and is contained in the same file. The 90th centile example measures 174 statements and spans 3 function boundaries over 3 files. We believe that the privilege protection counter-examples' characteristics would be helpful to focus developers' attention for security reviews. These counter-examples are also a first step toward explanations.

Index Terms

Software Maintenance, Access Control, Evolution, Static Analysis, Model Checking

5.1 Introduction

Many web applications require *access control*, which is ensuring that only the right user(s) can execute specific code. Sadly, they commonly suffer from various vulnerabilities, which are defined as “a flaw or weakness in a system’s design, implementation, or operation and management that could be exploited to violate the system’s security policy.” (RFC 4949 [165]). Access control vulnerabilities include *forced browsing*, *privilege escalation* and *unauthorized access*. Forced browsing [131, 142] typically happens when a page is unreachable by normal browsing unless the user has the necessary privileges (e.g. admin) and doesn’t verify that the user has these privileges. An unauthenticated user browsing directly to the URL of the page can then perform privileged operations. Privilege escalation [142] and unauthorized access refer to existing privilege checks in the application that do not validate for the right privilege(s). The presence of such vulnerabilities show how difficult it is for developers to reason about which security properties have been verified at any given point in the program.

A commonly used access control approach is role-based access control (RBAC), with developers and/or administrators assigning *privileges* to one or more *roles* as necessary. Developers often implement RBAC systems with privilege checks scattered across the code base according to the application’s security needs. These checks typically verify if a given privilege has been granted to the authenticated user. The NBAC (authenticatioN-Based Access Control) family of access control includes RBAC, Attribute-Based Access Control (ABAC) and identity-based access control (IBAC). A different paradigm is authoriZation-Based Access Control (ZBAC) [91]. In ABAC, an authenticated user is granted a set of read-only attributes, often expressed as key-value pairs. The policy decision point (PDP) verifies if the attribute set contains a specific attribute and value combination. In IBAC, the user’s identity is used by the PDP to determine access via an access control matrix. In ZBAC, a user is given authorizations upon authentication. The PDP checks if the user provided the required authorization and that this authorization is valid.

Software maintenance refers to modifications to a product after release. The types of maintenance are Corrective, Adaptive, Perfective and Preventive [83]. Many maintenance activities in all categories will be functional in nature - meaning that they affect the functionality of the application. This implies adding, removing or changing execution paths in the program.

On the other hand, some maintenance activities will be oriented towards security.

As an example, developers change the privilege checks protecting a piece of code. They may add a privilege check where none was, or they may move privilege checks to cover only some code sections instead of the whole script (or vice-versa). Also, developers add, remove and rename roles and privileges. Some of these decisions may be linked to new features added, but sometimes reflect a reorganization of the application's high-level security policy.

Non-security changes in the source code may change privilege protection. Some of these changes may be due to interprocedural side effects, since code protected by a privilege check often calls other code, and carries its protection to the called functions, which may be located in other files.

Whether or not code changes for security, privilege protection changes may have negligible, beneficial or harmful effects. Negligible privilege protection changes would, for instance, affect a code section that only accesses unrestricted resources for unrestricted operations. Beneficial privilege protection changes are synonymous with security improvements, often as a fix for a security vulnerability. Harmful privilege protection changes, on the other hand, may represent vulnerabilities in the application.

Maintainers' objective is that their changes comply with the security policies. In other words, code changes should only have a planned and desirable impact on security. To ensure this is the case, developers may resort to *regression security testing*, *penetration testing* and *inspection*. Regression security testing is a form of regression testing whereby tests are validating some security properties (e.g. non logged-in users should receive an error message when trying to view a given page). These tests look for previous application behavior and states that are no longer secure. Penetration testing is a dynamic evaluation often performed by an hired third party. Penetration testers use both automated scanning tools and manual interaction with the system to find vulnerabilities [13]. Inspections, on the other hand, are static formal investigation processes meant to verify that requirements are met, and security goals have been added to their scope [148]. They greatly reduce the defect rate, but they typically amount to 15% of the project cost [51].

Such validation efforts may be difficult and costly for many reasons, such as complexity between software and security, lack of security architecture documentation, lack of specifications, contradictory implementation, configurability, and testing complexity. First, the interactions between functionality and security is often complex, and maintainers' choices may have unintended and sometimes catastrophic security consequences (e.g. the Debian OpenSSL vulnerability[1]). Second, the security architecture of an application is not always documented. It may rely on "common sense", or be a mental model shared by the security architect and the developers. Third, few applications are developed using formal security specifications. This is especially the case for Free, Libre and Open Source Software (FLOSS)

[1]Debian maintainers caused CVE-2008-0166 by silencing a Valgrind warning about uninitialized memory access. Consequently, generated keys tended to be predictable.

web applications. Fourth, even when informal security policies are used, the implementation may deviate from them, and developers may not have useful tools to detect deviations. Furthermore, automated tests may not cover many security issues, as they are typically focused on functionality testing. Considering the constraints above, validation-style testing is very desirable. It is, however, a time consuming activity.

To illustrate these challenges, we introduce our first running example in Listing 5.1. `a.php` contains a check for privilege `p1` and its call to `foo` is in a protected context. The code change occurs in `b.php`, with an additional call to `foo`, which we underlined below. Since `b.php` has no privilege checks, it calls `foo` from an unprotected context. Previously, the user always had privilege `p1` when executing `foo` in `c.php`, but we are no longer sure.

```
/* in a.php */
if (!current_user_can('p1')) die();
foo();

/* new code in b.php */
foo();

/* in c.php */
function foo(){/* assumes privilege p1 held */}
```

Listing 5.1 Running Example - Adding an Unprotected Path

The *privilege protection* set for a statement is the set of privileges that are definitely *true* prior to its execution. This means that the privilege is positively verified on all paths leading to the statement's execution. For a statement s common to two releases (r_a and r_b), a *privilege protection change* occurs when the privilege protection of s differs between r_a and r_b.

In our running example, the privilege protection for statements in function *foo*, prior to the change, was $\{p1\}$. After the code change, it became $\emptyset$. In other words, the code change in `b.php` caused a *privilege protection change* in `c.php`. Specifically, it was a *privilege protection loss*.

A privilege protection loss for privilege p is a case of privilege protection change whereby the privilege protection for a statement s in r_a included p, but excludes it in r_b. In other words, the application positively verifies privilege p on all paths leading to s in r_a, but there is at least one path to s lacking positive verification in r_b. These paths are *definite privilege protection counter-examples*. This is derived from First Order Logic, as the negation of a universally quantified property (e.g., not definitely protected), is equivalent to the existentially quantified negated property (e.g., there exist a path that is not protected).

We call our approach *privilege protection loss examples* because we only generate definite privilege protection counter-examples for privilege protection losses. Statements that present privilege protection losses in r_b are statements that are no longer definitely protected for the same privilege. Therefore, *examples* of protection losses identify paths that are *counter-examples* of a definite protection property. Throughout this paper, we'll use both *protection loss examples* and *definite protection counter-examples* interchangeably to refer to these existentially quantified paths.

For code performing sensitive operations or accessing sensitive resources, a privilege protection loss may represent a vulnerability. Considering the validation challenges mentioned above, developers are likely to find privilege protection loss examples useful to determine if any code change violates the application's security policy. The reason for this is because privilege protection loss examples connect the cause and effect, something that may not be obvious for developers, especially when they occur in different functions or files. Since examples represent execution traces that violate security properties, they should be easy to analyze and act upon.

As with other static analysis tools, we must minimize the amount of information to review [137] in order to facilitate analysis. We do so via *brevity* and *locality.* For brevity, we generate short counter-examples, as done with many other model checking approaches [66, 98]. For locality, we provide counter-examples that spread in as few functions or files as possible, since local information is easier for developers to reason about [64, 113].

Our approach is warranted because generating definite protection counter-examples for every unprotected statement would result in too much data to analyze. And this data would not be useful for many parts of the applications consisting of code free of security assumptions and concerns. By computing privilege protection loss examples, we aim at generating reasonable amount of relevant data.

We also minimize the information to analyze via graph transformations. We transform Pattern Traversal Flow Analysis (PTFA) model checking automata, which are automatically generated from the application's source code.

To the authors' best knowledge, this is the first study of privilege protection loss examples, as well as the first longitudinal study of their characteristics. We examined the evolution of WordPress, a popular web-based content management system implemented in PHP. The study spanned 147 release pairs, from 2.0 to 4.5.1. This application's size ranges from 35708 physical lines of PHP code (PLOC) in release 2.0 to 301605 PLOC in release 4.5.1.

In this paper, we include the following contributions:

- We define privilege protection loss examples
- We provide an algorithm to compute definite privilege protection counter-examples that leverage graph transformations of PTFA automata
- We report on a longitudinal case study of privilege protection loss examples over 147 release pairs of WordPress.

This paper is organized as follows. We describe our methodology in Section 5.2. We examine the results of our research question in Section 5.3. We discuss our results and the previous literature in Sections 5.4 and 5.5, respectively. We consider threats to our results' validity in Section 5.6 and bring concluding remarks in Section 5.7.

5.2 Methodology

Developers are likely to understand shorter and more local traces better than long traces spread across the code base. As such, we study the distribution of privilege protection loss examples length in terms of states traversed, as well as the number of files and function boundaries they span.

We summarize PTFA in Section 5.2.1 and show how to determine privilege protection losses in PTFA models in Section 5.2.2. Then, we elaborate on our algorithm to calculate counter-examples in Section 5.2.3. Afterwards, we explain how we selected the release pairs for our study in Section 5.2.4.

5.2.1 Pattern Traversal Flow Analysis

We use Pattern-Traversal Flow Analysis (PTFA) [57, 61, 111], a domain-specific model-checking approach to determine privilege protection. PTFA models are generated automatically and statically from the application's source code. To help the reader better understand how we compute privilege protection loss examples, we now summarize essential details of PTFA.

PTFA considers privilege check satisfaction as Boolean properties interprocedurally propagated across a Control Flow Graph (CFG). Accordingly, our engine creates model checking automata from the CFG and computes the graph reachability from the starting state. PTFA automata are security-context-sensitive, meaning that the security context is distinguished in interprocedural analysis. Since PTFA automata track a single Boolean property at a time, we compute one automaton per privilege in the system.

A PTFA model checking automaton is a tuple $\mathcal{A} = (Q_\mathcal{A}, T_\mathcal{A}, q_0, V_\mathcal{A}, G_\mathcal{A}, A_\mathcal{A})$. $Q_\mathcal{A}$ is a finite set of states. $T_\mathcal{A} \subseteq Q_\mathcal{A} \times Q_\mathcal{A}$ is the set of transitions. $q_0 \in Q_\mathcal{A}$ is the initial state. $V_\mathcal{A}$ is the finite set of variables used in *guards* and *assignments*. The guards $G_\mathcal{A}$ annotate the transitions $T_\mathcal{A}$ with logical propositions over $V_\mathcal{A}$. These propositions must be satisfied for the edge to be traversed. The assignments $A_\mathcal{A}$ also annotate the transitions $T_\mathcal{A}$ with modifications to the variables.

The states $Q_\mathcal{A}$ are represented as $q_{i,j,k}$, where i is the corresponding CFG node identifier and j and k are protectedness flags set to either 0 or 1. j and k refer to the calling context's and the intra-procedural protectedness, respectively. The value 0 for j or k applies for states both with unknown privilege protection as well as those that are in code regions reachable only though an explicit negative check (e.g. `if (!current_user_ can('p')...)`).

Call sites in the CFG are converted to two states *cb* and *ce*, which respectively stand for *call_begin* and *call_end*. The relationship between them is recorded in the *cb2ce* function. Edges of type *call_target* connect a *call_begin* state to the start of the function or method. Edges of type *call_return* connect a state representing a return statement, or the end of a function or method to a *call_end* state.

In Figure 5.1, we see a simplified PTFA automaton for our running example. Please note that we kept the assignments, guards and unreachable edges away from the figure for the sake of simplicity; we also greyed out unreachable states.

5.2.2 Privilege Protection Loss

Because we compute reachability on the automata from state q_0 and prune unreachable edges and states, Boolean property satisfaction becomes privilege protection information straightforwardly. If the privilege p_i is verified on all execution paths leading to CFG vertex v_i, then at least one state $q_{v_i,j,1}$ and no state $q_{v_i,j,0}$ are reachable in the automaton for p_i. If a state $q_{v_i,j,0}$ is reachable in the automaton for p_i, then p_i is not verified in at least one path to v_i.

In order to determine privilege protection changes, we compare the reachable PTFA states for corresponding CFG vertices. We define a partial function between the CFG vertices of releases r_a and r_b named *vertexMap*. For vertex v_a in r_a, its corresponding vertex in r_b is $v_b = vertexMap(v_a, r_a, r_b)$. A privilege protection loss occurs when the privilege p_i is verified on all paths leading to v_a, but not for v_b.

In Figure 5.2, we show how we detect privilege protection losses from PTFA automata. The graphs are simplified as for Figure 5.1. We see how different states for function `foo` are reachable for each release. For Release A, only $q_{i,1,1}$ is reachable from q_0, but both $q_{i,1,1}$ and $q_{i,0,0}$ are reachable in Release B.

5.2.3 Computing Counter-Examples

Using PTFA, we verify the following property: that the privilege p_i is verified on all execution paths leading to v_i. As counter-examples look for the opposite property, we find one path where p_i is not verified prior to v_i.

This is straightforward with PTFA. Given that $q_{i,j,0}$ is reachable from q_0, and that shorter counter-examples are desirable, we compute the shortest path between q_0 and $q_{i,j,0}$. Such counter-examples still risk to be very long, due to the fact that we would traverse all functions called along the path. This is problematic when the same j, k states are reachable before and after the call. In this case, the subpath corresponding to the call does not provide any useful information to understand the root cause of the privilege protection loss.

As such, we reverse and modify the PTFA automaton and calculate the shortest path between $q_{i,j,0}$ and q_0. We keep only *call_return* edges when the *call_end* state(s) and their corresponding *call_begin* state(s) have different j, k values. Whenever a *call_return* edge is disconnected, we add a *cecb* edge between the *call_end* state(s) and their corresponding *call_begin* state(s). This transformation maintains soundness because the *cecb* edge is a summary of the interprocedural subgraph between the *call_begin* and *call_end* states. Because the protection is the same between them, the entire subgraph has no effect on privilege protection and does not need to be part of the counter-example. Since many interprocedural edges are removed, we also remove guards and assignments from the automaton. We handle interprocedural edge selection using a stack in the traversal.

In Figure 5.3, we show how the PTFA automaton from Figure 5.1 is modified for calculating counter-examples, with the same simplifications as before.

The special treatment for *call_return* edges is due to privilege protection checks done in functions, and the privilege protection acquired there is propagated to the relevant *call_end* state. We show an example in Listing 5.2, along with its simplified and reversed PTFA automata in Figure 5.4.

5.2.4 Release Pair Selection

Because many open source projects, such as WordPress, maintain and apply security fixes to multiple releases in parallel, we must take in consideration these subtleties to choose which releases to compare. As such, we cannot consider releases sequentially. A useful tool to tell

```
function f(){
  if (!current_user_can('p1')) die();
}

f();
/* ... */
/* This code is protected by privilege p1 */
```

Listing 5.2 Example of Privilege Check in Called Function

the development activities apart is *semantic versioning* [149]. In semantic versioning, releases are numbered with the form X.Y.Z, whereby X is the major release number, Y is the minor release number, and Z is the bugfix release number.

Because many releases are maintained and developed at the same time, a security fix may need to be applied to all these releases. If the code to be fixed has evolved between the releases, maintainers often resort to *backporting*, which refers to the practice of writing a patch on the latest release, and then adapting it to the earlier release(s). We must take this behavior in consideration when choosing the release pairs to compare. Otherwise, we would observe privilege protection changes reversed and later reinstated.

If we were to build a graph of the evolution, with each release being a vertex, and the evolution activities being edges between the releases, we would obtain a directed acyclic graph. An evolution analysis should thus consider each pair of releases connected by an edge.

Because it would be very difficult to create this graph, we approximate it using a tree based on the releases' semantic version numbers and their release dates. We first connect the vertices in each minor release in increasing bugfix release order. Then, we add edges to major and minor releases (i.e. ending by .0) from the latest prior bugfix release that was released earlier than the release we are trying to connect.

We show an excerpt of this tree in Figure 5.5.

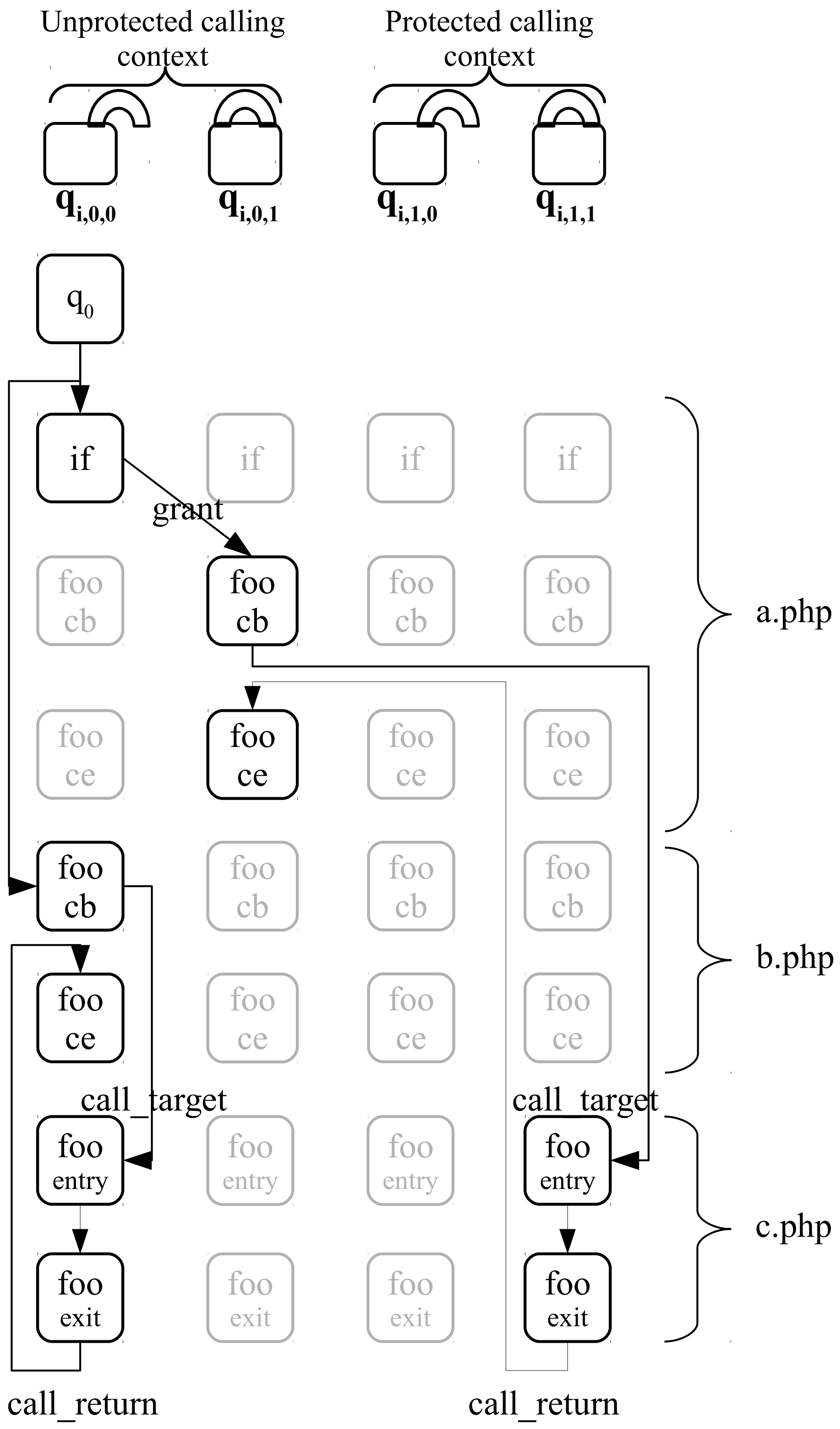


Figure 5.1 Simplified PTFA Model for our Running Example - Unreachable states are greyed out

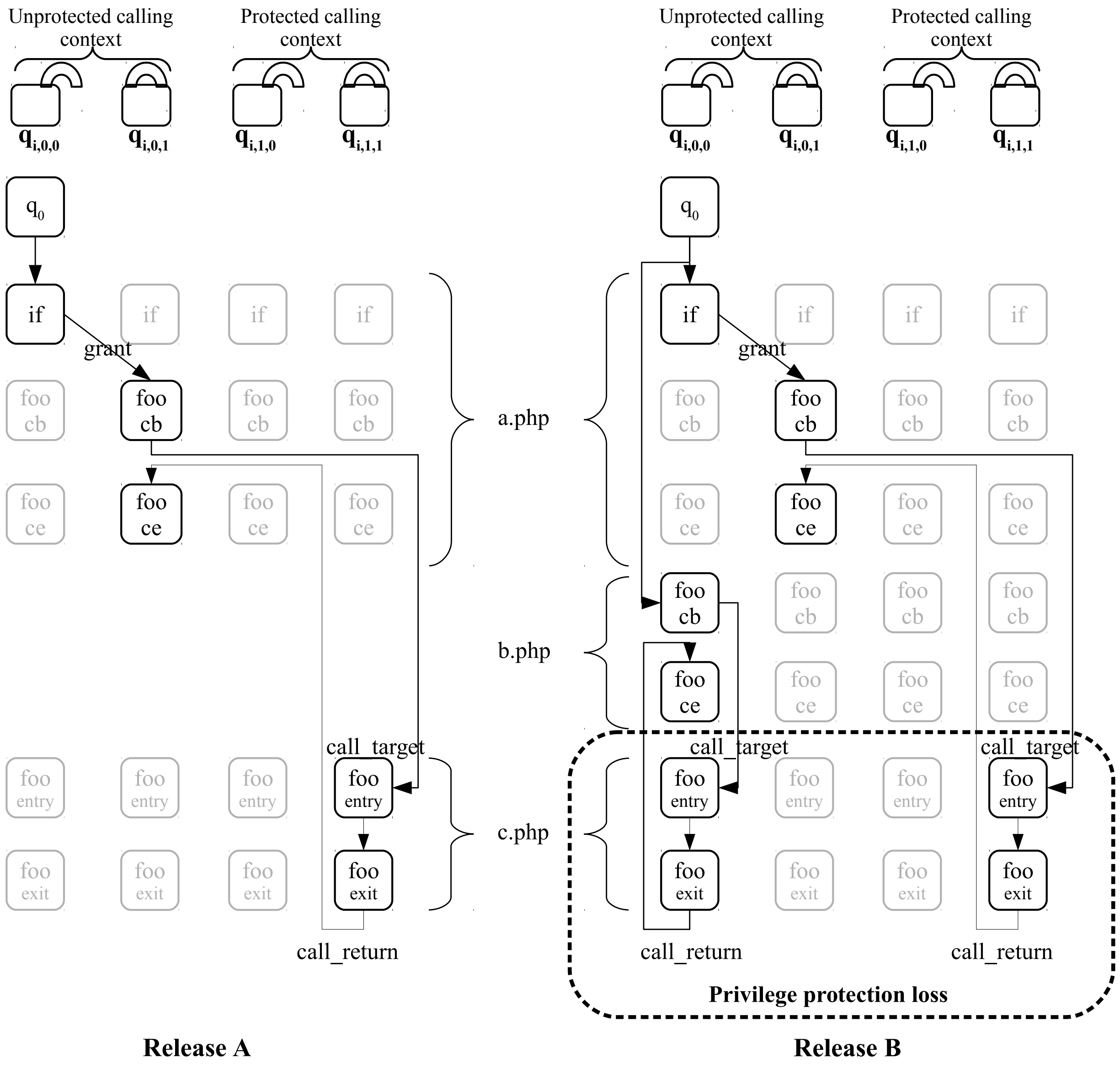


Figure 5.2 Privilege Protection Loss in Our Running Example. The dashed region represents the loss in Release B.

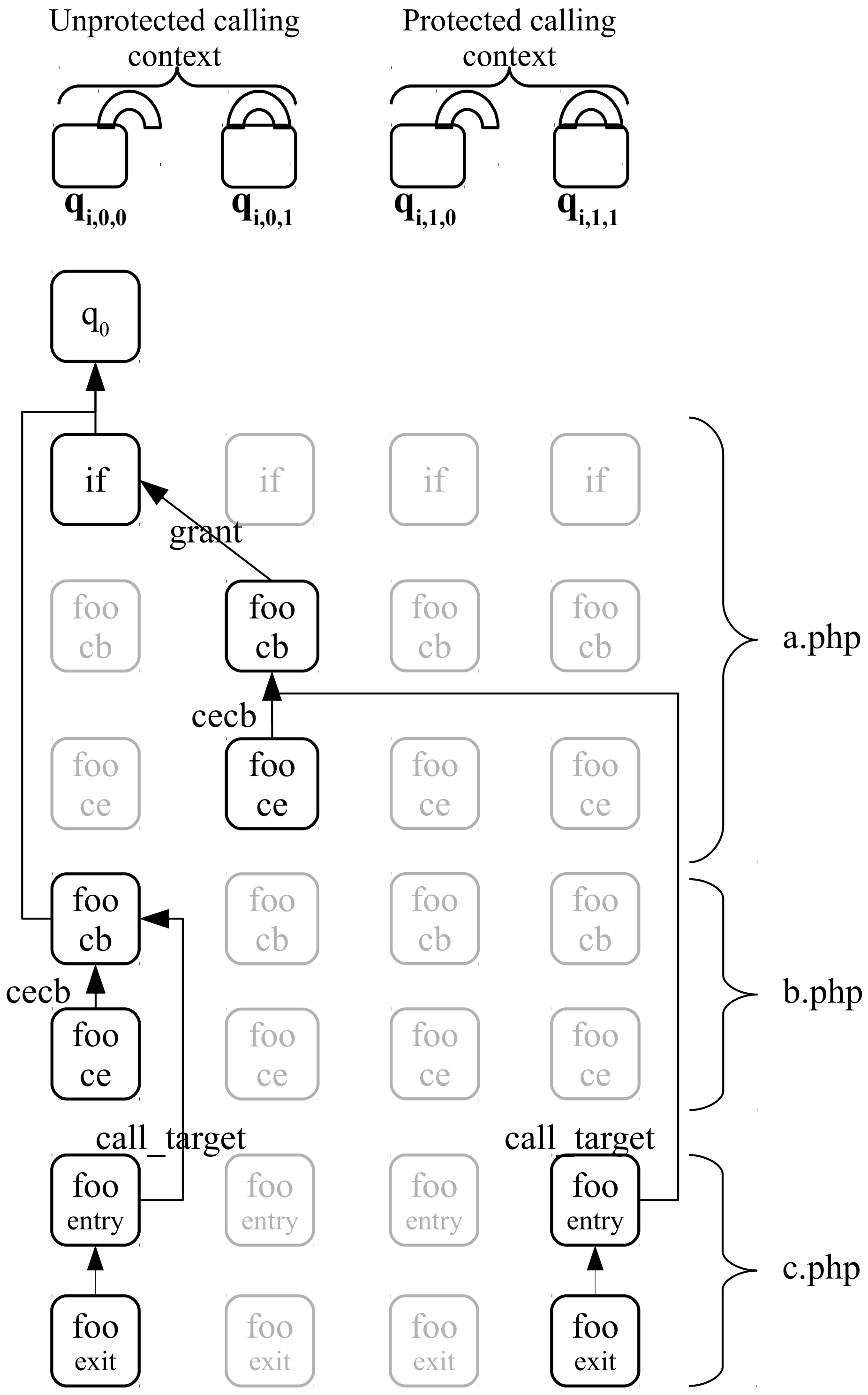
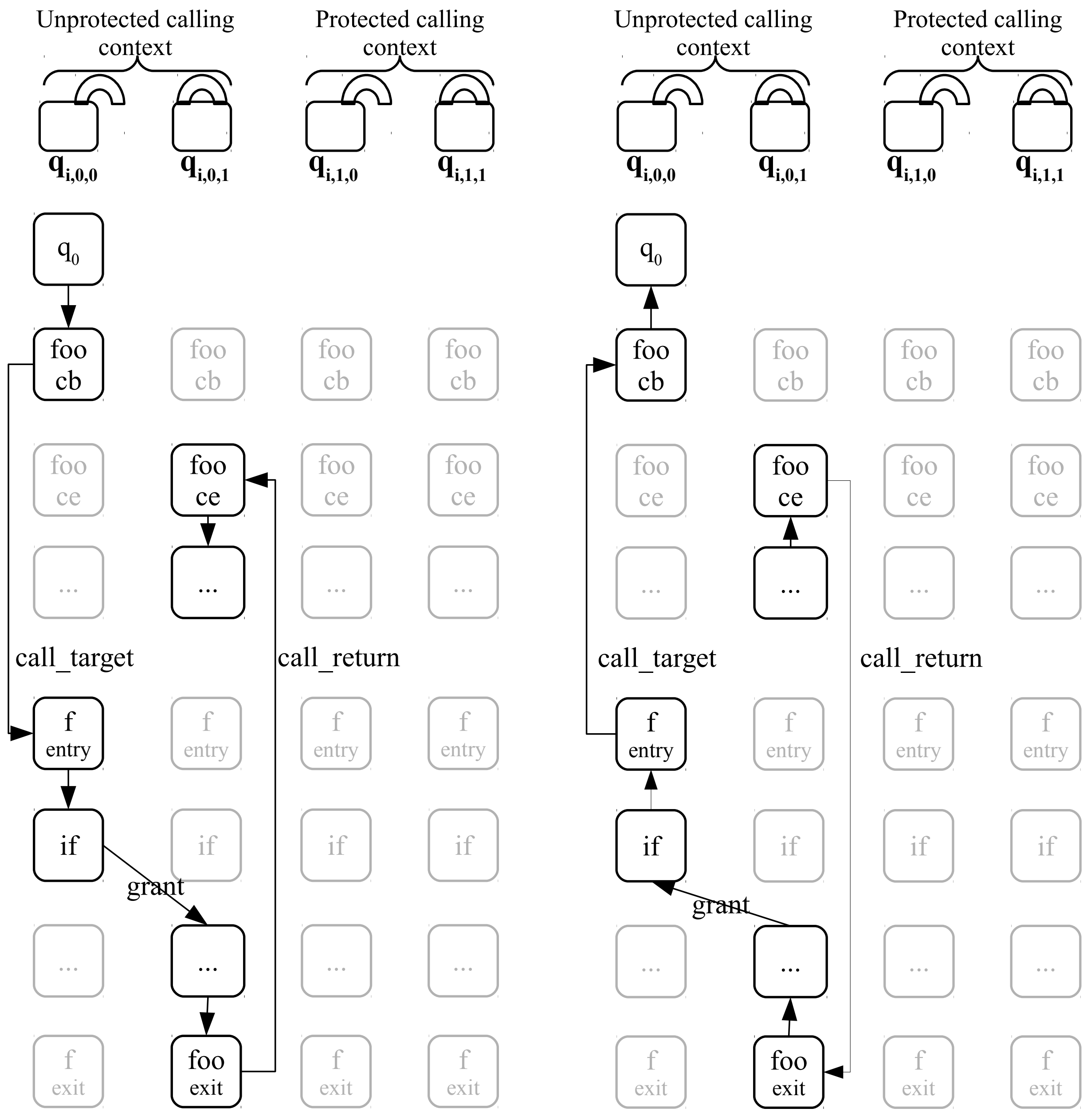


Figure 5.3 Reversed PTFA Model for our Running Example - Note that we removed *call_return* edges and added *cecb* edges



(a)
Original PTFA Automaton

(b)
Reversed PTFA Automaton

Figure 5.4 Reversed PTFA Model for Listing 5.2 - Note that we kept and reversed the *call_return* edge.

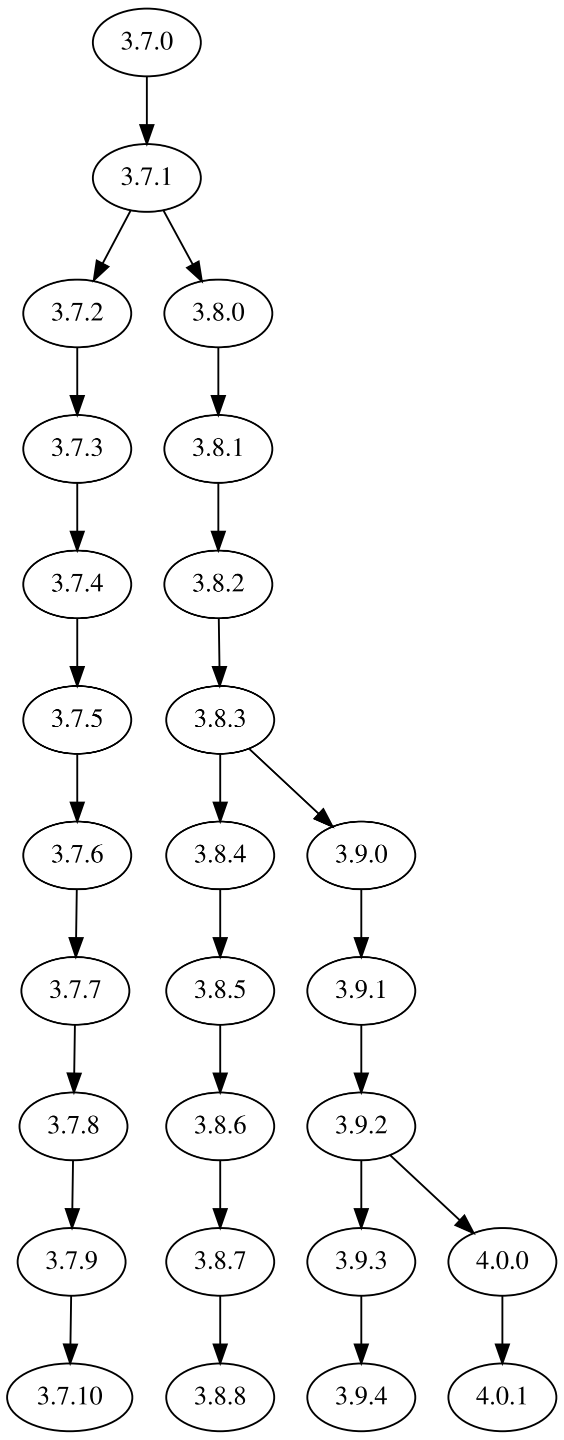


Figure 5.5 WordPress Release Tree Excerpt.

5.3 Results

We have implemented our analysis using a JavaCC grammar for handling PHP code, and our analyser and classifier are implemented in Java. Our analysis requires less than 13 minutes on a system powered by an Intel i7-3770 CPU @ 3.40GHz and configured the Oracle Java VM version 1.8.0_66 to use up to 8Gb of RAM. To determine the mapping between different releases' CFG vertices (*vertexMap*), we rely on line differences computed by GNU Diff. We compute physical lines of code (PLOC) reported below using the `wc` utility.

We compared privilege protections over 147 release pairs of WordPress. Protection changes may be classified as gains or losses. We found 14116 protection losses spread over 31 release pairs, for which we computed counter-examples. The results below only refer to these release pairs.

In Figure 5.6, we show the cumulative distribution of the number of counter-examples per release pair. We have a median of 211 and a mean of 455.4 counter-examples.

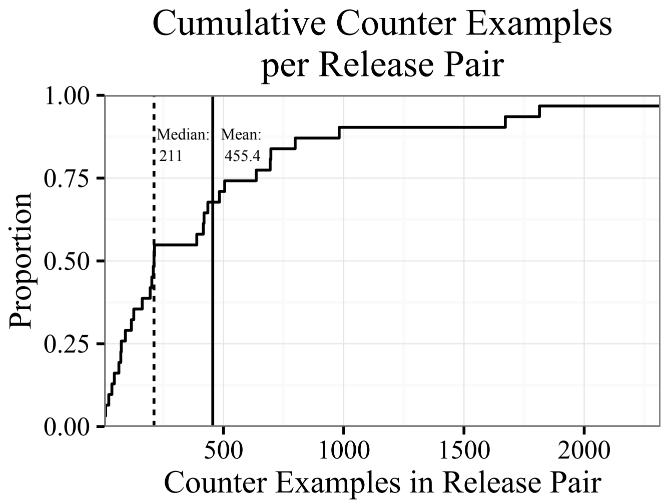


Figure 5.6 Cumulative Distribution of Counter-Examples per Release Pair. Its median is 211 counter-examples, and only 3 release pairs have more than 1000 counter-examples.

In Table 5.1, we see summary statistics for their length, the number of states visited before finding the example, function boundaries crossed, and files present. The median privilege protection loss example length is 88 states long, contained in one file, and across one function boundary.

Looking at the distribution of privilege protection loss example lengths in Figure 5.7, we observe a skew towards the left, and what appears to be a long tail of examples greater than 200 states long. We see the same situation in the cumulative density curve (Figure 5.8).

Table 5.1 Privilege Protection Loss Examples Examples Summary Statistics

Measure	**Mean**	**1st quart**	**Median**	**3rd quart**	**Max**
Length	97.64	56.00	88.00	127.00	364.00
States Visited	227.40	87.00	162.00	250.00	3187.00
Functions	1.06	0.00	1.00	2.00	5.00
Files	1.67	1.00	1.00	2.00	6.00

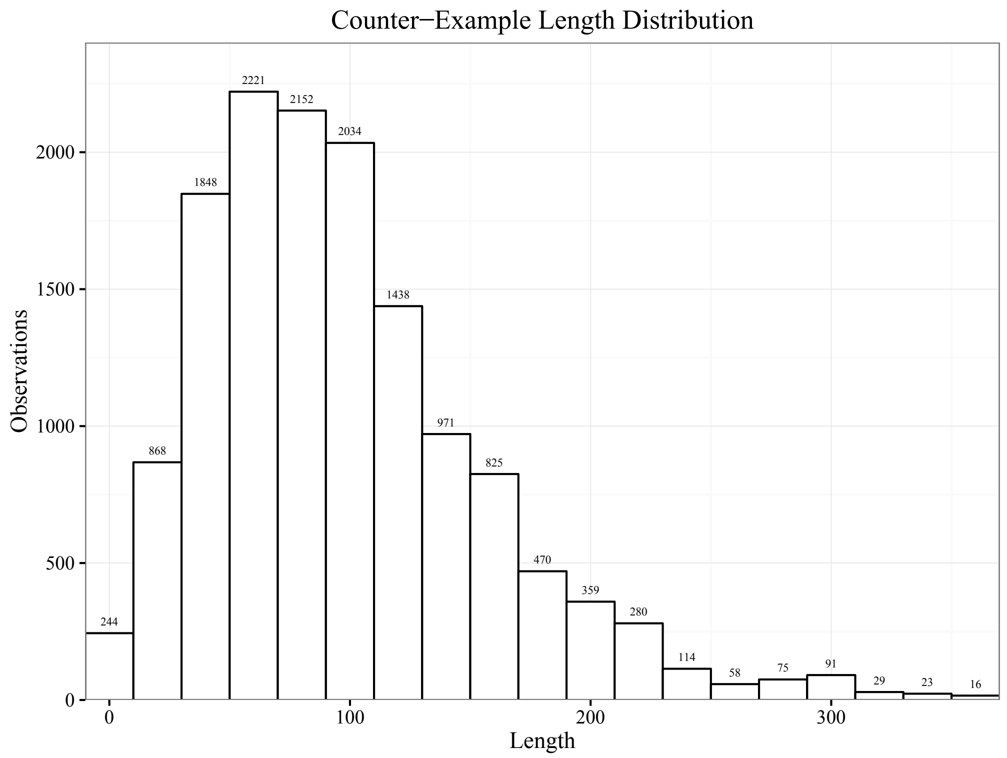


Figure 5.7 Distribution of Privilege Protection Loss Example Lengths. It is skewed towards the left.

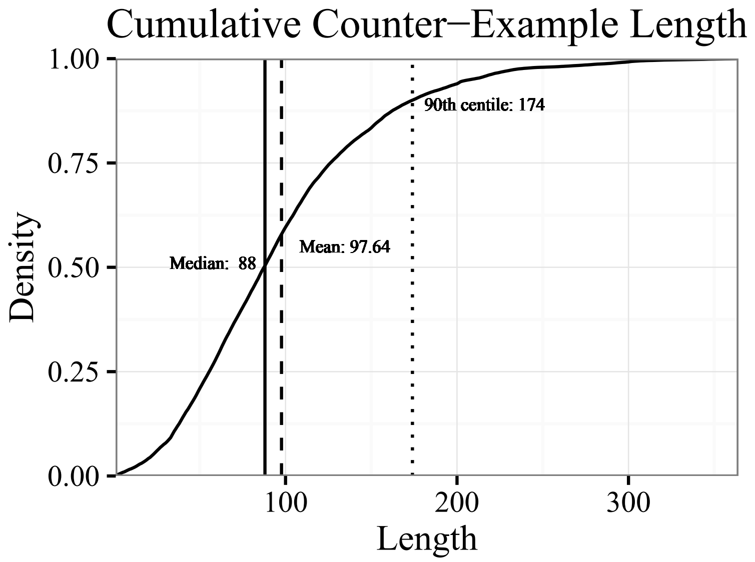


Figure 5.8 Cumulative Distribution of Privilege Protection Loss Example Lengths - The Median Length is 88.

In Figure 5.9, we represent the spread of privilege protection loss example lengths in relation to the program size. We use vertical bars to encompass the largest to smallest lengths, and a dot to represent the median. We see that privilege protection loss example lengths are across the spectrum and do not appear related to the program size.

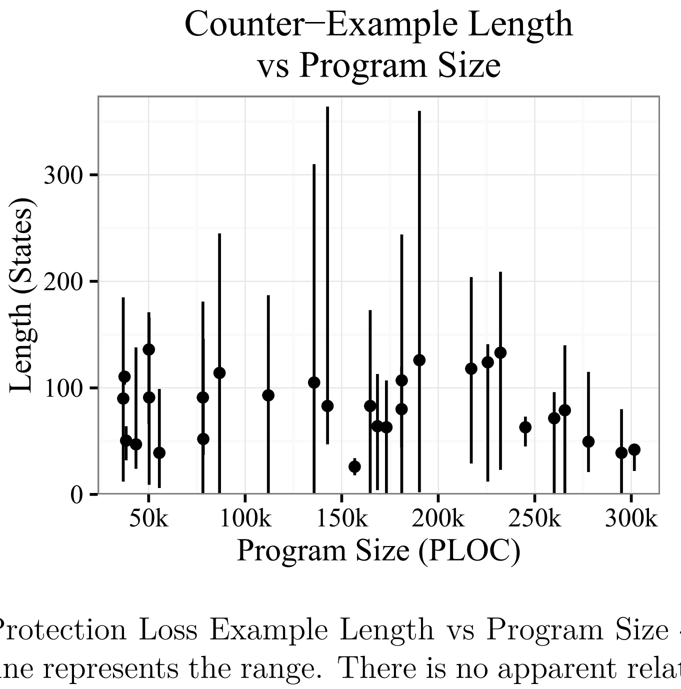


Figure 5.9 Privilege Protection Loss Example Length vs Program Size - The dots stand for the median, and the line represents the range. There is no apparent relationship between the length of privilege protection loss examples and the program size

In Table 5.2, we see that over a third of privilege protection loss examples occur within the same function scope, and that nearly another third of examples cross one function boundary.

Table 5.2 Distribution of Function Boundaries Crossed in Privilege Protection Loss Examples - More than one third of examples are within the same function.

Function Boundaries	**Observations**	**Proportion**
0	5525	39.14%
1	4539	32.16%
2	2384	16.89%
3	1150	8.15%
4	407	2.88%
5	111	0.79%
Total	14116	100.00%

In Table 5.3, we see that the majority of privilege protection loss examples fit within the same file, and that less than 2% of examples span four or more files.

Table 5.3 Distribution of Number of Files in Privilege Protection Loss Example - Most examples are all within the same file.

File Count	**Observations**	**Proportion**
1	7038	49.86%
2	4647	32.92%
3	2126	15.06%
4	229	1.62%
5	45	0.32%
6	31	0.22%
Total	14116	100.00%

5.4 Discussion

Most privilege protection loss examples are less than 100 states long and could be considered short. However, the far end of the distribution is ≥ 200 states long. We also note that most examples are contained in the same file, but are not contained in the same function.

Static analysis is powerful because it considers all program paths. However, it relies on approximations which may introduce spurious paths. To have an idea about the rate of spurious paths, we randomly selected and manually verified 378 counter-examples. The estimated spurious path rate of $10.96 \pm 3.18\%$ (95% confidence level) seems promising.

We expect that privilege protection loss examples will help developers to determine which code changes caused privilege protection losses, assess their security impact, and correct them if necessary. However, we expect that the examples in the tail end of the distribution represent examples long enough to discourage developers.

In that case, future research in generating explanations is likely to bridge the gap and present information that is even more useful for developers.

It is important to note that not all privilege protection losses amount to vulnerabilities.

A cornerstone of our approach is to restrict the number of counter-examples to generate. We do this by calculating them only privilege protection losses. This kind of targeting using privilege protection changes may be useful for generating counter-examples for other security properties in the future.

5.5 Related Work

Previous work underpinning our research was performed by Letarte, Gauthier and Merlo by defining the Pattern-Traversal Flow Analysis we are using [57, 61, 111, 112]. These include a single longitudinal study of privilege protection over 31 phpBB releases [112]. Their approach only handled a binary distinction between administrator and unprivileged users. Our study is markedly different, even though we also use PTFA and determine additions and deletions in the source code using textual differencing, since we handle a richer protection scheme, computed counter-examples and performed a larger case study.

Han et al. [69] also examined the evolution of privilege-based access control systems, but on the topic of their configuration. Their study leveraged Formal Concept Analysis (FCA) and found that privileges were hierarchically organized in the default Mediawiki configuration. In our study, we examine how these privileges used in the source code, with no hierarchization of any kind.

Montrieux et al. [134] presented a position paper showing that general software evolution approaches, aimed at consistency checking, were not sufficient for the evolution of RBAC. They do not address code-level RBAC implementation like we do, but show that RBAC evolution is an open research topic.

Hwang et al. [80] performed one of the first evolution studies of access control systems. They studied the evolution of access control policies in SELinux and for the NCSU Virtual Computing Lab. They aimed at creating a predictive model. They found that growth was linear and that the policies were changing frequently.

Letovsky and Soloway [113] demonstrated that purely local program understanding may lead to incorrect or inefficient modifications. Their study was based on programming plans involving source code scattered across the program named *delocalized programming plans*. While their study was focused on functionality rather than security, their results may plausibly be generalizable to security. Their study showed that programmers make incorrect (yet plausible) assumptions based on local information. They suggested to use automatic program analysis tools to make correct facts available to programmers.

Gordon [64] proposed a program clarity measure. He mentioned that locality of operation is a factor that influences the clarity of a program, alongside other factors related to program structure, such as the number of statements, the complexity of the control flow, the depth of statement nesting and the clustering of data references.

Groce and Kroening [66] minimize Bounded Model Checking (BMC) counter-examples on the length and semantic axes. They express counter-example generation as a minimization problem. They also show how to leverage the set of changes to determine causal dependence based on distance metrics. Their study was focused on verifying potentially complex properties in C programs, whereas we study relatively simple properties using a dedicated model-checking approach. Consequently, we would need to suitably redefine causal dependency.

Chaki et al. [35] perform error explanation by providing informative error explanation for property violations in C programs using an abstract state-space. They also express counter-example generation as optimization problem, but with a distance metric. They move from specific values in the counter-example to predicates over the variables (e.g. $x < y$ in the counter-example, but $x \geq y$ in the successful execution). They also use a scoring function for evaluating fault localization techniques that minimizes how much of the program an ideal user would have to read. While the idea of error explanation is attractive, their approach would also need to be redefined to apply on PTFA automata.

Son et al. introduced RoleCast [167], which automatically detects privilege protection checks in PHP and JSP applications. Their static approach identifies critical variables controlling the reachability of security sensitive statements. Through inferences, it partitions the application into roles and determines if checks are performed consistently. When reporting a potential vulnerability in a non-singleton context, RoleCast mentions the security sensitive statement, its calling context, which security variables are erroneously verified, or if a check is missing in the first place. As we do, they connect the security issue to its cause, but the calling context provides less information than an execution path. Their analysis is very different as well, since they rely on inferences and do not perform an evolutionary analysis.

5.6 Threats to Validity

1. Construction validity: Our PTFA analysis tool has some limitations. First, it supports PHP's constructs up to release 5.6. Also, it relies on approximations that may lead to spurious paths, it supports disjunctive privilege checks partially and performs only limited string analysis.

   Consequences: Newer constructs from PHP 7 that might have affected the control flow would not be taken in consideration, and spurious paths mean false positives. Disjunctive privilege are not included in definitive privilege protection, and limited string analysis means that some privileges checks refer to (partially or fully) unknown privilege names. We expect that these limitations would not significantly affect our results: few applications have made the switch to PHP 7, few privilege checks use a disjunctive form, we keep track of unknown privileges, and the high level of confirmed results indicate few spurious paths.

   Counter-measures: We could add support for newer versions of PHP, track disjunctive checks as separate privileges (`p1 || p2`), and improve the string analysis.

2. Construction validity: the implementation of our vertex mapping between revisions is line-based.

   Consequences: some unchanged vertices on changed lines are missing from the *vertex-Map* function.

   Counter-measures: Using a better differencing algorithm, such as a tree-based differencing approach (e.g. GumTree [52]) or clone analysis.

3. Internal validity (experimenter bias) : the estimated spurious path rate depends on our judgment and may be subject to experimenter bias. Other experimenters may find a slightly different spurious path rate.

   Counter-measures: It would be desirable to conduct formal studies involving multiple external experimenters.

4. Internal validity (selection bias) : we selected one system based on the availability of its source code and history of security changes.

   Consequences: Our conclusions depend on the change history of this single system.

   Counter-measures: In further research, we plan to include other systems in our analysis.

5. External validity (Generalizability): our study is on a single open source content management system implemented in PHP and may not be reproducible on other systems. Though our PTFA-based approach is language-independent, we require a front-end in

order to handle projects in languages other than PHP.

Consequences: Our conclusions depend on the change history of this single system

Counter-measures: In further research, we plan to include other systems in our analysis.

5.7 Conclusion and Future Work

In this study, we showed how to automatically leverage PTFA model checking automaton extracted from source code to determine privilege protection losses and calculate examples. We determined privilege protection losses by comparing PTFA state reachability for statements shared in release pairs. We calculated short examples by reversing, removing and adding edges to PTFA automata.

Over 147 release pairs, we found 14,116 privilege protection loss examples. They were spread over 31 release pairs. In these, the median is 211 and the mean of 455.4 examples per release pair. We found that the median privilege protection loss example is ≤ 88 states long, in a single file, and crosses one function boundary. However, our results show a long tail, with examples reaching up to 364 states long, spread across six files, and crossing five function boundaries.

In our future research, we would investigate the performance of our analysis on a simulated set of security changes. To do so, we would create a test bench by mutating, injecting and suppressing privilege checks in programs like WordPress.

We also intend to develop visualizations showing how a given privilege protection change came to be. We would especially like to show which path(s) were added or removed and its impact on the privilege protection.

As mentioned earlier, validation of the presented approach should be performed by conducting reviews with external experts – preferably WordPress developers. We would examine our results' accuracy, determine if privilege protection losses violated security assumptions and if privilege protection loss examples were helpful in determining so.

Finally, we would also like to extend our analysis to other systems, written in PHP or other programming languages.

Acknowledgements

The authors would like to acknowledge Mathieu Lavallée, who provided constructive feedback on this article, and Mathieu Mérineau for his support with the analysis tools and the related discussions.

CHAPTER 6 ARTICLE 3: DETECTION OF PROTECTION-IMPACTING CHANGES DURING SOFTWARE EVOLUTION

Marc-André Laverdière and Ettore Merlo

Published in the Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2018), 2018, pp. 434–444. Minor corrections added.

Abstract

Role-Based Access Control (RBAC) is often used in web applications to restrict operations and protect security sensitive information and resources. Web applications regularly undergo maintenance and evolution and their security may be affected by source code changes between releases. To prevent security regression and vulnerabilities, developers have to take re-validation actions before deploying new releases. This may become a significant undertaking, especially when quick and repeated releases are sought.

We define *protection-impacting changes* as those changed statements during evolution that alter privilege protection of some code. We propose an automated method that identifies *protection-impacting changes* within all changed statements between two versions. The proposed approach compares statically computed security protection models and repository information corresponding to different releases of a system to identify *protection-impacting changes*.

Results of experiments present the occurrence of *protection-impacting changes* over 210 release pairs of WordPress, a PHP content management web application. First, we show that only 41% of the release pairs present *protection-impacting changes*. Second, for these affected release pairs, *protection-impacting changes* can be identified and represent a median of 47.00 lines of code, that is 27.41% of the total changed lines of code. Over all investigated releases in WordPress, *protection-impacting changes* amounted to 10.89% of changed lines of code. Conversely, an average of about 89% of changed source code have no impact on RBAC security and thus need no re-validation nor investigation.

The proposed method reduces the amount of candidate causes of protection changes that developers need to investigate. This information could help developers re-validate application security, identify causes of negative security changes, and perform repairs in a more effective way.

Index Terms

Security Impact of Changes, Role Based Access Control, Static Analysis, Software Evolution, Software Maintenance

6.1 Introduction

Web applications are now entrusted with sensitive data and operations. Thus, they must comply with legal and organizational security requirements. Web applications are expected to ensure confidentiality, integrity, and availability.

These security expectations are summarized in industry standards, such as the OWASP Application Security Verification Standard, whose purpose is “to define what a secure application is.” (OWASP [144, p.5]). This standard covers *access control* and other security topics. Access control ensures that only the right users execute some specific code.

Web applications may suffer from vulnerabilities, which are defined as “a flaw or weakness in a system’s design, implementation, or operation and management that could be exploited to violate the system’s security policy.” (RFC 4949 [165]). An access control vulnerability allows the violation of a resource’s security policy (e.g. an unauthorized data access or performing unauthorized operations) [131, 143].

Role-based access control (RBAC) is commonly used for access control. RBAC belongs to a family of access control called authentication-based access control family (NBAC) [91]. In RBAC, developers assign *privileges* to one or more *roles* as necessary and insert privilege checks across the code base according to the application’s security needs.

Developers need quality assurance processes to prevent such issues from occurring and ensuring that their code changes only have a planned and desirable impact on security. They need quality assurance processes that encompass verification and validation. These often leverage testing [13, 53, 125], reviews [125, 148] and formal methods [152]. All these verification and validation approaches require developers’ effort and are typically time consuming. They also have limitations and thus they should be combined for better results [18, 49].

Like other kinds of software, web applications undergo maintenance and evolution. Software maintenance can be corrective, adaptive, perfective, and preventive [83]. Corrective and adaptive maintenance require an effort of about 30-50% [46, 109]. Maintenance activities oriented towards security [38] are included in the remaining 50-70% of effort. They include vulnerability correction, modifying security checks to accommodate RBAC policy changes and malicious changes from insiders [34]. RBAC policy changes involve adding, removing and renaming roles and privileges. These activities intentionally change the security protection of different parts of a system.

Many maintenance activities are not specifically meant to modify the security of the application. These include bug-fixing, refactoring, design changes, and changing the functionality of the application. These changes may unintentionally introduce security flaws [97]. Thus, code changes may impact the security protection in the application in different and seemingly unrelated parts of the application. This occurs because of added, removed or modified execution paths.

Because RBAC security issues are hard to reason about, developers should ideally verify and validate that each change is free from security regressions throughout the project's evolution. This would enable rapid correction, as well as avoiding complex regressions due to the combined effect of many changes. However, a constant re-verification of the whole application would be very effort-intensive. Even if individual changes alter a small part of the application, "software modified during maintenance should be subjected to the same review as newly developed software" (Landwehr et al. [97]). As developers increasingly adopt frequent and quick release cycles, this overhead represents a major hurdle.

Static analysis tools are known to support security reviews. For instance, a study of three industrial C++ projects estimated cost reduction of 17% for reported security bugs [19].

We propose an automated static analysis over privilege protection that automatically classifies changed lines of code as protection-impacting or non-protection-impacting. This is intended to safely reduce the amount of changes that developers must assess before a release. Our analysis finds *protection-impacting changes* (PIC), the subset of the code changes which contains the root causes of security changes related to privilege protection. This analysis relies on the automated detection of *definite protection differences (DPD)*, which are changes in statements' definite privilege protection (i.e. privilege protection changes [101, 103]).

We show that *protection-impacting changes* can be identified and are globally a small proportion of the total number of changes in a release. Our method reduces the pool of candidate causes of security changes that need to be investigated. Thus, our approach could help developers eliminate root causes of negative security changes effectively. Our main contributions are: (i) a method to automatically identify *protection-impacting changes* during software evolution; and (ii) a longitudinal survey of the prevalence of protection-impacting changes over 210 release pairs of WordPress, a popular PHP web-based content management system.

This article is organized as follows. In Section 6.2, we cover the background necessary to understand our contributions. Then, in Section 6.3, we define protection-impacting changes.

Afterwards, we show an example in Section 6.4 and describe our experiments in Section 6.5. Then, we share and discuss our results in Section 6.6 and 6.7, respectively. We describe related work in Section 6.8 and consider threats to validity in Section 6.9. We bring concluding remarks in Section 6.10.

6.2 Background

6.2.1 Pattern Traversal Flow Analysis

Our research objectives revolve around privilege protection, especially definite privilege protection. Pattern-Traversal Flow Analysis (PTFA) [57, 61, 111], is a domain-specific model-checking approach to determine privilege protection. PTFA verifies the property satisfaction that a specific code pattern was definitely executed on all paths reaching a statement s. Patterns are privilege verification checks for some privilege $priv \in Privileges$.

Protected PTFA states associated to a vertex w in the control flow graph (CFG) represent the existence of a path from v_0 to w in which the privilege $priv$ has been granted. PTFA states are represented as $q_{i,j,k}$, where i is the corresponding CFG vertex identifier. The flags j and k refer to the calling context's and the intra-procedural protectedness, respectively.

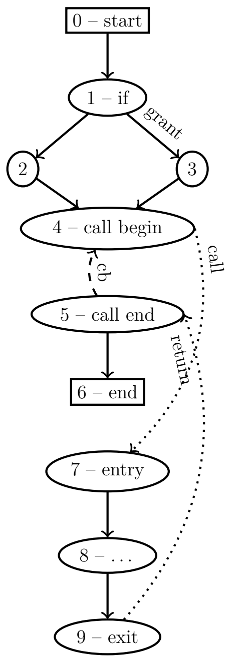
PTFA models are generated automatically and statically from the application's source code. This generation is based on graph rewriting rules applied on the application's CFG [111].

We illustrate the CFG into a PTFA model in Figure 6.1. Please note that, by convention, the i index of PTFA states $q_{i,j,k}$ refers to the CFG vertex number. In this example, we show the effect of grant and interprocedural edges in the CFG. Please note that we represented unreachable states in grey in this figure to facilitate comprehension.

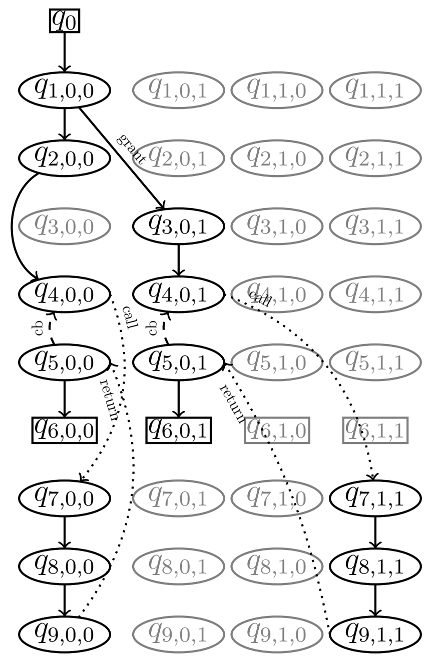
Definite privilege protection $DefProt(w)$ refers to privileges verified on all interprocedural paths prior to the execution of a statement (Equation 6.1). Without PTFA, it is difficult to determine the definite privilege protection over large systems with a complex interprocedural call graph. The predicate $Prot(v, p, priv)$ represents the protectedness of v for privilege $priv$ on a path p from the initial vertex v_0.

$$\begin{gathered} DefProt(w, priv) = \forall p\,(v_0, \ldots, w) \;\mid Prot(w, p, priv) \equiv \\ \nexists p'\,(v_0, \ldots, w) \mid \neg Prot(w, p', priv) \end{gathered} \tag{6.1}$$

In contrast, PTFA indicates definite privilege protection thanks to state reachability. All states present in the model are reachable from q_0. For CFG vertex w, a protected state $q_{w,j,1}$ exists only when there exists a protected path between v_0 and w. And an unprotected $q_{w,j,0}$



(a) Example CFG



(b) Corresponding PTFA Model

Figure 6.1 Example CFG and Corresponding PTFA Model – Unreachable States Greyed Out

exists only when there exists an unprotected path between v_0 and w. Thus, definite privilege protection for w is determined by the existence of protected states corresponding to w and the absence of unprotected states corresponding to w (Equation 6.2).

$$DefProt(w, priv) = \exists q_{w,j_1,k_1} \in Q_{\mathcal{A}} \ \mid \ k_1 = 1 \ \wedge \ \nexists q_{w,j_2,k_2} \in Q_{\mathcal{A}} \ \mid \ k_2 = 0 \tag{6.2}$$

6.2.2 Definite Protection Differences

A Definite Protection Difference (DPD) occurs when the definite privilege protection for a statement s (which is common to versions Ver_a and Ver_b) differs between versions.

We consider that statement s is *loss-affected* for privilege *priv*, if it was *priv* protected in Ver_a and is no longer *priv* protected in Ver_b (c.f. *lossAffected* in Equation 6.3). A similar definition applies for *gain-affected* statements, whenever statement s was granted a *priv* protection in Ver_b that was not there in Ver_a (c.f. *gainAffected* in Equation 6.4). Please note that statement s may be loss-affected with regards to privilege p_1, but gain-affected with regards to privilege p_2.

$$lossAffected(priv) = \left\{ v_b \;\middle|\; \begin{array}{c} (v_a, v_b) \in vertexMap \wedge \\ DefProt(v_a, priv) \;\wedge \neg DefProt(v_b, priv) \end{array} \right\} \tag{6.3}$$

$$gainAffected(priv) = \left\{ v_b \;\middle|\; \begin{array}{c} (v_a, v_b) \in vertexMap \wedge \\ \neg DefProt(v_a, priv) \;\wedge DefProt(v_b, priv) \end{array} \right\} \tag{6.4}$$

6.3 Method

Our long-term research objective is the identification of root causes for definite protection differences, a non-trivial problem. Thus, we compute a superset of the root causes, which we call *protection-impacting changes. Gain-impacting changes* are protection-impacting changes that caused at least one statement to be gain-affected, while *loss-impacting changes* are conversely defined (c.f. Section 6.2.2). These labels are not mutually exclusive, since some code changes may belong to both kinds of definite protection differences (e.g. changing a security check from privilege p_1 to privilege p_2.)

We define two criteria for protection-impacting changes in Sections 6.3.1 and 6.3.2. We then define protection-impacting changes in Section 6.3.3 based on these two criteria.

6.3.1 Added and Deleted Edges

The first criterion for protection-impacting changes is that they correspond to added and deleted edges in PTFA graphs.

We follow these steps to obtain added and deleted PTFA edges. First, we use line-level source code differencing and heuristics to create $vertexMap$, a mapping between the CFG vertices of the two versions. Using this mapping, we determine graph differences between PTFA models. Graph differences include changed edges, which are important because definite protection is a predicate over program paths, and thus over sequences of edges.

From this heuristic mapping at the CFG level, we compute the changes between PTFA models as follows. Q_a and T_a respectively are the set of states and the set of transitions in the PTFA model for version Ver_a. Q_b and T_b are similarly defined for the PTFA model for Ver_b. The predicate $deletedState(q_{i,j,k})$ (Equation 6.5) is true whenever $q_{i,j,k} \in Ver_a$ has no corresponding state in Ver_b. Its converse, *addedState* (Equation 6.6), is true whenever

$q_{i,j,k} \in Ver_b$ has no corresponding state in Ver_a. The symbols *dom* and *image* correspond to the function's domain and image, respectively.

deletedEdges is the set of deleted edges in the PTFA model for Ver_a (Equation 6.7). It contains all edges which either connect one or more deleted states or for which no corresponding edge is present in the PTFA model for Ver_b. Conversely, *addedEdges* is the set of added edges in to PTFA model for Ver_b (Equation 6.8). The function $bMap$ is the reverse-function of $vertexMap$.

$$deletedState(q_{i,j,k}) \doteq \ i \notin dom(vertexMap) \ \vee \ q_{vertexMap(i),j,k} \notin Q_b \tag{6.5}$$

$$addedState(q_{i,j,k}) \doteq \ i \notin image(vertexMap) \ \vee \ q_{vertexMap^{-1}(i),j,k} \notin Q_a \tag{6.6}$$

$$\begin{gathered} deletedEdges \doteq \\ \left\{ (q_{i_1,j_1,k_1} \ , \ q_{i_2,j_2,k_2}) \in T_a \ \middle| \ \begin{matrix} deletedState(q_{i_1,j_1,k_1}) \ \vee deletedState(q_{i_2,j_2,k_2}) \vee \\ \left(q_{vertexMap(i_1),j_1,k_1} \ , \ q_{vertexMap(i_2),j_2,k_2}\right) \notin T_b \end{matrix} \right\} \end{gathered} \tag{6.7}$$

$$\begin{gathered} addedEdges \doteq \\ \left\{ \left(\ q_{i_1,j_1,k_1} \ , \ q_{i_2,j_2,k_2} \ \right) \in T_b \ \middle| \ \begin{matrix} addedState(q_{i_1,j_1,k_1}) \ \vee \ addedState(q_{i_2,j_2,k_2}) \vee \\ \left(\ q_{bMap(i_1),j_1,k_1} \ , \ q_{bMap(i_2),j_2,k_2} \ \right) \notin T_a \end{matrix} \right\} \end{gathered} \tag{6.8}$$

6.3.2 Appropriately Protected Paths

The second criterion for protection-impacting changes is that they belong to *appropriately* protected paths to a definite protection difference. This means that the path connects to a definite protection difference and that the path has the appropriate protection.

First, protection-impacting changes are related to a) changes in paths leading to security sensitive operations and b) their traversal of security granting patterns, as computed by PTFA.

Let us consider a definite protection difference affecting v_a and v_b, where $v_b = vertexMap(v_a)$. Protection-impacting changes must belong to paths having changed privilege protection for v_a and v_b and precede the execution of v_a and v_b on these paths. In other words, a protection-impacting change v_{PIC} must belong to a path $(v_0, \ \dots \ , v_{PIC}, \ \dots \ , v_a)$ in Ver_a or $(v_0, \ \dots \ , v_{PIC}, \ \dots \ , v_b)$ in Ver_b.

These paths may be entirely new paths in Ver_b, entirely deleted paths in Ver_a or paths modified between Ver_a and Ver_b such that their protectedness differs.

Secondly, protection-impacting changes belong to *appropriately* protected paths to a definite protection difference. This means that the protection-impacting changes will either belong to unprotected or protected paths in Ver_a and unprotected or protected paths in Ver_b, depending on whether the code is gain-affected or loss-affected.

Given CFG vertices v_a and v_b, where $v_b = vertexMap(v_a)$, the following properties hold true when they are loss-affected for privilege *priv*, based on the definition of definite protection.

First, v_a is definitely protected for *priv*, which means that all paths to v_a are *priv*-protected. Second, v_b is not definitely protected for *priv*, which means that either it is unreachable (dead code) or there is at least one unprotected path for *priv* leading to it (Equation 6.9).

$$\begin{gathered} \forall p\,(v_0,\ldots,v_a) \;\mid\; Prot(v_a,p,priv) \;\wedge \\ (\exists p'\,(v_0,\ldots,v_b)\,, \neg Prot(v_b,p',priv) \;\vee\; \nexists p''\,(v_0,\ldots,v_b)) \end{gathered} \tag{6.9}$$

As we saw in Section 6.2.1, these universally and existentially quantified predicates correspond to safety and liveness property verification in PTFA models. Equation 6.9 thus translates to Equation 6.10. Please note that *Reach* is a predicate refers to forward-reachability in the PTFA model. We define the *posProtStateExists* and *negProtStateExists* predicates in Equation 6.11. For Equation 6.10 to hold true, we only need to consider the protected paths to $q_{a,j,1}$ and (if any) the unprotected paths to $q_{b,j',0}$. Thus, protection-impacting changes must belong to these paths.

$$\begin{multlined} posProtStateExists(a) \;\wedge\; \neg negProtStateExists(a) \;\wedge \\ (negProtStateExists(b) \;\vee\; \nexists j,k \;\mid\; \neg Reach(q_{b,j,k})) \end{multlined} \tag{6.10}$$

$$\begin{aligned} posProtStateExists(i) &\doteq Reach(q_{i,0,1}) \vee Reach(q_{i,1,1}) \\ negProtStateExists(i) &\doteq Reach(q_{i,0,0}) \vee Reach(q_{i,1,0}) \end{aligned} \tag{6.11}$$

The situation is similar when dealing with gain-affected CFG vertices. First, v_a is not definitely protected for *priv*, which means that either it is unreachable (dead code) or there is at least one unprotected path for *priv* leading to it. Second, v_b is definitely protected for *priv*, which means that all paths to v_b are *priv*-protected (Equation 6.12).

$$\begin{gathered} \forall p\,(v_0,\ldots,v_b) \;\mid\; Prot(v_b,p,priv) \;\wedge \\ (\exists p'\,(v_0,\ldots,v_a) \;\mid\; \neg Prot(v_a,priv) \;\vee\; \nexists p''\,(v_0,\ldots,v_a)) \end{gathered} \tag{6.12}$$

Equation 6.12 thus translates to Equation 6.13. For Equation 6.13 to hold true, we only need to consider the protected paths to $q_{b,j,1}$ and (if any) the unprotected paths to $q_{a,j',0}$. Thus, protection-impacting changes must belong to these paths.

$$\begin{gathered} posProtStateExists(b) \ \wedge\ \neg negProtStateExists(b) \wedge \\ (negProtStateExists(a) \ \vee\ \nexists j,k \ \mid\ \neg Reach(q_{a,j,k})) \end{gathered} \tag{6.13}$$

6.3.3 Definition of Protection-Impacting Changes

As we mentioned earlier, protection-impacting changes are code changes which belong to an appropriately protected path leading to a definite protection difference. Thus, the definition of protection-impacting changes varies between gain-affected and loss-affected code.

For loss-affected code, the protection-impacting changes PIC_{loss} are the deleted edges belonging to positively-protected paths to v_a and the added edges belonging to negatively-protected paths to v_b (Equation 6.16). This definition depends on *posProtEdges* (Equation 6.14) and *negProtEdges* (Equation 6.15), which return the set of edges belonging to paths between q_0 and, respectively, protected and unprotected states. Please note that $ReachEdges(q_{i,j,k})$ is a function returning all edges between the initial state q_0 and $q_{i,j,k}$.

$$posProtEdges(i) \doteq \bigcup_{j\in\{0,1\}} ReachEdges(q_{i,j,1}) \tag{6.14}$$

$$negProtEdges(i) \doteq \bigcup_{j\in\{0,1\}} ReachEdges(q_{i,j,0}) \tag{6.15}$$

$$\begin{aligned} &posProtDel \doteq deletedEdges \cap posProtEdges(v_a) \\ &negProtAdd \doteq addedEdges \cap negProtEdges(v_b) \\ &PIC_{loss} \doteq (posProtDel\ ,\ negProtAdd) \end{aligned} \tag{6.16}$$

Similarly, for gain-affected code, protection-impacting changes PIC_{gain} are the deleted edges belonging to negatively-protected paths to v_a (*negProtDel*) and the added edges belonging to positively-protected paths to v_b (*posProtAdd*) (Equation 6.17).

$$\begin{aligned} &negProtDel \doteq deletedEdges \cap negProtEdges(v_a) \\ &posProtAdd \doteq addedEdges \cap posProtEdges(v_b) \\ &PIC_{gain} \doteq (negProtDel\ ,\ posProtAdd) \end{aligned} \tag{6.17}$$

6.4 Example

We now provide a ficticious example of definite protection differences caused by a code change in Figure 6.2. Please note that we combined the CFGs of Ver_a and Ver_b, as well as their PTFA models, due to space constraints.

The code change (Figure 6.2a) adds a call to function `foo()` from an unsecured context. Previously, `foo()` was exclusively called from protected contexts. Following that change, the code in `foo()` (vertices 5 to 7) becomes loss-affected.

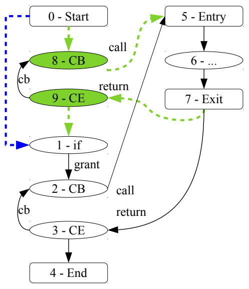
The CFGs (Figure 6.2b) are combined and we show differences using colour and dashed edges. In order to avoid confusion, and contrary to the representation in Figure 6.1, we represented all other edges solid. The blue dashed edge is a deleted edge from Ver_a. We use green for added vertices and edges Ver_b. Counterintuively, adding code caused a deleted edge, since the control flow no longer flows directly from vertex 0 to 1.

The PTFA models (Figure 6.2c) are similarly combined and coloured. We greyed out unreachable states. We also marked all protection-impacting edges with "**PIC**".

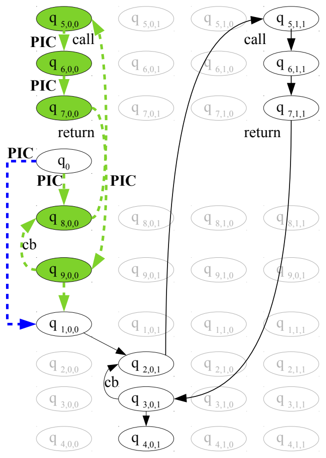
Please note that not all added and deleted edges in our example are protection-impacting. For instance, the added edge $(q_{9,0,0}\ , q_{1,0,0})$ is not protection-impacting, since it does not belong to an appropriately protected path to vertices 5 to 7.

```
+ foo();
  if (!current_user_can('p'))
    die();
  foo()
```

(a) Code Change. Added Code Shown with '+'.



(b) Corresponding CFGs. Added Vertices and Edges in Green. Deleted Edge in Blue.



(c) Corresponding PTFA Models. Added Vertices and Edges in Green. Deleted Edge in Blue.

Figure 6.2 Example Code, CFG and PTFA Model

6.5 Experiment Design

Having defined protection-impacting changes, we now describe our research question and experiments.

Our research question is the following:

RQ *Is the size of protection-impacting changes smaller than the set of code changes?*

Our goal is to quantify the amount and proportion of code changes that are protection-impacting in real systems.

This research question relates to our research goal to focus developers' attention during re-validation. This is achieved when the set of protection-impacting changes is a proper subset of code changes.

6.5.1 Approach

To determine if protection-impacting changes are smaller than code changes in real software projects, we conduct a survey [156] of one open source project, WordPress. In this survey, we determine code changes and protection-impacting changes between major releases, as detailed in Subsection 6.5.2.

We chose this application because it is a widely-used software system which uses RBAC and has a long release history. WordPress is a popular web-based content management system implemented in PHP. Our study encompasses all releases of WordPress available as of March 2017 and which used RBAC – 2.0 to 4.7.3. This application's PHP code, in physical lines of code (LOC), ranges from roughly 35 KLOC in release 2.0 to 340 KLOC in release 4.7.3. For the same releases, the combined HTML, JavaScript and CSS code amounted to roughly 13 KLOC and 179 KLOC, respectively.

Our process consists of a PHP front-end, a PTFA engine, a DPD classifier and a protection-impacting change analyzer. The PHP front-end feeds CFGs to the PTFA engine, which then generates PTFA models. The DPD classifier determines definite protection differences using code differences and definite privilege protection. Finally, the protection-impacting change analyzer uses the DPD data, code differences and PTFA models to determine protection-impacting changes.

6.5.2 Data Collection and Processing

Our approach relies on comparing pairs of versions, and we must choose *which* versions. As an initial experiment, we choose to analyze official final releases (i.e. non-alpha, non-beta, non-release candidate versions). A finer-grained (e.g. commit-level) analysis is possible, which we discuss in Section 6.7. Since these are the versions that end-users normally deploy, we analyze the official release archives [188]. These archives are bundled with files that are not present in the repository. Additionally, we expect that these releases will be better tested and reviewed than intermediary versions.

Because WordPress developers maintain and apply security fixes to multiple releases in parallel, we cannot consider releases sequentially. We compared release pairs according to their semantic versioning [149] and release date information. We organize release pairs in a tree. The edges of that tree are *release pairs*, which are the releases we compare. This is the same approach we used in our earlier studies [101, 103].

6.5.3 Measures

In our survey, we record release pairs affected by definite protection differences. For those that are, we measure protection-impacting changes.

In the presented experiments, the independent variables are the program we analyze, which of its versions we analyze, and the software running the experiment. The dependent variables are code differences and protection-impacting changes.

Please note that we have no access to an oracle that classifies definite protection differences (e.g. classifying as beneficial, harmful or irrelevant.) Consequently, we report protection-impacting change metrics for all definite protection differences. We discuss oracles in more detail in Section 6.7.

We chose to report our metrics at line granularity, because measures in terms of lines of code are the de-facto standard in the industry. This is evidenced by popular differencing and version control tools (e.g. `diff` and `git`), which report code changes at a line granularity by default. In the experiments, we project the PTFA edges to lines of code while taking line change information in consideration.

Respectively, $PIC_{loss,lines}$ and $PIC_{gain,lines}$ are the projection of PIC_{loss} and PIC_{gain} to a set of lines. We define PIC (Equation 6.18) as the union of both these projections. Additionally, we combined the PIC's elements into the metric $allPIC$ (Equation 6.19). To simplify the equations below, we use the projection function π_i to extract the ith element of the pair $x = (Ver_a, Ver_b)$.

$$PIC = \Big(\pi_1(PIC_{loss,lines}) \cup \pi_1(PIC_{gain,lines}) \ , \pi_2(PIC_{loss,lines}) \cup \pi_2(PIC_{gain,lines}) \Big) \tag{6.18}$$

$$allPIC = |\pi_1(PIC)| \ + \ |\pi_2(PIC)| \tag{6.19}$$

The relative size of protection-impacting changes per release pair is the ratio between the total protection-impacting lines and the total modified lines of code (Equation 6.20). In the equations below, *delLines* and *addLines*, respectively are the number of lines deleted from Ver_a and added to Ver_b.

We define *security-affected*, *loss-affected* and *gain-affected* release pairs. Release pairs with definite protection differences are *security-affected*. *Gain-affected* and *loss-affected* release pairs have gain-affected and loss-affected code, respectively.

$$\begin{aligned}
relPIC_{loss} &= \left(\frac{\left|\pi_1(PIC_{loss,lines})\right|}{delLines} , \frac{\left|\pi_2(PIC_{loss,lines})\right|}{addLines} \right) \\
relPIC_{gain} &= \left(\frac{\left|\pi_1(PIC_{gain,lines})\right|}{delLines} , \frac{\left|\pi_2(PIC_{gain,lines})\right|}{addLines} \right) \\
relPIC &= \left(\frac{|\pi_1(PIC)|}{delLines} , \frac{|\pi_2(PIC)|}{addLines} \right) \\
relAllPIC &= \frac{|\pi_1(PIC)| \ + \ |\pi_2(PIC)|}{delLines \ + \ addLines}
\end{aligned} \tag{6.20}$$

6.6 Experimental Results

We implemented our processing workflow using Java. We used a system powered by an i7950@3.07GHz CPU. We configured the Oracle Java VM version 1.8.0_66 to use up to 8Gb of RAM. To compute *delLines* and *addLines*, we used `diff` from GNU diffutils 3.3 and `diffstat` 1.61.

In order to answer "**RQ** *Is the size of protection-impacting changes smaller than the set of code changes?*", we computed protection-impacting changes for 210 release pairs of WordPress. For all release pairs of WordPress, our analysis completed in 27.5 hours, an average of 7.80 minutes per release pair. We have 85 gain-affected release pairs and 46 loss-affected release pairs. Combined, this represents 87 release pairs affected by definite protection differences. We found 258997 protection-impacting LOC (*allPIC*) globally.

We present summary statistics of protection-impacting changes per release pair in Table 6.1. We report the measures that we detailed in Section 6.5.3. We report two distributions: the distribution for all security-affected release pairs and the distribution for all release pairs. We present the latter distribution, because it gives a picture of protection-impacting changes over the lifetime of the project.

We turn our attention to the statistics of $allPIC$ and $relAllPIC$ for security-affected pairs in Table 6.1. On average, security-affected release pairs had 2976.98 LOC (26.28%) protection-impacting lines of code. The median value is different from the mean, with 47.00 LOC (27.41%) protection-impacting changes. This difference is due to outliers, which we discuss below.

If we separate the PIC results according to code deleted (Ver_a) and added (Ver_b), we obtain the respective medians 14.00 and 34.00 LOC. In relative terms ($relPIC$), these terms represent 31.58% and 24.00% of their respective code changes. The mean code changes are 974.41 LOC (30.76%) and 2002.56 LOC (25.07%) for Ver_a and Ver_b, respectively. In other words, deleted code is more likely to be protection-impacting than added code.

Over all release pairs in our survey, protection-impacting changes are an even smaller percentage of code changes. The median is not significant, since most release pairs have no definite protection differences, and therefore no protection-impacting changes. However, the mean protection-impacting changes represents 10.89% of code changes.

We show the histogram of protection-impacting changes per release pair for security-affected release pairs in Figure 6.3. In this histogram, we report protection-impacting changes for gains (PIC_{gain}), losses (PIC_{loss}), their combined measure (PIC) and the combined metric $allPIC$. Please note that protection-impacting changes may be both gain-impacting and loss-impacting. We used a non-linear X axis, with bins of different widths, in order to better represent the distribution.

In Figure 6.3, we observe that, among security-affected release pairs, 50/87 (57%) release pairs have less than 100 lines of code classified as protection-impacting changes. We also observe that 61/87 (70%) security-affected release pairs have less than 500 protection-impacting lines of code. Only 26/87 (30%) release pairs have 500 or more lines of protection-impacting changes. The distribution is long-tailed in reality. We observe outliers on the right-hand side of the distribution. We found 8/87 (9%) release pairs for which $allPIC > 10,000$. All these release pairs have a minor or major release number (i.e. a .0 release) for Ver_b. The volume of code changes for these releases is very high, with a mean of over 31 KLOC and 16 KLOC added and deleted lines, respectively. Thus, the relative proportion of protection-impacting

Table 6.1 Protection-Impacting Changes Per Release Pair

Measure	**1st quarter**	**Median**	**Mean**	**3rd quarter**
Security-Affected Release Pairs (87 / 210)				
$PIC_{loss,lines}$ (Ver_a) (LOC)	0.00	2.00	773.20	1065.50
$relPIC_{loss}$ (Ver_a)	0.00%	4.66%	10.86%	15.56%
$PIC_{loss,lines}$ (Ver_b) (LOC)	0.00	2.00	1624.29	1820.50
$relPIC_{loss}$ (Ver_b)	0.00%	3.24%	10.46%	17.64%
$PIC_{gain,lines}$ (Ver_a) (LOC)	6.00	14.00	744.53	677.50
$relPIC_{gain}$ (Ver_a)	9.09%	20.32%	27.48%	37.50%
$PIC_{gain,lines}$ (Ver_b) (LOC)	14.50	34.00	1662.40	1492.50
$relPIC_{gain}$ (Ver_b)	10.66%	22.19%	22.88%	31.73%
PIC (Ver_a) (LOC)	6.00	14.00	974.41	1188.50
$relPIC$ (Ver_a)	10.82%	31.58%	30.76%	42.12%
PIC (Ver_b) (LOC)	14.50	34.00	2002.56	2447.50
$relPIC$ (Ver_b)	11.95%	24.00%	25.07%	34.49%
$allPIC$ (LOC)	23.00	47.00	2976.98	4026.50
$relAllPIC$	12.74%	27.41%	26.28%	36.07%
deleted (LOC)	24.00	109.00	3060.62	4117.50
added (LOC)	75.00	341.00	6446.60	9770.00

Measure	**1st quarter**	**Median**	**Mean**	**3rd quarter**
All Release Pairs (210)				
$PIC_{loss,lines}$ (Ver_a) (LOC)	0.00	0.00	320.32	0.00
$relPIC_{loss}$ (Ver_a)	0.00%	0.00%	4.50%	0.00%
$PIC_{loss,lines}$ (Ver_b) (LOC)	0.00	0.00	672.92	0.00
$relPIC_{loss}$ (Ver_b)	0.00%	0.00%	4.33%	0.00%
$PIC_{gain,lines}$ (Ver_a) (LOC)	0.00	0.00	308.45	9.00
$relPIC_{gain}$ (Ver_a)	0.00%	0.00%	11.38%	16.40%
$PIC_{gain,lines}$ (Ver_b) (LOC)	0.00	0.00	688.71	24.00
$relPIC_{gain}$ (Ver_b)	0.00%	0.00%	9.48%	16.54%
PIC (Ver_a) (LOC)	0.00	0.00	403.69	10.50
$relPIC$ (Ver_a)	0.00%	0.00%	12.74%	18.41%
PIC (Ver_b) (LOC)	0.00	0.00	829.63	24.00
$relPIC$ (Ver_b)	0.00%	0.00%	10.39%	19.51%
$allPIC$ (LOC)	0.00	0.00	1233.32	38.00
$relAllPIC$	0.00%	0.00%	10.89%	22.57%
deleted (LOC)	15.00	42.00	1338.00	182.50
added (LOC)	38.00	117.50	2841.61	482.25

change in these release pairs is not very high, with a mean $relAllPIC = 38.75\%$. This situation explains the gap between the medians and means in Table 6.1.

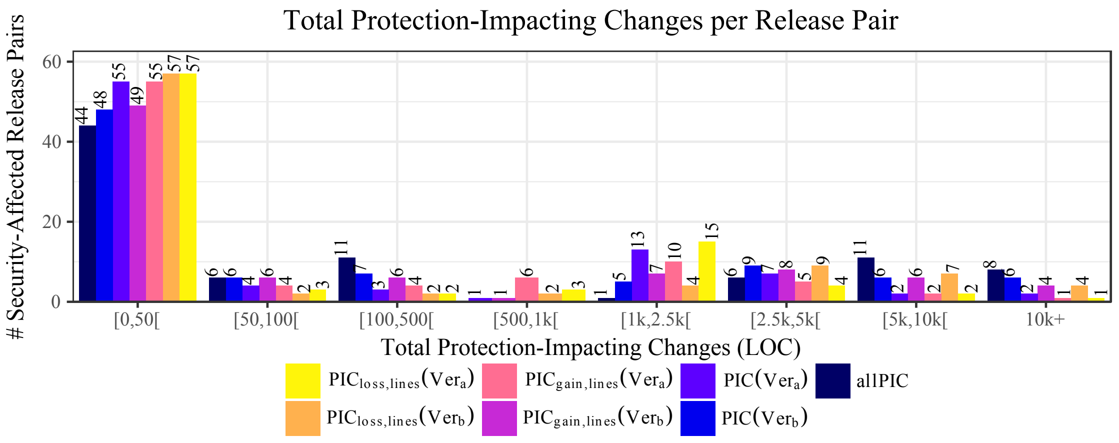


Figure 6.3 Distribution of Protection-Impacting Lines of Code per Release Pair for security-affected Release Pairs

In Figure 6.4, we present an histogram of the relative distributions $relPIC_{loss}$, $relPIC_{gain}$, $relPIC$ and $relAllPIC$. We first observe the large spike at zero in the distribution of $relPIC_{loss}$. In part, this is because 46 release pairs have no loss-affected code in the first place The relative combined measure ($relAllPIC$) is somewhat skewed towards the left. Among security-affected release pairs, 16/87 (18%) release pairs, have less than 10% of changed lines of code classified as protection-impacting changes. Additionally, 42/87 (48%) security-affected release pairs have less than 25% of changed lines of code considered as protection-impacting. We also see that the distribution has 8/87 (9%) security-affected release pairs above 50%. Thus, protection-impacting changes are generally very small compared to the total number of code changes between versions. Two distributions appear long tailed, $relPIC_{gain}$ and $relPIC$ for Ver_a. These occur for release pairs with a small number of changes (<100 LOC) and appear to be very targeted.

As we mentioned earlier, the difference in shape between Figures 6.3 and 6.4 is explained by the large volume of changes. For instance, the releases which have over 5000 protection-impacting changes ($allPIC > 5000$) show a large amount of code changes. In these release pairs, the largest measure of $relAllPIC$ is 54%.

The median of $relAllPIC$ is very encouraging. We further discuss this result in Section 6.7.

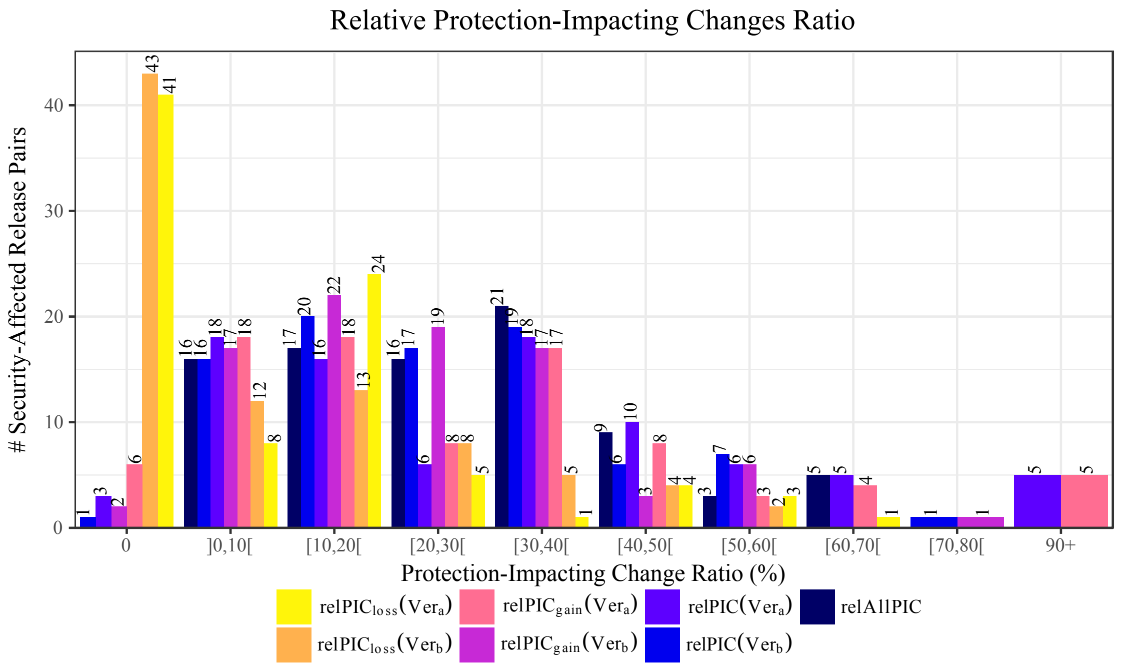


Figure 6.4 Distribution of Protection-Impacting Lines of Code per Release Pair for security-affected Release Pairs

6.7 Discussion

When definite protection differences are detected, developers can focus their review efforts on protection-impacting changes. Since, on average, in security-affected release pairs, only about 26% of code changes are protection-impacting. Our approach gives an opportunity to reduce review efforts, though further research is needed to investigate and confirm this.

Asking our research question "*Is the size of protection-impacting changes smaller than the set of code changes?*", we found that 10.89% of code changes were protection-impacting.

Static analysis tools should reduce the amount of information to review [137]. For security-affected releases, although the mean is large (2002.56 LOC), the median of protection-impacting changes appear small (34.00 LOC).

While our approach does not require one, it may benefit from an oracle that classifies definite protection differences. Using an oracle that classifies differences as beneficial, harmful or irrelevant, our approach could compute protection-impacting changes only for harmful security changes. Otherwise, we could report on how many code changes affected security, and on the distribution of protection-impacting changes populations.

Suitable oracles include formal security specifications, a classification of security-affected lines by developers, or a vulnerability oracle. Thus, when used alongside formal methods, our method pinpoints which code changes are likely to have caused policy violations.

To review our results, we randomly sampled 108 definite protection difference from our survey results. We considered all 97 privileges and 210 release pairs. Manual assessment of these sampled definite protection differences revealed that 81/108 of them were real code changes (75%). The remaining 27/108 (25%) cases are due to infeasible paths.

Our approach could be used as an early warning system for security issues, since it can be used between any two versions of the application. For instance, developers could use it after each commit. Or they could integrate our approach in automated build systems and treat definite protection differences as regressions.

Furthermore, we envision adding our approach to integrated development environments (IDE), which would allow developers to determine definite protection differences and protection-impacting changes interactively. On-demand analysis may also accelerate the assessment of vulnerability reports and the creation of security patches. For instance, a developer receiving a vulnerability report may quickly obtain protection-impacting changes for the vulnerable code segment. Then, she may determine which code changes are responsible for the vulnerability and which correction is needed.

Our approach may be suited for test selection. Ideally, when testing the RBAC implementation for regressions, one would only run the tests executing protection-impacting changes.

Although our results are promising, further research towards identifying root causes is desirable. Fine-grained (e.g. commit-level) surveys of protection-impacting changes should also be performed. Human studies are also needed to confirm the psychological acceptability of our results.

6.8 Related Work

Other researchers used PTFA for evolution studies of RBAC systems. Letarte et al. [112] surveyed privilege protection over 31 phpBB releases, which only handled a binary distinction between administrator and unprivileged users. In our study, we detect protection-impacting changes, handle a richer protection scheme and performed a larger survey.

We previously performed surveys of definite protection difference over WordPress. We defined definite protection differences in a first study [101], and produced counter-examples for loss-affected code. In this article, we defined and detected protection-impacting changes. We also proposed a classification of definite protection differences [103]. In contrast, this paper focuses on the identification of a superset of the root causes of definite protection differences and is not aimed at the classification and distribution of these differences.

Other researchers used formal methods and static analysis tools to analyze RBAC in applications. MARGRAVE [55] is a tool leveraging formal methods to support access control evolution. It translates XACML to decision diagrams and differences the policies. However, MARGRAVE does not take the implementation of the RBAC policy in consideration.

Other approaches use formal static analysis methods for RBAC systems. One system, ROLECAST [167], automatically detects privilege protection checks in PHP and JSP applications. ROLECAST uses inferences to identify critical variables controlling the reachability of security sensitive statements, infers roles and determines if checks are performed consistently. When reporting a potential vulnerability, ROLECAST mentions many details, including the security sensitive statement, its calling context, which security variables are erroneously verified. Our contribution is very different as we rely on graph reachability and code differences to connect definite protection differences to their potential causes.

An extension tool, FIX ME UP [168], statically finds access-control errors of omission and proposes candidate repairs. The approach generates an access control template, detects deviations from this template and changes them to conform. Their algorithm relies on slicing, a more complex algorithm than ours, and does not detect definite protection differences.

RBAC policy evolution is also useful to perform test selection [81]. This approach relies on semantic differencing of XACML policies and requires running the full test suite at least once. Another study [108] used testing to detect hidden and implicit security mechanisms. In comparison, our approach is static and identifies potential root causes for protection changes.

Model checking is also useful to identify candidate causes of errors. Ball et al. [20] proposed an method which identifies transitions in an error trace that are in no correct trace and injects halt statements to produce additional error traces. Groce et al. [66] expressed counter-example generation as a minimization problem whose distance metric considers causal dependence. While we have in common the use of model checking, these approaches do not address privilege protection.

BUGGININGS [166] identifies the cause of a bug in the context of software evolution by differencing dependence graphs. In contrast, our approach targets only definite protection differences. We also determine differences at the PTFA level and rely on reachability in PTFA models.

6.9 Threats to Validity

Threats to internal validity refer to confounding variables that may influence our results. Our results depend on the accuracy of the PTFA engine we used. Because PHP applications like WordPress rely on many dynamic features, the engine relies on sound but conservative approximations, especially for dynamic calls in the call graph, that may lead to spurious paths. In turn, spurious paths may lead to the identification of spurious protection-impacting changes. Consequently, the real set of protection-impacting changes may be smaller than reported. We did not calculate the spurious path rate in this study, but we previously reported a spurious path rate for PTFA on WordPress of 10.96 ± 3.18% (95% confidence level) [101].

Our results also depend on the differencing algorithm we used. We extracted line-level differences between releases using GNU `diff`. Since there may be many CFG vertices on the same line of code, we over-estimate the changed CFG vertices. However, this should only affect our results minimally, since we present our results at a line granularity. Other differencing algorithm such as those supplied by versioning systems could also be used and may produce slightly different results.

Because we are lacking a formal oracle, the precision and recall of our experiments is unknown. Although our informal evaluation of 108 samples is promising, a formal statistical analysis should be performed on a larger sample to assess its significance. Future research should also include a robust evaluation to determine the precision and recall of our approach. This may be achieved with a test bench containing known protection-impacting changes – for instance by mutating privilege checks in representative applications.

Threats to external validity relate to the generalizability of our results. We did not have a vulnerability oracle for our study, though our approach may use one. Consequently, we cannot determine a specific distribution of protection-impacting changes for vulnerabilities. To counter this issue, studies using a vulnerability oracle (e.g. a testbench with known vulnerabilities) should be performed.

Another threat to generalizability is that our study surveys a single open source content management system implemented in PHP. We may obtain different results when studying other systems, whether they are written in PHP or other languages. In the experiments, we used a PHP front-end for WordPress. However, approach itself is reproducible and language-independent. To avoid that our conclusions depend on the change history of this single system, studies that include other systems should be performed.

6.10 Conclusion and Future Work

In order to focus developers' efforts during re-evaluation of RBAC-enabled applications applications, we proposed a method to detect *privilege-protection changes*.

We presented a novel language-independent algorithm to automatically identify protection-impacting changes. This algorithm relies on identifying added and deleted edges between two PTFA models. This information is combined with interprocedural graph reachability information to obtain protection-impacting changes.

We also reported a survey on the prevalence of protection-impacting changes during the evolution of a Web application. We examined 210 release pairs of WordPress and determined that only 87/210 (41%) of them contained *protection-impacting changes*, while the other releases contain changes that do not affect privilege protection.

For security-affected release pairs, *RBAC protection-impacting changes* that may have caused the observed definite protection differences represent a median of 47.00 lines of code (27.41% of total changes). Over all releases, this represents an average of 1233.32 lines of code per release pair (10.89% of changed code). In other words, over WordPress' evolution, the proposed method reduces the number of changed source code lines to review by about 89%.

RQ *Is the size of protection-impacting changes smaller than the set of code changes?*
Using protection-impacting changes, developers would only validate a median of 27.41% and a mean of 26.28% of code changes in security-affected release pairs.

Our research question was "*Is the size of protection-impacting changes smaller than the set of code changes?*" Results allow to answer affirmatively. The identification of protection-impacting changes reduces lines of code to review by 89% on average for all release pairs, or 74% for security-affected release pairs.

According to our manual evaluation of 108 results, these protection-impacting changes are true positives and really affect privilege protection in 75% of cases. The remaining cases represent infeasible paths.

Our promising results hint that our approach is well poised to save considerable effort during security re-validation. Since more than a half of the release pairs have no *protection-impacting changes*, developers may focus their security verification towards security impacted releases.

Developers would need to review security-affected releases. While definite privilege protection information may help in this task, a reduction of candidate root causes is likely to reduce the required effort. In these releases, the median *protection-impacting changes* represent about

27% of total changes. Thus, when re-validating the RBAC security of their applications, developers need not investigate 73% of code changes, because these changes have no impact on protection differences.

Future research includes the investigation of the distribution of *protection-impacting changes* corresponding to vulnerabilities, whenever a security oracle is available. This evaluation would allow us to quantify how many protection-impacting changes caused or fixed vulnerabilities.

Further research could also be devoted to investigating the interaction with developers and the visualization of *protection-impacting changes*. An approach about determining explanatory counter-examples [101] could also be used, combined with visualization techniques to supply information to developers.

As mentioned earlier, further research may include the assessment of the real impact of the proposed techniques to application security verification and validation in collaboration with real human developers. Such studies would measure the developers validation effort during an application evolution and assess the advantages of the proposed approach.

Finally, we would also like to extend the proposed analysis to other systems and to other programming languages.

CHAPTER 7 ARTICLE 4: RBAC PROTECTION-IMPACTING CHANGES : A CASE STUDY OF PHP APPLICATION EVOLUTION

Marc-André Laverdière and Ettore Merlo

Submitted to IEEE Transactions on Software Engineering. Minor corrections added.

Abstract

Web applications often use Role-Based Access Control (RBAC) to restrict operations and protect security sensitive information and resources. Web applications' RBAC security may be affected by source code changes between releases. Developers should re-validate their application prior to release, which may become resource-intensive. We define *protection-impacting changes* (PIC), changed statements during evolution that potentially alter privilege protection. Our automated and static PIC identification relies on graph reachability and differencing of two versions' protection models. We surveyed two PHP web applications, WordPress and MediaWiki. We found PICs in only respectively 87/210 (41%) and 42/192 (22%) release pairs. For these release pairs, median PICs are 27.41% and 13.85% of the code changes. Over all release pairs, these applications respectively have 90.89% and 97.38% of source code changes are not protection-impacting. We also propose a method identifying source code changes containing root causes of definite protection differences. By reverting these changes, we found that all root causes of protection differences for a release pair were conservatively identified in 87% to 93% of examined pairs. This method reduces the amount of candidate causes of protection differences that developers need to investigate. PICs could help developers re-validate application security, identify causes of security changes, and perform repairs.

Index Terms

F.3.2.f Program analysis, D.4.6 Security and Privacy Protection, D.4.6.a Access controls, D.4.6.g Verification, K.4.4.f Security, K.4.4.g Internet security policies

7.1 Introduction

We routinely interact with web applications. They are now entrusted with sensitive data and operations and must comply with legal and organizational security requirements. Web applications are expected to ensure confidentiality, integrity and availability. These security expectations are summarized in industry standards, such as the OWASP Application Security Verification Standard 3.0, whose purpose is "to define what a secure application is." [144]. This standard covers *access control* among other security topics.

"Access control is the mechanism by which services know whether to honor or deny requests." [91] In other words, access control enforces the web application's access control policy. This policy specifies which users may execute specific operations and access resources. Resources are defined in an application-specific manner. By implication, access control determines which users may execute specific code. We illustrate these concepts with an example

Let us consider an imaginary e-commerce application's access control policy. The user categories are customers, customer service representatives and order fulfillment staff. The resources are customer orders and the information on these orders (e.g. order number, shipping information, billing information, items and prices). Customers are allowed to view their own orders. They may also cancel and modify unshipped orders. Customer service representatives are allowed to view any order. They can also cancel and modify any unshipped orders. The policy also states that order fulfillment staff are allowed to view a subset of the order (order number, shipping information and items) and update the shipping information.

Web applications may suffer from vulnerabilities, which are defined as "a flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy" [165]. Therefore, an access control vulnerability allows the violation of the access control policy.

Let us consider three access control vulnerabilities in our example. First, the application is vulnerable if a customer service agent updates the shipping status. Second, the application is vulnerable if a customer is able to view other users' orders. Finally, the application is vulnerable if an order fulfillment employee views a customer's billing information.

Role-based access control (RBAC) is commonly used for access control. RBAC is an authentication-based access control [91]. In RBAC, developers assign *privileges* to one or more *roles* as necessary. In the systems we analyzed, developers insert privilege checks across the code base according to the application's security needs. In order to prevent RBAC vulnerabilities, software development processes need to integrate RBAC implementation Verification and Validation (V&V) to their quality assurance practices. To ensure that their code changes only have a planned and desirable impact on security, maintainers may use testing [13, 53, 126], reviews [126, 148] and formal methods [152].

Notable security testing techniques are *security regression testing* [53] and *penetration testing* [13, 126] techniques. In general, security testing is known to be hard because vulnerabilities are typically hard to expect side-effect behavior [172]. Because it may be difficult to create RBAC policy tests, developers may consider generating them from functional tests [135].

Security reviews [126] and inspections [148] typically include code reviews and architectural analysis. Inspections additionally investigate that requirements and security goals are met. Reviews are resource intensive and do not find all vulnerabilities. For instance, a study by Edmunston et al. found that 15 developers have about a 95% chance of finding all known vulnerabilities in a PHP web application [49].

The goal of formal and semi-formal validation methods is to ensure that changes comply with the security policy [152]. Because specifying access control policies is time-consuming, developers may consider inferring or extracting policies from code. For instance, it is possible to generate Secure UML policy models from PHP source code [7, 59].

All these verification and validation approaches require developers' effort and are typically time consuming. Additionally, developers should use multiple V&V approaches for better results [18].

Like other kinds of software, web applications undergo maintenance and evolution. Software maintenance can be corrective, adaptive, perfective and preventive [83]. Corrective and adaptive maintenance require an effort of about 30-50% [46, 109]. Maintenance activities oriented towards security [38] are included in the remaining 50-70% of effort.

Non-security maintenance activities include bug-fixing, refactoring, design changes, and changing the functionality of the application. They may unintentionally introduce security flaws [97]. These unintended security impacts may be in different and seemingly unrelated parts of the application. This occurs because of added, removed or modified execution paths.

Security maintenance activities include vulnerability correction, malicious changes from insiders [34] and modifying security checks to conform to the RBAC policy. To do so, developers may add, move and remove privilege checks. Security-oriented maintenance also includes RBAC policy changes, involving adding, removing and renaming roles and privileges. These changes may be linked to new features added, but sometimes reflect a reorganization of the application's high-level security policy. These activities intentionally change the security protection of different parts of a system.

Vulnerability correction may occur proactively or reactively. Proactive vulnerability fixing occurs when developers and security experts purposefully examine the application to find vulnerabilities and then fix them. This activity should ideally occur before releases. Reactive vulnerability fixing happens in response to an external (i.e. third-party) discovery of a vulnerability. This often occurs after release and often warrants an emergency fix.

We show a vulnerability patch for WordPress in Figure 7.1. This patch is represented in the Unified Diff format, which is the default differencing output format in `git`. In this format, a line added is shown with '`+`' in the left margin. This patch fixes vulnerability CVE-2008-0664 by adding one security check and error handling. That correction, in lines 3 to 5, ensures that the user has the `edit_page` privilege in order to edit a page.

```
1      //...
2      if (!empty($content_struct["post_type"]) && ($content_struct["
          post_type"] == "page")){
3  +      if( !current_user_can( 'edit_page', $post_ID ) ) {
4  +        return(new IXR_Error(401, __("Sorry, you do not have the right
        to edit this page.")));
5  +      }
6  +
7        $post_type = "page";
8      }
9      //...
```

Figure 7.1 Reformatted Vulnerability Correction by Adding Privilege Checks. '+' Represents Added Code (CVE-2008-0664).

Another form of security maintenance is modifying the set of statements protected by a privilege. For instance, a piece of code checks for the right privilege, but this check is improperly located. We show an example in Figure 7.2, which is an excerpt of the fix for vulnerability CVE-2008-0664. We also represent the patch in Unified Diff format, with lines added annotated with '`+`' in the left margin, and lines removed similarly annotated with '`-`'.

Maintainers sometimes abandon, rename, split and merge privileges. An abandoned privilege is a privilege that still exists for backwards compatibility, but which does nothing[1]. A privilege rename is an operation whereby the name of a privilege is changed to another name, but the parts of the code it protects is identical[2]. A privilege split is an operation whereby the code protected by a privilege is divided in two or more sets, which are protected by two or more privileges [3]. A privilege merge is the reverse of a split. The code protected by two or

[1]Examples of abandoned privileges are `edit_files` in WordPress [187] and `emailconfirmed` in MediaWiki [185].

[2]An example of privilege rename in MediaWiki is the renaming of `hiderevision` to `suppressrevision` [185].

[3]An example of privilege split in Mediawiki is the split of `editusercssjs` into `editusercss` and `edituserjs` in [185].

```
+ $publish = ! ( isset( $entry->draft ) && 'yes' == trim( $entry->
    draft ) );
+ $cap = ($publish) ? 'publish_posts' : 'edit_posts';
+ if ( !current_user_can($cap) )
+     $this->auth_required(__('Sorry, you do not have the right to edit
    /publish new posts.'));
  $catnames = array();
  if ( !empty( $entry->categories ) ) {
  foreach ( $entry->categories as $cat ) {
//...
  array_push($post_category, $cat->term_id);
  }

- $publish = ! ( isset( $entry->draft ) && 'yes' == trim( $entry->
    draft ) );
- $cap = ($publish) ? 'publish_posts' : 'edit_posts';
- if ( !current_user_can($cap) )
-     $this->auth_required(__('Sorry, you do not have the right to edit
    /publish new posts.'));
```

Figure 7.2 Excerpt of Vulnerability Correction by Moving Privilege Check. '+' and '-' Represents Added and Deleted Code, Respectively (CVE-2012-4421)

more privileges become protected by a single privilege. This latter privilege may be one of the merged privileges or a new one[4].

Because RBAC security issues are hard to reason about, developers should ideally verify and validate that each change is free from security regressions throughout the project's evolution (e.g. after each commit). This would enable rapid correction, as well as avoiding complicated regressions due to the combined effect of many changes.

A frequent re-verification of the whole application would be very effort-intensive. Even if individual changes alter a small part of the application, "software modified during maintenance should be subjected to the same review as newly developed software." [97]. However, it is possible to identify parts of the program which do not require a full review based on security property analyses (e.g. RBAC test selection [76, 81]). Such analyses should be conservative – meaning that they may incorrectly identify security issues, but they are guaranteed to find all security issues they were designed for.

As developers more and more opt for frequent and quick release cycles, the overhead of regular reviews may become a major hurdle.

The potential overhead of regular security reviews warrants additional tool support for security properties. Thus, we propose an automated static analysis over privilege protection.

[4]An example of privilege merge in WordPress is the merge of `edit_files` into `import` [30].

This analysis automatically partitions changed lines of code as protection-impacting or non-protection-impacting. This is intended to soundly reduce the amount of changes that developers must assess before a release. Our approach is motivated by the successes of static analysis tools in lowering effort for securing software (e.g. MOPS [39], PQL [124], Bek [72]). For instance, a study of three industrial C++ projects estimated cost reduction of 17% for reported security bugs using the Coverity Prevent static analysis tool [19].

Our analysis automatically detects *Definite Protection Differences* (DPD) and *Protection-Impacting Changes* (PIC). DPDs are changes to CFG vertices' definite privilege protection. PICs are the set of those code changes that are responsible for DPDs and that contain some root causes of definite protection differences.

Our main contributions are: (i) a method to automatically identify *protection-impacting changes* during software evolution; (ii) a longitudinal survey of the prevalence of protection-impacting changes over 210 release pairs of WordPress, and 192 release pairs of Mediawiki, two popular PHP web-based content management systems; (iii) a longitudinal survey of DPD root causes whose code changes belong to paths in WordPress and MediaWiki.

We show that *protection-impacting changes* can be identified and are a small proportion of the total number of changes in a release. Our method reduces the pool of candidate causes of security changes that need to be investigated. We show that PICs conservatively represented all root causes of protection differences for 87% to 93% of release pairs. Thus, we think that our approach could help developers eliminate root causes of negative security changes effectively.

This article is organized as follows. In Section 7.2, we cover the background necessary to understand our contributions. Then, in Section 7.3, we define protection-impacting changes. Afterwards, we describe our experiments in Section 7.4. Then, we share and discuss our results in Section 7.5 and 7.6, respectively. We describe related work in Section 7.7 and consider threats to validity in Section 7.8. We bring concluding remarks in Section 7.9. Additionally, we show an example in Appendix A and provide additional methodological details in Appendix B. Finally, we describe our algorithms in appendixes C to F.

7.2 Background

7.2.1 Access Control Methods

A commonly used access control approach is role-based access control (RBAC), with developers and/or administrators assigning *privileges* to one or more *roles* as necessary. Please note that some RBAC implementations use the terms *capability* and *group* instead of privilege and role.

RBAC systems typically rely on an architecture that decouples policy enforcement and decision. The Policy Decision Point (PDP) is responsible to interpret the policy and either grant or deny requests. The Policy Enforcement Point (PEP) communicates with the PDP and enforces the policy decision regarding a privileged action. Developers often put PEPs across their code based in the form of privilege checks. The location of these checks and the privilege they verify depends on the access control policy.

The NBAC (authenticatioN-Based Access Control) family of access control includes RBAC, Attribute-Based Access Control (ABAC) and identity-based access control (IBAC). A different paradigm is authoriZation-Based Access Control (ZBAC) [91]. In RBAC, access control decisions are based on the user's group memberships. In ABAC, access control decisions are based on whether the user's granted attribute set contains a specific attribute and value combination. In IBAC, access control decisions are based on the user's identity and an access control matrix. In ZBAC, access control decisions depend on the validity of authorizations granted to the user.

7.2.2 DPD Root Cause Analysis

A root cause is "the fundamental reason for the occurrence of a problem" [44]. When the link between the root cause and the problem is not direct, one may use Root Cause Analysis (RCA) to identify the root cause. This identification often relies on finding the causal chain between the problem and its root cause.

Properly addressing the root cause of an issue prevents it from re-occurring.

The ultimate root cause behind definite protection differences may be unknowable, as it likely originates from the developers' minds. As such, for software systems, one may perform RCA over the software artifacts, especially the source code. For instance, RCA can identify the cause of failures and defects [162] and automatically patch bugs [120]. In the same vein, to find the root causes of definite protection differences (DPD root causes), we restrict our analysis to code changes between versions.

In programs where the RBAC policy is implemented as privilege checks throughout the code, definite protection differences are caused by code changes between versions Ver_a and Ver_b. In the worst case, when all changes are related to RBAC, the set of DPD root causes is equal the set of code changes. However, the root causes are often a subset of the code changes in practice.

Protection-impacting changes contain a subset of root causes of protection differences. If the experimental reversal of PICs made the DPD disappear, we assumed that PICs contained all root causes of DPD.

Please note that it is possible to make DPD root causes disappear by performing different source code modifications than those that actually occurred in the repository.

7.2.3 Control Flow Graphs

The Control Flow Graph (CFG) is a directed graph represented as $CFG = (V, E)$, where V is the set of vertices, and E is the set of edges. Its initial vertex, which connects to the entry points, is v_0. Paths between v_0 and $v \in V$ are represented by $p\,(v_0, \ldots, v)$. Our CFG model is an extension over the typical interprocedural CFG, whereby we add *grant* edges.

Nonetheless, we elaborate on the vertex and edge types to facilitate reader comprehension. Our CFG vertices may be of the following types. *Entry* and *exit* vertices respectively represent the start and end of a function. *Call begin* and *call end* vertices represent the start and the end of a call site. The start and end vertices respectively represent an entry point and the end of the program's execution. Finally, a default type of vertex exists for other cases.

CFG edges can be of various types. Normal edges are for intraprocedural control flow with no impact on the property satisfaction. Call edges connect call begin and function entry vertices. Return edges connect exit and call end vertices. Grant edges are intraprocedural edges which affect the property satisfaction. They can be either positive or negative. Finally, we mention the *cb* function, which connects the *call end* with its corresponding *call begin* vertex.

7.2.4 Definite Privilege Protection

Our research objective relates to privilege protection, especially definite privilege protection.

Definite privilege protection refers to privileges verified on all interprocedural paths prior to the execution of a statement. A weaker form of protection is *possible* privilege protection. Possible privilege protection refers to privileges verified on some, but not all, paths prior to the execution of the statement. Since our approach focuses on definite privilege protection, we will only refer to definite privilege protection in the remainder of this article.

In the context of CFGs, the predicate $\mathit{DefProt}(v, \mathit{priv})$ is true if and only if all paths between v_0 and v are protected by priv (Equation 7.1). Consequently, $\mathit{DefProt}$ is always false whenever an unprotected path exists between v_0 and v. The predicate $Prot(v,p,priv)$ represents the protectedness of v for privilege *priv* on a path p from the initial node v_0.

$$\begin{aligned} DefProt(v,priv) = \forall p\,(v_0,\ldots,v) \;\mid\; Prot(v,p,priv) \equiv \\ \nexists p'\,(v_0,\ldots,v) \mid \neg Prot(v,p',priv) \end{aligned} \tag{7.1}$$

7.2.5 Pattern Traversal Flow Analysis

We determine definite privilege protection using Pattern-Traversal Flow Analysis (PTFA) [57, 61, 111], a domain-specific model-checking approach for Boolean properties.

PTFA verifies the property satisfaction that a specific predicate is true on all paths reaching a statement s. In our case, we verify predicates about the granting of privileges. These predicates are determined by automatically detecting code patterns of privilege verification checks for some privilege $priv \in Privileges$.

A PTFA model checking automaton is a tuple $\mathcal{A} = (Q_\mathcal{A}, T_\mathcal{A}, q_0, V_\mathcal{A}, G_\mathcal{A}, A_\mathcal{A})$. $Q_\mathcal{A}$ is a finite set of states. $T_\mathcal{A} \subseteq Q_\mathcal{A} \times Q_\mathcal{A}$ is the set of transitions. $q_0 \in Q_\mathcal{A}$ is the initial state. $V_\mathcal{A}$ is the finite set of variables used in *guards* and *assignments*. The guards $G_\mathcal{A}$ annotate the transitions $T_\mathcal{A}$ with logical predicates over $V_\mathcal{A}$, which must all be satisfied for the transition to occur. The assignments $A_\mathcal{A}$ also annotate the transitions $T_\mathcal{A}$ with modifications to the variables $V_\mathcal{A}$.

The model states associated to a node w in the CFG represent the existence of a path from the entry points to node w, in which the checked property is satisfied. Specifically, states in $Q_\mathcal{A}$ are represented as $q_{i,j,k}$, where i is the corresponding CFG node identifier and j and k are property satisfaction flags set to either 0 or 1. j and k holds the calling context's and the intra-procedural property satisfaction, respectively. The value 0 for j or k applies whenever property satisfaction is unknown and whenever the predicate is not satisfied.

In the context of privilege protection, the property to satisfy is that privilege p has been verified prior to the execution of CFG vertex v. Therefore, j and k respectively refer to the calling context's and the intraprocedural protectedness. A value of zero for j or k either means that the privilege was not verified yet, or that the privilege was verified and found to be lacking. An example of the latter case is with an explicit negative check (e.g. `if (!current_user_can('p'))`).

PTFA models are automatically and statically created. They are generated by rewriting the application's CFG [111]. The CFG's vertex and edge types are carried over to PTFA states and transitions. In other words, the type of state $q_{i,j,k} \in Q_{\mathcal{A}}$ is the same type as vertex $i \in V$, and the type of transition $(q_{v,b,c}\ ,\ q_{w,j,k}) \in T_{\mathcal{A}}$ is the same as for the edge $(v, w) \in E$.

After generating the PTFA model from the application's CFG, we simplify the former into the *reachable PTFA model.* The reachable PTFA model $\mathit{Reach}\mathcal{A} = (Q, q_0, T)$ only retains reachable states and transitions from the original PTFA model (Equation 7.2). The guards and assignments are implicit in the transitions and are otherwise removed from the model.

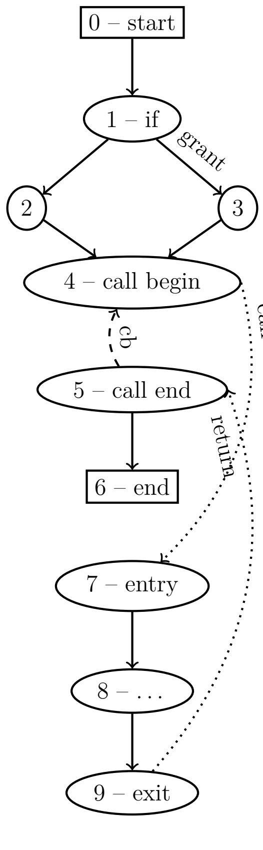
$$\begin{aligned} Q &\doteq \{q_{i,j,k} \in Q_{\mathcal{A}} \ \mid \ \mathit{reachable}(q_{i,j,k})\} \\ T &\doteq \{(q_{a,b,c}\ ,\ q_{i,j,k}) \in T_{\mathcal{A}} \mid q_{a,b,c} \in Q \ \wedge \ q_{i,j,k} \in Q\} \end{aligned} \tag{7.2}$$

We illustrate the conversion of a CFG into a reachable PTFA model in Figure 7.3. The i index of PTFA states $q_{i,j,k}$ refers to the CFG vertex number, which we included in Figure A.2. In this example, we show the effect of grant and interprocedural edges in the CFG. A reachable PTFA model indicates definite protection by the states belonging to the model. For a CFG vertex v, a protected state $q_{v,j,1}$ belongs to the model if and only if there exists a protected path between v_0 and v. And an unprotected state $q_{v,j,0}$ exists in the model if and only if there exists an unprotected path between v_0 and v. Therefore, definite protection for v is determined by which states corresponding to v are present in the model (Equation 7.3).

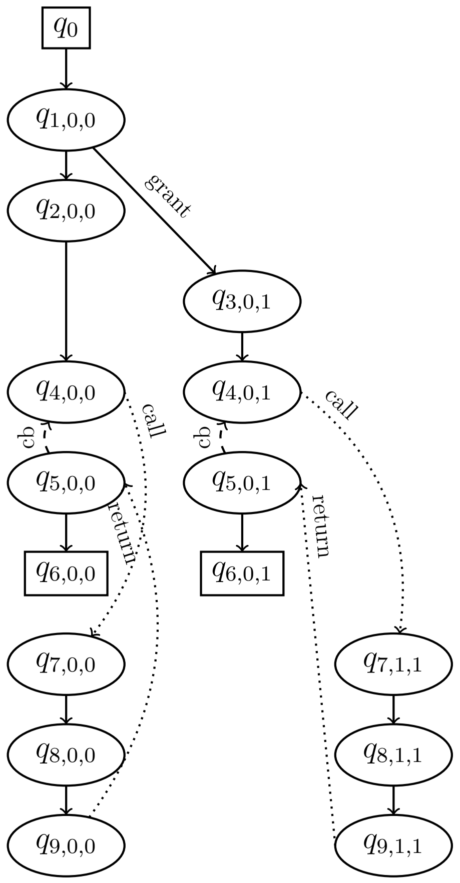
$$\mathit{DefProt}(v, \mathit{priv}) = (q_{v,0,1} \in Q \ \vee \ q_{v,1,1} \in Q) \ \wedge \ q_{v,0,0} \notin Q \ \wedge \ q_{v,1,0} \notin Q \tag{7.3}$$

PTFA models are identical to those that would be constructed by a fully context sensitive analysis. This is because PTFA models rely on *predicate context sensitivity.* Predicate context sensitivity is a merging of contexts that are equivalent, as far as the satisfaction of the security property is concerned. This form of context sensitivity embeds the caller's predicate satisfaction in the PTFA models, using the j flag. This flag is taken in consideration in the graph rewriting rules.

In the context of privilege protection, predicate context sensitivity means that the protectedness of the calling context is propagated to the callee. For instance, calling from a protected context results in the callee being also protected. We show an example in Figure 7.3. The CFG vertices 7 to 9 are called in both protected and unprotected contexts.



(a) Example CFG



(b) Corresponding Reachable PTFA Model

Figure 7.3 Example CFG and Corresponding Reachable PTFA Model. Local Protection is Propagated to Called Code.

7.2.6 Definite Protection Differences

A Definite Protection Difference (DPD) occurs when the definite protection *DefProt* for a statement s (which is common to versions Ver_a and Ver_b) changes between versions [101].

We consider that statement s is *loss-affected* for privilege *priv*, if it was definitely protected for *priv* in Ver_a and is not definitely protected for *priv* in Ver_b (c.f. *lossAffected* in Equation 7.4). A similar definition applies for *gain-affected* statements, whenever statement s was granted a *priv* protection in Ver_b that was not there in Ver_a (c.f. *gainAffected* in Equation 7.5). Please note that statement s may be loss-affected with regards to privilege p_1, but gain-affected with regards to privilege p_2.

We illustrate DPDs in Figure 7.4 using the Unified Diff format. A '`+`' or '`-`' on the left margin respectively represents an added or deleted line. We underline gain-affected and loss-affected statements. Figure 7.4a shows a local loss-affected statement caused by removing a protection check. Figure 7.4b shows an interprocedural loss-affected statement caused by adding an unprotected path. Figure 7.4c shows a local gain-affected statement caused by adding a protection check. Finally, Figure 7.4d shows an interprocedural gain-protected statement caused by adding a protection in the calling context.

```
- if (!current_user_can('p')) die();
  print "loss-affected statement";
```

(a) Local Loss Impact

```
  function foo(){
    print "loss-affected statement";
  }
+ foo();
  if (!current_user_can('p')) die();
  foo();
```

(b) Interprocedural Loss Impact

```
+ if (!current_user_can('p')) die();
  print "gain-affected statement";
```

(c) Local Gain Impact

```
  function foo(){
    print "gain-affected statement";
  }
+ if (!current_user_can('p')) die();
  foo();
```

(d) Interprocedural Gain Impact

Figure 7.4 Examples of gain-affected and loss-affected CFG Vertices. Code Changes Indicated in the Left Margin. Added lines are shown with '+'. Deleted lines are shown with '-'.

We define the *gainAffected* and *lossAffected* predicates in Equations 7.4 and 7.5. These predicates are true if a CFG vertex v_b in Ver_b is respectively gain and loss-affected. The function $vertexMap$ is an invertible injective function that associates comparable CFG vertices from Ver_a to Ver_b. Its inverse is $bMap$. We use the symbol *range* to refer to range of a function.

$$lossAffected(v_b, priv) \doteq$$
$$DefProt(bMap(v_b), priv) \ \wedge \neg DefProt(v_b, priv) \ \wedge v_b \in range(vertexMap) \quad (7.4)$$

$$gainAffected(v_b, priv) \doteq$$
$$\neg DefProt(bMap(v_b), priv) \ \wedge DefProt(v_b, priv) \ \wedge v_b \in range(vertexMap) \quad (7.5)$$

7.3 Method

We introduce *protection-impacting changes* to determine which code changes on execution paths potentially impact definite privilege protection. Our analysis rely on graph reachability in reachable PTFA models.

Protection-impacting changes contains DPD root causes whose code changes belong to paths. This is category of DPD root causes that occur when the code changes belongs to paths. Consequently, whenever definite protection differences are entirely caused by DPD root causes whose code changes belong to paths, developers only need to focus their attention on protection-impacting changes.

Gain-impacting changes are protection-impacting changes that caused one or more statements to be gain-affected, while *loss-impacting changes* are conversely defined (c.f. Section 7.2.6). These labels are not mutually exclusive. Some code changes may belong to both kinds of definite protection differences. For instance, consider Figure 7.5, where a security check is inverted. This change will cause the first `print` statement to become loss-impacted, and the second `print` statement to become gain-impacted. We show the processing steps to

```
- if ( current_user_can('p1'))
+ if (!current_user_can('p1'))
    print 'Loss-affected'
  else
    print 'Gain-affected'
```

Figure 7.5 Code Change That is Gain-Impacting and Loss-Impacting

determine protection-impacting changes in Figure 7.6. The reachable PTFA change detector (c.f. Section 7.3.1) determines which edges have been deleted and added between two reachable PTFA models. The reachable edges finder (c.f. Section 7.3.2 determines which edges in reachable PTFA models connect to definite protection differences. Finally, the PIC finder (c.f. Section 7.3.3) combines the reachable edges, and the added and deleted edges to determine the protection-impacting changes.

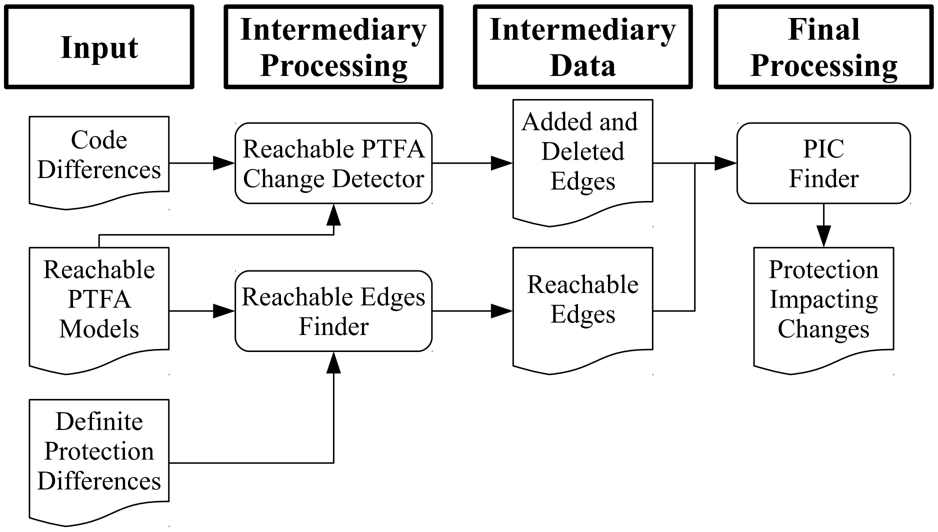


Figure 7.6 Processing Steps to Determine Protection-Impacting Changes

7.3.1 Reachable PTFA Change Detector

Our approach relies on detecting added and deleted edges in reachable PTFA graphs. Definite protection is a predicate over program paths, and paths are composed of edges. Therefore, we must determine edge-level differences to determine protection-impacting changes. The code change detector uses the reachable PTFA models and code difference information to do so.

When comparing two versions, Ver_a and Ver_b typically have some parts in common, and some different parts. Thus, for the parts in common, we can build a one-to-one correspondence between a subset of their reachable PTFA models ($PTFA_a$ and $PTFA_b$, respectively). From this subset, we add states and transitions to form $PTFA_b$. Given $vertexMap$, the construction of $PTFA_b$ from $PTFA_a$ is possible using Equation 7.6. Q_a and T_a respectively are the set of states and the set of transitions in the reachable PTFA model for version Ver_a. Q_b and T_b are similarly defined for the reachable PTFA model for Ver_b. Please note that *deletedStates* and *addedStates* are respectively the set of deleted and added PTFA states, and that *deletedEdges* and *addedEdges* are respectively the deleted and added edges.

$$\begin{aligned}
&PTFA_a \doteq (Q_a, q_0, T_a)\\
&PTFA_b \doteq (Q_b, q_0, T_b)\\
&Q_b = addedStates \;\cup\; \left\{ q_{i',j,k} \;\middle|\; q_{i,j,k} \in\; Q_a \setminus\; deletedStates \;\wedge i' = vertexMap(i) \right\}\\
&T_b = addedEdges \;\cup\; \left\{ \left(\, q_{a',b,c} \;,\; q_{d',e,f} \,\right) \;\middle|\; \begin{array}{c} (q_{a,b,c} \;,\; q_{d,e,f}) \in T_a \setminus deletedEdges \\ \wedge\; a' = vertexMap(a) \;\wedge \\ d' = vertexMap(d) \end{array} \right\}
\end{aligned} \tag{7.6}$$

We determine *addedStates*, *deletedStates*, *addedEdges* and *deletedEdges* as follows. The set *deletedStates* contains all states $(q_{i,j,k}) \in Q_a$ with no corresponding state in Ver_b (Equation 7.7). This occurs when code is deleted, changed or when the code remains unchanged while its privilege protection of the state is modified. Similarly, *addedStates* contains all states $q_{i,j,k} \in Q_b$ with no corresponding state in Ver_a (Equation 7.8). This occurs when code is added, changed, or for unchanged code whose privilege protection is modified. The symbols *dom* and *range* correspond to the function's domain and range, respectively.

deletedEdges is the set of deleted edges in the PTFA model for Ver_a (Equation 7.9). It corresponds to the set of all edges which either connect one or more deleted states or for which no corresponding edge is present in the PTFA model for Ver_b. Finally, *addedEdges* is the set of added edges in to PTFA model for Ver_b (Equation 7.10). It is defined in a similar manner of *deletedEdges*.

$$deletedStates \doteq \left\{\; (q_{i,j,k}) \in Q_a \;\middle|\; i \notin dom(vertexMap) \;\vee q_{vertexMap(i),j,k} \notin Q_b \right\} \tag{7.7}$$

$$addedStates \doteq \left\{\; (q_{i,j,k}) \in Q_b \;\middle|\; i \notin range(vertexMap) \;\vee q_{bMap(i),j,k} \notin Q_a \right\} \tag{7.8}$$

$$\begin{aligned}
&deletedEdges \doteq\\
&\left\{ \left(q_{i_1,j_1,k_1} \;,\; q_{i_2,j_2,k_2} \right) \in T_a \;\middle|\; \begin{array}{l} q_{i_1,j_1,k_1} \in deletedStates \;\vee q_{i_2,j_2,k_2} \in deletedStates \;\vee \\ \left(\, q_{vertexMap(i_1),j_1,k_1} \;,\; q_{vertexMap(i_2),j_2,k_2} \,\right) \notin T_b \end{array} \right\}
\end{aligned} \tag{7.9}$$

$$\begin{aligned}
&addedEdges \doteq\\
&\left\{ \left(\, q_{i_1,j_1,k_1} \;,\; q_{i_2,j_2,k_2} \,\right) \in T_b \;\middle|\; \begin{array}{l} q_{i_1,j_1,k_1} \in addedStates \;\vee q_{i_2,j_2,k_2} \in addedStates \;\vee \\ \left(\, q_{bMap(i_1),j_1,k_1} \;,\; q_{bMap(i_2),j_2,k_2} \,\right) \notin T_a \end{array} \right\}
\end{aligned} \tag{7.10}$$

7.3.2 Reachable Edges Finder

The reachable edges finder is responsible to find edges that belong to *appropriately* protected paths to a definite protection difference. We explain and show how to determine edges that belong to paths to a definite protection difference in Section 7.3.2. We then define appropriately protected paths in Section 7.3.2.

Paths to Definite Protection Differences

The first criterion, that the path connects to a definite protection difference, is because protection-impacting changes are related to a) changes in paths leading to security sensitive operations and b) their traversal of security granting patterns, as computed by PTFA.

Let us consider a definite protection difference. Vertex v_a belongs to version Ver_a. It has a corresponding vertex in Ver_b, v_b (i.e. $v_b = vertexMap(v_a)$).

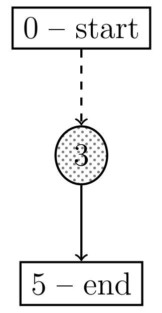
Protection-impacting changes belong to paths having changed privilege protection for v_a and v_b. They also precede the execution of v_a and v_b on these paths. In other words, there exists edges e_{PICa} in Ver_a or e_{PICb} in Ver_b, such that Equation 7.11 is satisfied.

$$\exists p \left(\quad (v_0, v_1),\ \dots\ , e_{PICa},\ \dots\ , (v_n, v_a) \quad \right) \vee \exists p \left(\quad (v_0, v_1),\ \dots\ , e_{PICb},\ \dots\ , (v_n, v_b) \quad \right) \tag{7.11}$$

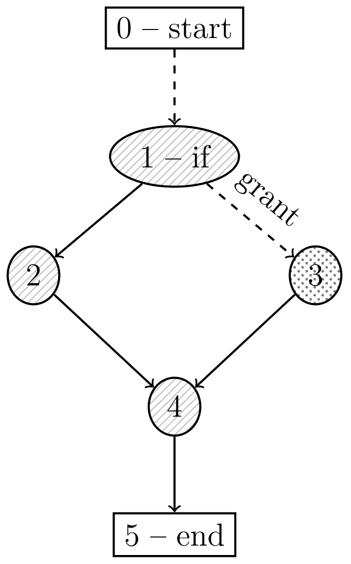
The paths containing e_{PICb} may be entirely new paths in Ver_b. The paths containing e_{PICa} may be entirely deleted paths in Ver_a. Or they can be paths modified between Ver_a and Ver_b such that their definite protection differs.

To illustrate, please consider the example in Figure 7.7. Figures 7.7a and 7.7b respectively show the CFGs for Ver_a and Ver_b. Please note that $vertexMap$, in this case, is the identity function. Between Ver_a and Ver_b, vertex v_3 is gain-affected. The only edge in Ver_a that belong to a path from v_0 to v_3 in Ver_a is (v_0, v_3). The edges in Ver_b that belong to a path from v_0 to v_3 are $\{\ (v_0, v_1), (v_1, v_3)\ \}$. The other edges connecting code changes in Ver_b (i.e. $\{\ (v_1, v_2), (v_2, v_4), (v_3, v_4), (v_4, v_5)\ \}$) cannot affect the definite privilege protection of v_3 in any way.

We show an example of paths to a definite protection difference in Figure 7.8. We show the CFG Ver_a (Figure 7.8a) and Ver_b (Figure 7.8b). In this case, $vertexMap$ is the identity function. One vertex, v_3, is loss-affected between The only edges in Ver_a that belong to a path from v_0 to v_3 are $\{\ (v_0, v_1), (v_1, v_2), (v_2, v_3)\ \}$. The only edges in Ver_b that belong to a path from v_0 to v_3 are $\{\ (v_0, v_1), (v_1, v_2), (v_2, v_3), (v_1, v_3)\ \}$. The other edges cannot affect the definite privilege protection of v_3 in any way.



(a) CFG of Ver_a



(b) CFG of Ver_b

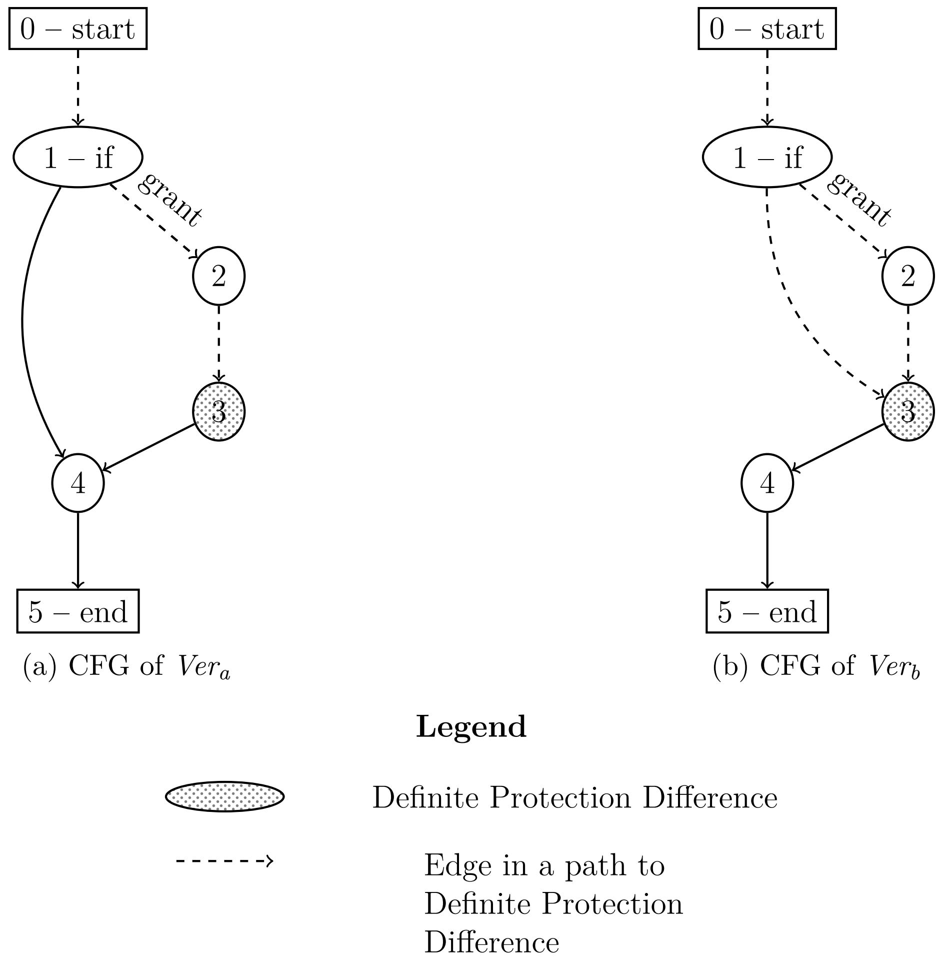
Legend

▨ Added vertices

▒ Definite Protection Difference

- - - -> Edge in a path to Definite Protection Difference

Figure 7.7 Example Paths to definite protection differences


0 – start
1 – if
grant
2
3
4
5 – end
0 – start
1 – if
grant
2
3
4
5 – end
Legend
Definite Protection Difference
Edge in a path to Definite Protection Difference

(a) CFG of Ver_a

(b) CFG of Ver_b

Figure 7.8 Example Paths to definite protection differences

Appropriately Protected Path

The reachable edges finder does not look for all edges in all paths to definite protection differences. Protection-impacting changes belong to *appropriately* protected paths to a definite protection difference. By appropriately protected, we mean that protection-impacting changes belong to either unprotected or protected paths, depending on whether the code is gain-affected or loss-affected.

The consideration of appropriately protected paths is a refinement over Section 7.3.2.

Given CFG vertices v_a in Ver_a and v_b in Ver_b, such that $v_b = vertexMap(v_a)$, the following properties hold when they are loss-affected for privilege *priv*, based on the definition of definite protection.

First, v_a is definitely protected for *priv*, which means that all paths to v_a are protected by *priv*. Second, v_b is not definitely protected for *priv*. By inversion of the universally quantified property, there exists at least one unprotected path for *priv* leading to v_b. Alternatively, v_b may be unreachable (Equation 7.12).

As we saw before, these universally and existentially quantified predicates correspond to the existence of states in reachable PTFA models. In Equation 7.12, Q_a and Q_b respectively refer to the set of states in the reachable PTFA models for Ver_a and Ver_b. For Equation 7.12 to hold true, we only need to consider the protected paths to $q_{a,j,1}$ and (if any) the unprotected paths to $q_{b,j,0}$. Thus, protection-impacting changes must belong to these paths.

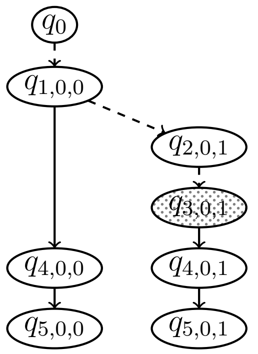
$$\begin{gathered}
\forall p\,(v_0,\ldots,v_a) \;\mid\; Prot(v_a,p,priv) \,\wedge \\
(\exists p'\,(v_0,\ldots,v_b)\,, \neg Prot(v_b,p',priv) \;\vee\; \nexists p''\,(v_0,\ldots,v_b)) \\
\equiv \\
(q_{a,0,1} \in Q_a \;\vee\; q_{a,1,1} \in Q_a) \;\wedge\; q_{a,0,0} \notin Q_a \,\wedge \\
q_{a,1,0} \notin Q_a \;\wedge \Big(\; q_{b,0,0} \in Q_b \vee q_{b,1,0} \in Q_b \;\vee\; \nexists j,k \in \{0,1\} \mid q_{b,j,k} \in Q_b \;\Big)
\end{gathered} \tag{7.12}$$

The situation is similar when dealing with gain-affected CFG vertices. Vertex v_a is not definitely protected and v_b is definitely protected for *priv*. Thus, the relation in Equation 7.12 only needs to be adapted by exchanging the versions (Equation 7.13). For Equation 7.13 to hold true, we only need to consider the protected paths to $q_{b,j,1}$ and (if any) the unprotected paths to $q_{a,j,0}$. Thus, protection-impacting changes must belong to these paths.

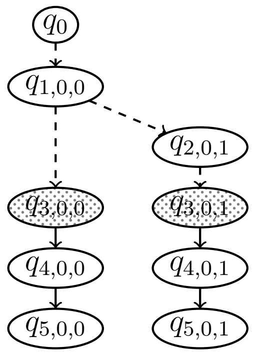
$$
\begin{gathered}
\forall p\,(v_0, \ldots, v_b) \;\mid\; Prot(v_b, p, priv) \,\wedge \\
(\exists p'\,(v_0, \ldots, v_a) \;\mid\; \neg Prot(v_a, priv) \;\vee\; \nexists p''\,(v_0, \ldots, v_a)) \\
\equiv \\
(q_{b,0,1} \in Q_b \;\vee\; q_{b,1,1} \in Q_b) \;\wedge\; q_{b,0,0} \notin Q_b \,\wedge \\
q_{b,1,0} \notin Q_b \;\wedge\; \Big(\; q_{a,0,0} \in Q_a \;\vee\; q_{a,1,0} \in Q_a \;\vee\; \nexists j, k \in \{0,1\} \;\mid\; q_{a,j,k} \in Q_a \;\Big)
\end{gathered}
\tag{7.13}
$$

Taking appropriately protected paths to definite protection differences in consideration is a refinement over considering all paths to definite protection differences. We illustrate this refinement with the same example as in Figure 7.8. We convert the CFGs to reachable PTFA models and highlight the appropriately protected paths (Figure 7.9).

In *Ver*$_a$, there is only one path (i.e. $(q_0,\; q_{1,0,0},\; q_{2,0,1},\; q_{3,0,1})$) leading to a protected state for CFG vertex v_3 (Figure 7.9a). In *Ver*$_b$, there are two paths leading to states corresponding to CFG vertex v_3 (Figure 7.9b). However, there is only one path leading to an unprotected state (i.e. $(q_0,\; q_{1,0,0},\; q_{3,0,0})$), which contributes to protection-impacting changes.



(a) Reachable PTFA Model for *Ver*$_a$



(b) Reachable PTFA Model for *Ver*$_b$

Legend

▨	Definite Protection Difference
- - - - - ->	Edge in a path to Definite Protection Difference

Figure 7.9 Reachable PTFA Models Corresponding to Figure 7.8

7.3.3 Formal Definition of Protection-Impacting Changes

Protection-impacting changes are code changes which belong to an appropriately protected path leading to a definite protection difference. Thus, the definition of protection-impacting changes varies between gain-affected and loss-affected code.

For loss-affected code, the protection-impacting changes PIC_{loss} are the deleted edges belonging to positively-protected paths to v_a and the added edges belonging to negatively-protected paths to v_b (Equation 7.14). For gain-affected code, the protection-impacting changes PIC_{gain} are similarly defined, with the protectedness reversed (Equation 7.15). Please note that $ReachachableEdges(q_{i,j,k})$ is a function returning all edges between the initial state q_0 and $q_{i,j,k}$ in the reachable PTFA model. To simplify the definitions, we introduce the $partialPIC$ function, which combines changed edges (i.e. $addedEdges$ or $deletedEdges$) with reachable edges for either Ver_a or Ver_b.

$$\begin{aligned} partialPIC(changes, i, k) \doteq\ & changes \cap \\ & (ReachableEdges(q_{i,0,k}) \cup ReachableEdges(q_{i,1,k})) \end{aligned}$$

$$PIC_{loss} \doteq \Big(\ partialPIC(deletedEdges, v_a, 1)\ ,\ partialPIC(addedEdges, v_b, 0)\ \Big) \quad (7.14)$$

$$PIC_{gain} \doteq \Big(\ partialPIC(deletedEdges, v_a, 0)\ ,\ partialPIC(addedEdges, v_b, 1)\ \Big) \quad (7.15)$$

7.4 Experiment Design

In this study, we answer two research questions, which we detail in Sections 7.4.1 and 7.4.2

7.4.1 RQ1: Reduction of Code Changes to Evaluate

Our first research question is the following:

Research Question
RQ1 *Is the size of protection-impacting changes smaller than the set of code changes?*

Our goal is to quantify the amount and proportion of code changes that are protection-impacting in real systems.

Approach

We conduct a survey [156] of two representative open source projects implemented in PHP, WordPress and MediaWiki. A survey is a descriptive quantitative study of a phenomenon [156]. In this survey, we determine code changes and protection-impacting changes between major releases, as detailed in Subsection 7.4.1.

We chose these two applications because they are large, RBAC-using, widely-used software systems with a long release history.

WordPress is a popular web-based content management system implemented in PHP. Its RBAC implementation and configuration is relatively simple. At the time of writing, WordPress' access control policy is designed with offers six roles and 63 capabilities (i.e. privileges). The specified role hierarchy is strictly linear, meaning that each role has a superset of the capabilities of the previous role in the hierarchy [187]. Our study encompasses all releases of WordPress available as of March 2017 and which used RBAC – 2.0 to 4.7.3. This application's PHP code, in physical lines of code (LOC), ranges from roughly 35 KLOC in release 2.0 to 340 KLOC in release 4.7.3. For the same releases, the combined HTML, JavaScript and CSS code amounted to roughly 13 KLOC and 179 KLOC, respectively.

MediaWiki is another content management system. It is well-known thanks to its flagship user, Wikipedia. Our study encompasses all releases of MediaWiki available as of November 2017 and which used RBAC – 1.5 to 1.29.1 [185]. At the time of writing, MediaWiki's access control policy has 7 groups (i.e. roles), and 75 permissions (i.e. privileges) [185]. This application's PHP code ranges from roughly 149 KLOC in release 1.5 to 149 KLOC in release 1.29.1. For the same releases, the combined HTML, JavaScript and CSS code amounted to roughly 4.5 KLOC and 188 KLOC, respectively. The peak PHP code size is roughly 1.35 MLOC in release 1.22.3.

Our processing pipeline (Figure 7.10) consists of a PHP Front-End which provides CFGs to the PTFA Analyzer. The PTFA Analyzer generates PTFA models and definite protection data, which are used by the protection-impacting changes analyzer and DPD Classifier. The code differencing component computes line differences for the source code, which it stores in intermediary files. The code differences and definite protection summaries are used to compute definite protection difference. The *vertexMap* function is computed internally in these components using the algorithm in Appendix C. Finally, the Protection-Impacting Change Analyzer uses the DPD data, code differences and PTFA models and determines protection-impacting changes. It implements the *ReachableEdges* function using backwards reachability, which we describe in Appendix F.

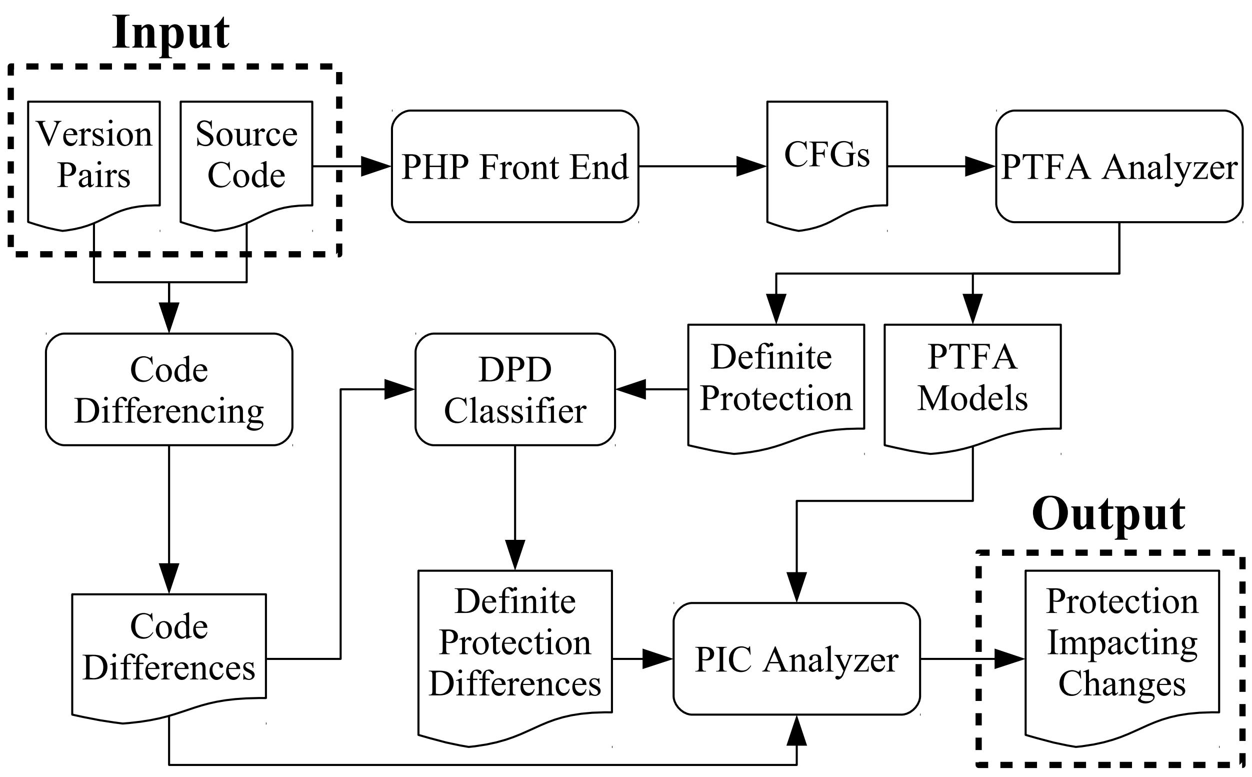


Figure 7.10 Processing Pipeline

Dependent and Independent Variables

In this survey, the independent variables are which programs, which versions we analyze, and the software running the experiment (including the source code differencing tool, the language front-end and the PTFA analyzer). The dependent variables are code differences and protection-impacting changes.

Data Collection and Processing

Our approach relies on comparing versions. We must therefore choose *which* versions we consider, and which ones to compare against each other.

As a preliminary study, we conduct our survey by comparing official final releases (i.e. non-alpha, non-beta, non-release candidate versions). We thus omit releases such as MediaWiki's 1.5.0 Beta 2, and 1.9.0 RC1.

We select these versions since these are the versions that end-users normally deploy. Additionally, we expect that these releases will be better tested and reviewed than intermediary versions. The impact of this decision is twofold. Since there may be a large number of code changes between two releases, the number of protection-impacting changes is likely to be large in absolute terms. However, pre-release security reviews may lower the relative amount of protection-impacting changes. Please note that we analyzed official release archives (for more details, please consult Appendix B). A finer-grained (e.g. commit-level) analysis is possible, which we discuss in Section 7.6.

To calculate protection-impacting changes, we compare versions Ver_a with Ver_b. In our survey, we call the 2-tuple (Ver_a, Ver_b) a *release pair*. It would be tempting to arrange the releases in a line, sorted by their release number, and compare them pairwise. However, WordPress and MediaWiki developers maintain and apply security fixes to multiple releases in parallel. Considering releases sequentially would artificially inflate the number of definite protection differences and protection-impacting changes we observe. For instance, consider vulnerability CVE-2016-6331, which was simultaneously corrected in MediaWiki releases 1.23.15, 1.26.4, and 1.27.1. If one were to compare releases linearly, they would observe a gain-affected code in 1.23.15 which would become loss-affected in 1.24.0, only to become gain-affected again in 1.26.4 and so on until 1.27.1.

We avoid this issue by organizing releases in a tree. To do so, we rely on two properties of each version: its release number and release date. The release numbers in MediaWiki and WordPress are organized using the semantic versioning convention [149]. Each release is also tagged with a release date, which is the day where the the release is made available to the public. For each edge (Ver_a, Ver_b) in the tree, we ensure that the release date of Ver_a precedes or equals the release date of Ver_b. We also ensure that the release number of Ver_a is the highest-numbered release that is lower-numbered than Ver_b. We describe our algorithm in more detail in Appendix D. We show an excerpt of the MediaWiki release tree built using our algorithm in Figure 7.11. Please consider the effect of our algorithm for release 1.9.0. The edge $(1.8.2, 1.9.0)$ is explained by the fact that releases 1.8.3 onwards were released after 1.9.0.

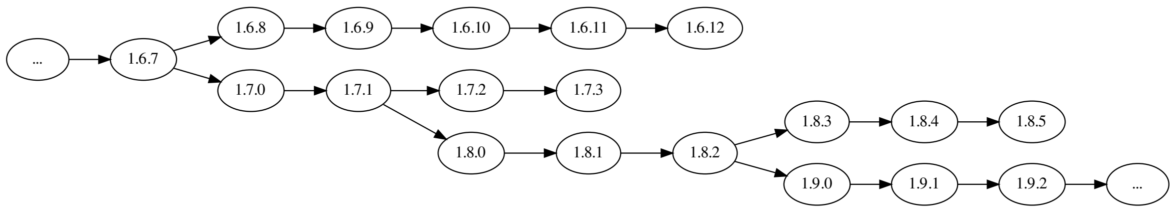


Figure 7.11 Release Tree Excerpt for MediaWiki

We compute *vertexMap* (and its inverse, *bMap*) using an heuristic mapping based on line-level differences (c.f. Appendix C).

Measures

In our survey, we record *security-affected* release pairs, which are release pairs in which at least one definite protection difference. For all security-affected release pairs, we measure protection-impacting changes and code changes. Our measures are at two levels. We measure at the level of PTFA edges and lines of source code.

We measure at line granularity because measures in terms of lines of code are the de-facto standard in the industry. This is evidenced by popular differencing and version control tools (e.g. `diff` and `git`) that report code changes at a line granularity by default.

We have no access to an oracle which classifies definite protection differences either as beneficial, harmful or irrelevant. Therefore, we report protection-impacting change metrics for all definite protection differences. We discuss oracles in Section 7.6.

Release pairs with gain-affected vertices are also called gain-affected and release pairs with loss-affected vertices are also called loss-affected.

Protection-impacting changes (PIC_{loss} and PIC_{gain}) are a 2-tuple of reachable PTFA edge sets. We measure the set size of each component of the tuple. We also process these PIC_{loss} and PIC_{gain} into additional measures likewise expressed as 2-tuples. These measures may be harder to interpret. As such, we also define measures that combine the components of the 2-tuple. To do so, we use the projection function $\pi_i(x)$ to extract the ith component of x. In other words, $\pi_1(x)$ extracts the component related to Ver_a in x and $\pi_2(x)$ does likewise for Ver_b.

First, we combine PIC_{loss} and PIC_{gain} as PIC (Equation 7.16). We do so by performing the union of corresponding components of PIC_{loss} and PIC_{gain}. We also add the set cardinalities of each component of PIC to obtain the combined metric *allPIC* (Equation 7.17).

$$PIC = \Big(\ |\pi_1(PIC_{loss}) \cup \pi_1(PIC_{gain})| \ , \ |\pi_2(PIC_{loss}) \cup \pi_2(PIC_{gain})| \ \Big) \tag{7.16}$$

$$allPIC = \pi_1(PIC) + \pi_2(PIC) \tag{7.17}$$

We also define relative protection-impacting changes measures. The rationale for these measures is that protection-impacting changes should be interpreted in the context of all changed lines of code. The measures $relPIC_{loss}$, $relPIC_{gain}$ and $relPIC$ are 2-tuples containing the ratios between protection-impacting edges and the added and deleted edges (Equation 7.18). We also define a combined relative measure *relAllPIC* similarly.

$$\begin{aligned} relPIC_{loss} &= \left(\frac{|\pi_1(PIC_{loss})|}{|deletedEdges|}, \frac{|\pi_2(PIC_{loss})|}{|addedEdges|}\right) \\ relPIC_{gain} &= \left(\frac{|\pi_1(PIC_{gain})|}{|deletedEdges|}, \frac{|\pi_2(PIC_{gain})|}{|addedEdges|}\right) \\ relPIC &= \left(\frac{\pi_1(PIC)}{|deletedEdges|}, \frac{\pi_2(PIC)}{|addedEdges|}\right) \\ relAllPIC &= \frac{allPIC}{|deletedEdges|+|addedEdges|} \end{aligned} \tag{7.18}$$

We project the PTFA edges to lines of code using heuristics that take line change information in consideration. We do so because some vertices in *addedEdges* and *deletedEdges* may belong to unchanged lines of code, and reporting them as protection-impacting is likely to confuse the users.

Please consider the example in Figure 7.12a. We show the corresponding reachable PTFA graph for Ver_b in Figure 7.12b. We represent added edges in grey, and protection-impacting changes with larger arrows. If we map all states belonging to edges in the set of protection-impacting changes to their line of code, lines $\{1, 2, 3\}$ appear protection-impacting. Since the only code change is the addition of line 2, we only report that line 2 is protection-impacting.

These mapped metrics are $PIC_{loss,lines}$ and $PIC_{gain,lines}$. Similarly, the combined measure $allPIC_{lines}$ is defined in Equation 7.20.

$$PIC_{lines} \doteq \left(|\pi_1(PIC_{loss,lines}) \cup \pi_1(PIC_{gain,lines})| \ , \ |\pi_2(PIC_{loss,lines}) \cup \pi_2(PIC_{gain,lines})| \right) \tag{7.19}$$

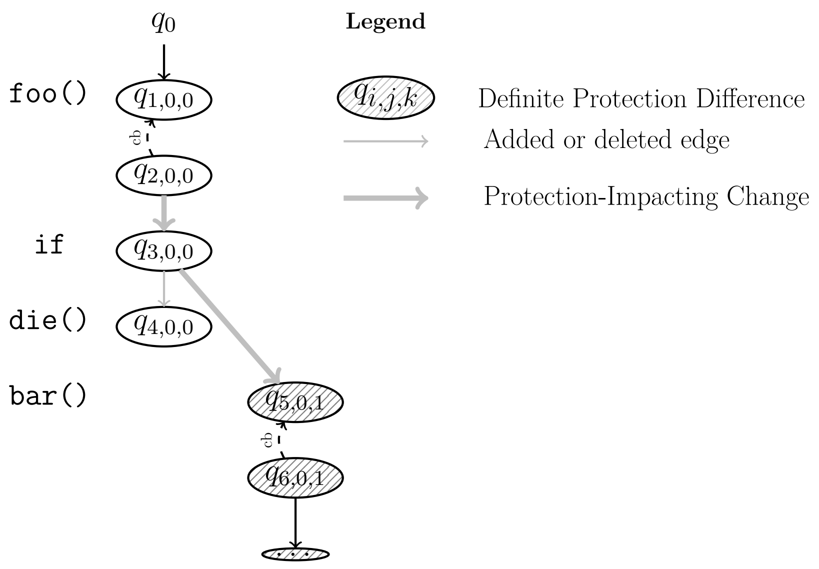
$$allPIC_{lines} \doteq \pi_1(PIC_{lines}) \ + \ \pi_2(PIC_{lines}) \tag{7.20}$$

We also determine the relative size of protection-impacting changes per release pair as the ratio between the total protection-impacting lines and the total modified lines of code (Equation 7.21). In the equations below, *delLines* and *addLines*, which respectively are the number of lines deleted from Ver_a and added to Ver_b.

$$\begin{aligned} relPIC_{loss,lines} &= \left(\frac{|\pi_1(PIC_{loss,lines})|}{delLines}, \frac{|\pi_2(PIC_{loss,lines})|}{addLines}\right) \\ relPIC_{gain,lines} &= \left(\frac{|\pi_1(PIC_{gain,lines})|}{delLines}, \frac{|\pi_2(PIC_{gain,lines})|}{addLines}\right) \\ relPIC_{lines} &= \left(\frac{|\pi_1(PIC_{lines})|}{delLines}, \frac{|\pi_2(PIC_{lines})|}{addLines}\right) \\ relAllPIC_{lines} &= \frac{allPIC_{lines}}{delLines + addLines} \end{aligned} \tag{7.21}$$

```
1     foo();
2  +  if (!current_user_can('p')) die();
3     bar();
4     //...
```

(a) Example Code Change



(b) Reachable PTFA Model for Ver_b

Figure 7.12 Example Illustrating the Need to Use Heuristics When Projecting Protection-Impacting-Changes to Lines of Code. A Naive Projection Would Inaccurately Report Lines 2 and 3 as Protection-Impacting.

Implementation

We implemented the workflow in Figure 7.10 using Java. To compute *delLines* and *addLines*, we used `diff` from GNU diffutils 3.3 and `diffstat` 1.61.

7.4.2 RQ2 : PICs containing all DPD root causes

Our second research question is the following:

Research Question
RQ2 *How often PICs contain all DPD root causes in a release pair?*

The goal of this research question is to determine the relevance of DPD root causes whose code changes belong to paths. In other words, the goal of this research question is to determine how often do protection-impacting changes contain all DPD root causes. To do so, we use two different strategies to revert protection-impacting changes and observe if definite protection differences disappear (Section 7.4.2).

Approach

We perform reversals at two different granularities. The first strategy is *complete reversal* (Section 7.4.2). For a complete reversal, we revert all protection-impacting changes in each security-affected release pairs. Our second strategy is *individual reversal* (Section 7.4.2). For individual reversals, we revert protection-impacting changes related to each definite protection difference in one release pair of each system.

Reverting a set of code changes produces a new version, the *reverted version*. As we covered in Section 7.2.2, the reverted version is free of definite protection differences if all DPD root causes have been reverted.

We measure, for both strategies, the number of successfully reverted versions. Each strategy has a different definition of what a successful reversal is, which we detail in the following Sections.

To do so, we leverage many of the processing steps from **RQ1**. To these, we add the protection-impacting change reverter, which creates a *reverted version* based on protection-impacting changes as well as the source code of versions Ver_a and Ver_b. We show the steps in Figure 7.13. We represent the new component and its output in bold, whereas components from Figure 7.10 are show in normal font. We describe the protection-impacting change reverter in Section 7.4.2.

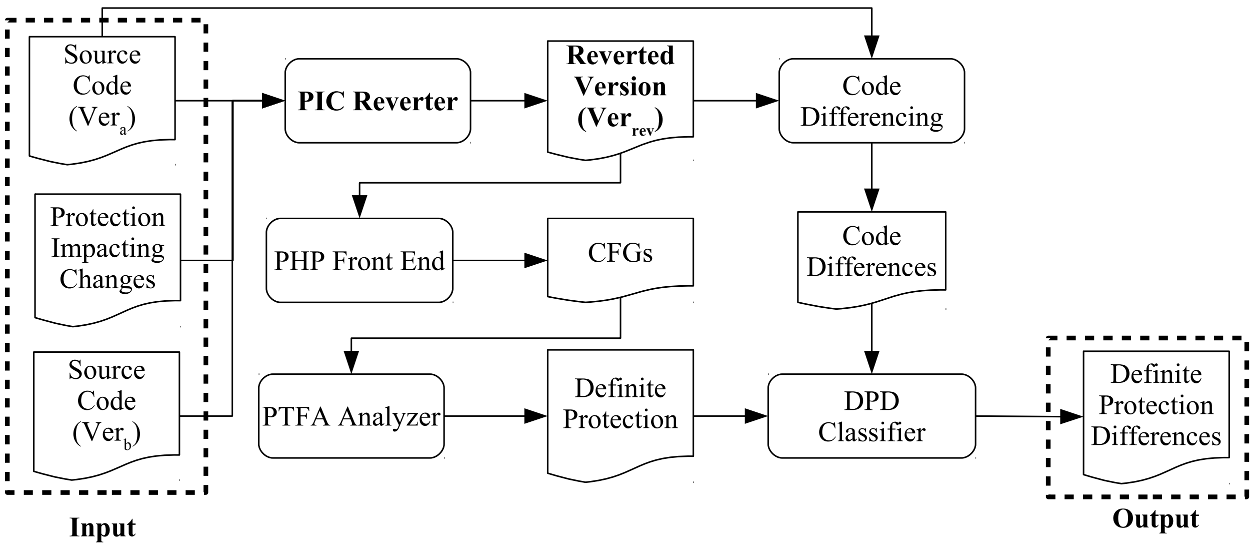


Figure 7.13 Workflow for Reversing Protection-Impacting Changes. New Components bolded.

Dependent and Independent Variables

In these strategies, we inherit the independent variables from our survey, which we will not repeat here. Additional independent variables are which protection-impacting changes we revert and the reversal method. The dependent variable is definite protection differences.

Data Collection and Processing (General)

For these two strategies, we use the source code, line differences, PTFA models and protection-impacting changes from our survey of WordPress and MediaWiki (c.f. Section 7.4.1).

The following general approach is shared by the two strategies: we create a reverted version Ver_{rev} and compare definite protection between Ver_a and Ver_{rev}. As illustrated in Figure 7.13, Ver_{rev} is a function of the source code of Ver_a and Ver_b, as well as protection-impacting changes. The protection-impacting change reverter creates Ver_{rev} by copying contains all source code files of Ver_a, except for files of Ver_b which have no protection-impacting changes (c.f. our decision chart in Appendix E).

Data Collection and Processing (Complete Reversal)

In our first strategy, we create Ver_{rev} for all protection-impacting changes we found between Ver_a and Ver_b. We consider that the reversal is successful if there are no of definite protection differences between Ver_a and Ver_{rev}.

Data Collection and Processing (Individual Reversal)

A definite protection difference may also depend on synchronous multiple changes. In this evaluation strategy, we separate the problems of individual DPD reversal from the multiple DPD reversal. We create a reverted release pair for each definite protection difference and examine if that definite protection difference disappeared.

We describe the high-level algorithm in Figure 7.14. We do the following for each definite protection difference. First, we compute the protection-impacting changes for this definite protection difference, PIC_{DPD}. Then, we create a reverted version Ver_{rev} for PIC_{DPD}. Afterwards, we calculate $vertexMap_{rev}$, which is $vertexMap$ between Ver_a and Ver_{rev}. Finally, we compare the definite protection of the vertex in Ver_a with the appropriate vertex in Ver_{rev}. In this algorithm, DPD is a set of 2-tuples $(v_b, priv)$, where v_b belongs to Ver_b, and v_b is a definite protection difference for privilege $priv$. The function $calculatePIC$ determines protection-impacting changes. The function $revert$ creates a reverted version, and $BuildVertexMap$ calculates a vertex map between two versions (defined in Appendix C).

```
**for all** $(v_b, priv) \in DPD$
  $v_a \leftarrow bMap(v_b)$
  $PIC_{DPD} \leftarrow calculatePIC(v_a, v_b, priv)$
  $Ver_{rev} \leftarrow revert(Ver_a, Ver_b, PIC_{DPD})$
  $vertexMap_{rev} \leftarrow BuildVertexMap(Ver_a, Ver_{rev})$
  $v_{rev} \leftarrow vertexMap_{rev}(v_a)$
  **if** $DefProt(v_a, priv) = DefProt(v_{rev}, priv)$
    print $SUCCESS$
  **else**
    print $FAILURE$
```

Figure 7.14 Algorithm for Individual Reversal

We do so for all DPD in the latest security-affected release pair of each system. These release pairs are large (333 KLOC for WordPress and 197 KLOC for MediaWiki) and contain a mixture of functional and RBAC security changes and appear representative of the code changes in other release pairs. For WordPress the release pair is 4.7.1 vs 4.7.2. This release pair has 87 definite protection differences and its $allPIC = 84$. For MediaWiki, the release pair is 1.28.2 vs 1.29.0. This release pair has 46 definite protection differences and its $allPIC =$37,532.

As indicated in Figure 7.14, the reversal of PIC_{DPD} is successful if v_{rev} has the same definite privilege protection for $priv$ as v_a.

Implementation

We have implemented the protection-impacting change reverter in Java.

7.5 Experimental Results

We ran our experiments on a system powered by an i7950@3.07GHz CPU. We configured the OpenJDK Java VM version 1.8.0_131 to use up to 8Gb of RAM for WordPress and 12 Gb of RAM for MediaWiki.

7.5.1 RQ1 : Reduction of Code Changes to Evaluate

In order to answer **RQ1** *Is the size of protection-impacting changes smaller than the set of code changes?*, we computed protection-impacting changes for 210 release pairs of WordPress and 192 release pairs of MediaWiki. We summarize the release pairs, execution time, security-affected release pairs and total protection-impacting changes in Table 7.1.

Table 7.1 Survey Summary

	WordPress	**MediaWiki**
Release Pairs	210	192
Total Execution Time (hours)	34	58
Average Execution Time (minutes)	9.72	17.42
Gain-Affected Release Pairs	85	30
Loss-Affected Release Pairs	46	30
Security-Affected Release Pairs	87	42
Total $allPIC$	13,307,209	87,263,613
Total $allPIC_{lines}$	258,997	881,767

In Tables 7.2 and 7.3, we present summary statistics of the measures we defined in Section 7.4.1. We report two distributions: the distribution for all security-impacted release pairs and the distribution for all release pairs. We present the latter distribution because it gives a picture of protection-impacting changes over the lifetime of the project. We turn our attention to the statistics for $allPIC_{lines}$ and $relAllPIC_{lines}$ in Tables 7.2 and 7.3. We found that, on average, security-impacted release pairs had 2976.98 (26.28%) protection-impacting lines of code for WordPress and 20,994.45 (15.45%) for MediaWiki. The median is different, with 47.00 (27.41%) and 13,540.00 (13.85%) protection-impacting lines of code, respectively. This difference is due to outliers in the distribution, which we discuss below.

We also observe that some metrics' distributions have a mean greater than their 3rd quartile. This is notably the case for PIC for both Ver_a and Ver_b. This situation is due to a few very large outliers, which we discuss below.

In MediaWiki (Table 7.3), there is a large difference between the relative measures at the PTFA level and at the line level. This is explained by the fact that a large proportion of code changes belonged to tests and language support folders. Test code was bundled in the releases up to the 1.15 series. Release archives from 1.16 onwards (except for the 1.20 series) have no test code. Language support folders have many PHP files that mostly consist of string definitions in a dictionary data structure. This represents very few added and deleted edges. Therefore, language support changes affect relative line measures more than relative edge measures.

If we separate the PIC results according to code deleted (Ver_a) and added (Ver_b), we obtain the respective medians of 14.00 and 7391.00 LOC. In relative terms ($relPIC$), these terms represent 31.58% and 12.86% of their respective code changes. The mean code changes are 974.41 LOC (30.76%) and 12,709.67 LOC (16.16%) for Ver_a and Ver_b, respectively. While it appears that deleted code is more likely to be protection-impacting than added code, this difference is not statistically significant (t-test $p = 0.065$ for WordPress and $p = 0.601$ for MediaWiki).

Over all release pairs in our survey, protection-impacting changes are an even smaller percentage of code changes. For both systems, the median protection-impacting changes is not significant, since most release pairs have no definite protection differences, and therefore no protection-impacting changes. However, the mean protection-impacting changes respectively represents 10.89% and 3.38% of code changes for WordPress and MediaWiki.

Table 7.2 – Protection-Impacting Changes Per Release Pair (WordPress)

Measure	**1st quartile**	**Median**	**Mean**	**Standard Dev**	**3rd quartile**
PIC Release Pairs (87 / 210)					
PIC_{loss} (Ver_a)	0.00	25.00	34,505.40	124,328.33	14,400.00
$relPIC_{loss}$ (Ver_a)	0.00%	5.22%	16.76%	26.51%	20.11%
PIC_{loss} (Ver_b)	0.00	60.00	10,343.75	20,491.79	11,044.00
$relPIC_{loss}$ (Ver_b)	0.00%	0.71%	5.40%	8.48%	8.80%
$PIC_{loss,lines}$ (Ver_a)	0.00	2.00	773.20	1728.18	1065.50
$relPIC_{loss,lines}$ (Ver_a)	0.00%	4.66%	10.86%	15.28%	15.56%
$PIC_{loss,lines}$ (Ver_b)	0.00	2.00	1624.29	3257.90	1820.50
$relPIC_{loss,lines}$ (Ver_b)	0.00%	3.24%	10.46%	14.39%	17.64%

Table 7.2 – Protection-Impacting Changes Per Release Pair (WordPress – Continued)

Measure	**1st quartile**	**Median**	**Mean**	**Standard Dev**	**3rd quartile**
PIC_{gain} (Ver_a)	119.00	302.00	6017.15	15,015.23	5179.00
$relPIC_{gain}$ (Ver_a)	8.01%	16.86%	24.64%	20.58%	37.12%
PIC_{gain} (Ver_b)	307.00	869.00	113,270.61	282,525.43	62,417.00
$relPIC_{gain}$ (Ver_b)	20.36%	36.64%	41.30%	25.09%	54.94%
$PIC_{gain,lines}$ (Ver_a)	6.00	14.00	744.53	1784.40	677.50
$relPIC_{gain,lines}$ (Ver_a)	9.09%	20.32%	27.48%	23.44%	37.50%
$PIC_{gain,lines}$ (Ver_b)	14.50	34.00	1662.40	3440.55	1492.50
$relPIC_{gain,lines}$ (Ver_b)	10.66%	22.19%	22.88%	15.64%	31.73%
PIC (Ver_a)	138.50	364.00	36,927.74	125,242.10	15,398.50
$relPIC$ (Ver_a)	16.97%	33.15%	38.18%	25.49%	51.72%
PIC (Ver_b)	307.00	869.00	116,028.69	282,644.22	66,650.50
$relPIC$ (Ver_b)	24.30%	43.35%	43.77%	24.17%	57.18%
PIC_{lines} (Ver_a)	6.00	14.00	974.41	2169.55	1188.50
$relPIC_{lines}$ (Ver_a)	10.82%	31.58%	30.76%	23.69%	42.12%
PIC_{lines} (Ver_b)	14.50	34.00	2002.56	3912.68	2447.50
$relPIC_{lines}$ (Ver_b)	11.95%	24.00%	25.07%	15.97%	34.49%
$allPIC$	473.00	1171.00	152,956.43	379,643.71	93,863.50
$relAllPic$	27.58%	37.45%	42.15%	22.10%	56.52%
$allPIC_{lines}$	23.00	47.00	2976.98	5754.48	4026.50
$relAllPIC_{lines}$	12.74%	27.41%	26.28%	16.50%	36.07%
$deletedEdges$	423.00	1314.00	64,239.03	171,739.81	48,458.00
$addedEdges$	753.50	3279.00	167,956.75	338,315.67	112,930.00
$deleted$ (LOC)	24.00	109.00	3060.62	5686.95	4117.50
$added$ (LOC)	75.00	341.00	6446.60	11,006.58	9770.00
All Release Pairs (210)					
PIC_{loss} (Ver_a)	0.00	0.00	14,295.10	81,552.48	0.00
$relPIC_{loss}$ (Ver_a)	0.00%	0.00%	6.94%	18.91%	0.00%
PIC_{loss} (Ver_b)	0.00	0.00	4285.27	14,102.26	0.00
$relPIC_{loss}$ (Ver_b)	0.00%	0.00%	2.24%	6.05%	0.00%
$PIC_{loss,lines}$ (Ver_a)	0.00	0.00	320.32	1172.48	0.00
$relPIC_{loss,lines}$ (Ver_a)	0.00%	0.00%	4.50%	11.17%	0.00%
$PIC_{loss,lines}$ (Ver_b)	0.00	0.00	672.92	2238.46	0.00
$relPIC_{loss,lines}$ (Ver_b)	0.00%	0.00%	4.33%	10.58%	0.00%
PIC_{gain} (Ver_a)	0.00	0.00	2492.82	10,079.65	193.00
$relPIC_{gain}$ (Ver_a)	0.00%	0.00%	10.21%	17.95%	11.65%
PIC_{gain} (Ver_b)	0.00	0.00	46,926.40	189,665.60	378.00
$relPIC_{gain}$ (Ver_b)	0.00%	0.00%	17.11%	25.98%	27.87%
$PIC_{gain,lines}$ (Ver_a)	0.00	0.00	308.45	1202.23	9.00
$relPIC_{gain,lines}$ (Ver_a)	0.00%	0.00%	11.38%	20.26%	16.40%
$PIC_{gain,lines}$ (Ver_b)	0.00	0.00	688.71	2354.71	24.00
$relPIC_{gain,lines}$ (Ver_b)	0.00%	0.00%	9.48%	15.11%	16.54%
PIC (Ver_a)	0.00	0.00	15,298.63	82,382.24	193.00
$relPIC$ (Ver_a)	0.00%	0.00%	15.82%	24.95%	30.00%
PIC (Ver_b)	0.00	0.00	48,069.03	190,144.31	380.00
$relPIC$ (Ver_b)	0.00%	0.00%	18.13%	26.60%	32.47%
PIC_{lines} (Ver_a)	0.00	0.00	403.69	1472.53	10.50
$relPIC_{lines}$ (Ver_a)	0.00%	0.00%	12.74%	21.48%	18.41%
PIC_{lines} (Ver_b)	0.00	0.00	829.63	2697.62	24.00
$relPIC_{lines}$ (Ver_b)	0.00%	0.00%	10.39%	16.07%	19.51%
$allPIC$	0.00	0.00	63,367.66	254,972.56	573.00

Table 7.2 – Protection-Impacting Changes Per Release Pair (WordPress – Continued)

Measure	**1st quartile**	**Median**	**Mean**	**Standard Dev**	**3rd quartile**
relAllPic	0.00%	0.00%	17.46%	25.18%	33.29%
$allPIC_{lines}$	0.00	0.00	1233.32	3973.24	38.00
$relAllPIC_{lines}$	0.00%	0.00%	10.89%	16.74%	22.57%
deleted (LOC)	15.00	42.00	1338.00	3932.58	182.50
added (LOC)	38.00	117.50	2841.61	7698.90	482.25

Table 7.3 – Protection-Impacting Changes Per Release Pair (MediaWiki)

Measure	**1st quartile**	**Median**	**Mean**	**Standard Dev**	**3rd quartile**
PIC Release Pairs (42 / 192)					
PIC_{loss} (Ver_a)	0.00	68,108.50	766,078.14	1,828,892.88	831,027.50
$relPIC_{loss}$ (Ver_a)	0.00%	91.15%	64.88%	42.51%	94.04%
PIC_{loss} (Ver_b)	0.00	81,114.00	236,206.57	348,820.08	390,433.00
$relPIC_{loss}$ (Ver_b)	0.00%	17.80%	17.40%	14.87%	34.45%
$PIC_{loss,lines}$ (Ver_a)	0.00	2959.00	7257.07	9181.69	14,438.25
$relPIC_{loss,lines}$ (Ver_a)	0.00%	10.59%	14.83%	13.56%	27.97%
$PIC_{loss,lines}$ (Ver_b)	0.00	4549.50	11,282.86	14,117.74	20,160.25
$relPIC_{loss,lines}$ (Ver_b)	0.00%	6.93%	10.35%	10.55%	18.71%
PIC_{gain} (Ver_a)	0.00	29,434.00	127,190.43	194,685.73	202,035.25
$relPIC_{gain}$ (Ver_a)	0.00%	20.15%	18.46%	18.97%	35.09%
PIC_{gain} (Ver_b)	0.00	150,787.00	1,241,225.79	2,129,286.82	1,448,084.75
$relPIC_{gain}$ (Ver_b)	0.00%	91.33%	62.00%	42.73%	95.42%
$PIC_{gain,lines}$ (Ver_a)	0.00	2892.50	6751.93	8868.15	11,998.50
$relPIC_{gain,lines}$ (Ver_a)	0.00%	5.90%	11.16%	13.13%	23.62%
$PIC_{gain,lines}$ (Ver_b)	0.00	5696.50	11,106.17	14,108.29	18,202.50
$relPIC_{gain,lines}$ (Ver_b)	0.00%	10.44%	13.78%	14.30%	22.10%
PIC (Ver_a)	6313.25	117,906.00	789,400.86	1,819,781.00	831,249.50
$relPIC$ (Ver_a)	37.91%	91.15%	72.40%	32.87%	94.18%
PIC (Ver_b)	64,220.75	541,058.50	1,288,304.21	2,104,996.71	1,448,092.25
$relPIC$ (Ver_b)	35.18%	91.33%	70.02%	32.08%	95.44%
PIC_{lines} (Ver_a)	256.50	4231.00	8284.79	8816.45	15,216.50
$relPIC_{lines}$ (Ver_a)	6.30%	17.81%	17.64%	12.76%	28.42%
PIC_{lines} (Ver_b)	1434.50	7391.00	12,709.67	13,576.70	20,160.25
$relPIC_{lines}$ (Ver_b)	6.62%	12.86%	16.16%	13.08%	23.50%
allPIC	138,279.25	709,668.50	2,077,705.07	3,063,388.87	2,526,230.25
relAllPic	45.17%	88.71%	70.26%	28.99%	94.44%
$allPIC_{lines}$	3380.25	13,540.00	20,994.45	21,479.58	37,212.00
$relAllPIC_{lines}$	6.16%	13.85%	15.45%	10.43%	24.34%
deletedEdges	9735.25	278,466.50	894,557.50	1,927,881.29	858,968.75
addedEdges	90,331.25	693,545.50	1,459,471.71	2,166,280.91	1,561,758.00
deleted (LOC)	9609.00	45,120.50	42,694.64	33,808.46	61,079.00
added (LOC)	24,161.00	65,793.50	76,603.57	62,964.74	99,749.75
All Release Pairs (192)					
PIC_{loss} (Ver_a)	0.00	0.00	167,579.59	904,889.78	0.00
$relPIC_{loss}$ (Ver_a)	0.00%	0.00%	14.19%	33.33%	0.00%
PIC_{loss} (Ver_b)	0.00	0.00	51,670.19	188,954.40	0.00
$relPIC_{loss}$ (Ver_b)	0.00%	0.00%	3.81%	9.98%	0.00%
$PIC_{loss,lines}$ (Ver_a)	0.00	0.00	1587.48	5209.99	0.00
$relPIC_{loss,lines}$ (Ver_a)	0.00%	0.00%	3.24%	8.79%	0.00%
$PIC_{loss,lines}$ (Ver_b)	0.00	0.00	2468.12	8040.75	0.00

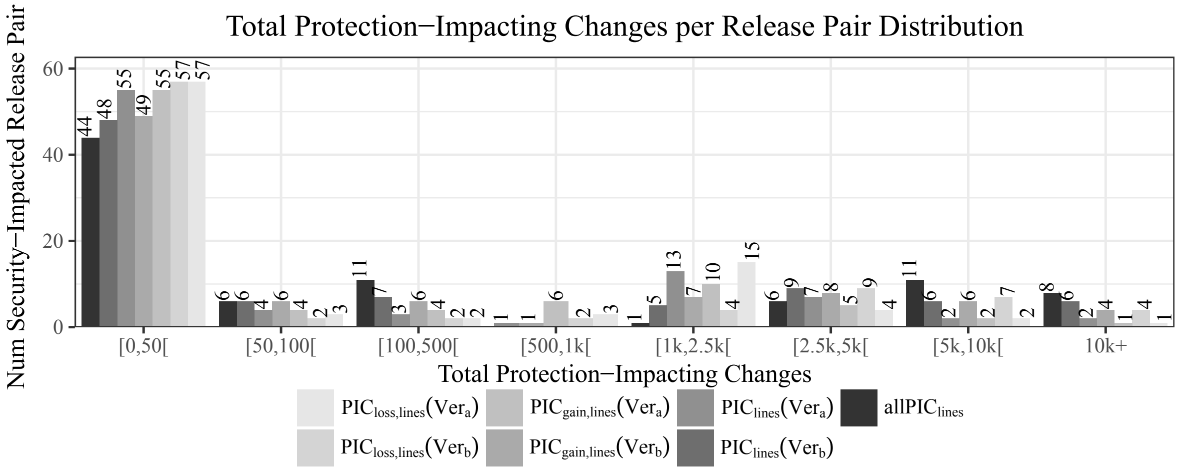
Table 7.3 – Protection-Impacting Changes Per Release Pair (MediaWiki – Continued)

Measure	**1st quartile**	**Median**	**Mean**	**Standard Dev**	**3rd quartile**
$relPIC_{loss,lines}$ (Ver_b)	0.00%	0.00%	2.26%	6.50%	0.00%
PIC_{gain} (Ver_a)	0.00	0.00	27,822.91	104,476.38	0.00
$relPIC_{gain}$ (Ver_a)	0.00%	0.00%	4.04%	11.65%	0.00%
PIC_{gain} (Ver_b)	0.00	0.00	271,518.14	1,112,613.67	0.00
$relPIC_{gain}$ (Ver_b)	0.00%	0.00%	13.56%	32.44%	0.00%
$PIC_{gain,lines}$ (Ver_a)	0.00	0.00	1476.98	4971.27	0.00
$relPIC_{gain,lines}$ (Ver_a)	0.00%	0.00%	2.44%	7.64%	0.00%
$PIC_{gain,lines}$ (Ver_b)	0.00	0.00	2429.47	7994.80	0.00
$relPIC_{gain,lines}$ (Ver_b)	0.00%	0.00%	3.01%	8.75%	0.00%
PIC (Ver_a)	0.00	0.00	172,681.44	904,389.96	0.00
$relPIC$ (Ver_a)	0.00%	0.00%	15.84%	33.65%	0.00%
PIC (Ver_b)	0.00	0.00	281,816.55	1,111,885.51	0.00
$relPIC$ (Ver_b)	0.00%	0.00%	15.32%	32.60%	0.00%
PIC_{lines} (Ver_a)	0.00	0.00	1812.30	5336.38	0.00
$relPIC_{lines}$ (Ver_a)	0.00%	0.00%	3.86%	9.40%	0.00%
PIC_{lines} (Ver_b)	0.00	0.00	2780.24	8204.77	0.00
$relPIC_{lines}$ (Ver_b)	0.00%	0.00%	3.53%	9.03%	0.00%
$allPIC$	0.00	0.00	454,497.98	1,660,135.18	0.00
$relAllPic$	0.00%	0.00%	15.37%	32.07%	0.00%
$allPIC_{lines}$	0.00	0.00	4592.54	13,219.64	0.00
$relAllPIC_{lines}$	0.00%	0.00%	3.38%	8.02%	0.00%
deleted (LOC)	16.25	121.50	15,535.27	64,917.30	7589.75
added (LOC)	37.75	245.50	20,608.67	42,949.09	17,595.25

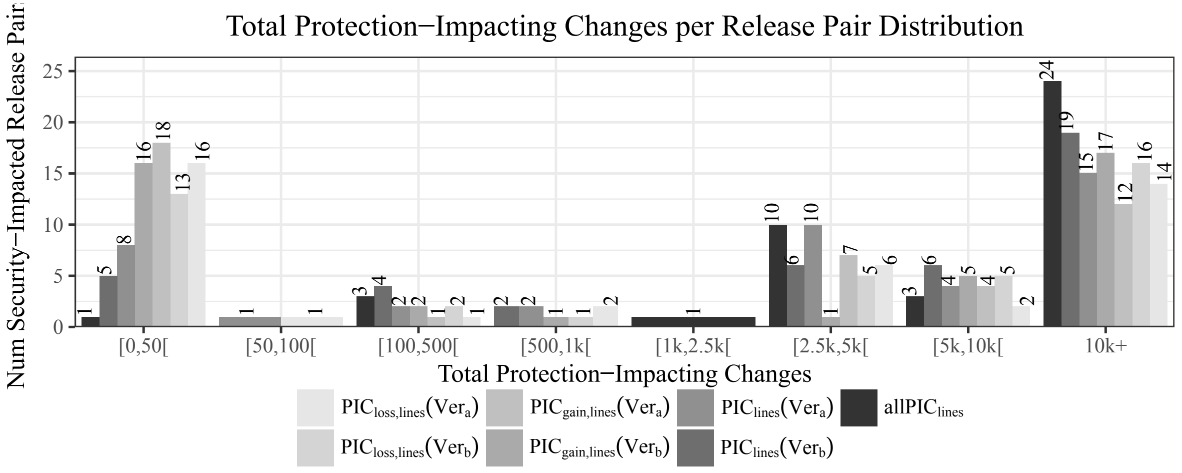
We show the histogram of protection-impacting changes per release pair for security-impacted release pairs in Figure 7.15. In these histograms, we report the measures we defined in Section 7.4.1. These metrics' distributions for WordPress are in Figure 7.15a and in Figure 7.15b for MediaWiki. We used bins of different widths to better represent the distribution, which results in a non-linear X axis.

In Figure 7.15, we observe that, among security-impacted release pairs, 50/87 (57%) and 1/42 (2%) release pairs, respectively of WordPress and MediaWiki, have less than 100 lines of code classified as protection-impacting changes. We also observe that respectively 61/87 (70%) and 4/42 (10%) security-impacted release pairs have less than 500 protection-impacting lines of code. Respectively for WordPress and MediaWiki, 26/87 (30%) and 38/42 (90%) release pairs have 500 or more lines of protection-impacting changes. The distributions are very different between WordPress and MediaWiki. The WordPress distributions are long-tailed, with outliers on the right-hand side of the distribution. We found 8/87 (9%) release pairs for which $allPIC_{lines} \geq$ 10,000. In the case of MediaWiki, the distributions are skewed towards the right with a few outliers on the left. We find 24/42 (57%) release pairs for which $allPIC_{lines} \geq$ 10,000. The outliers also explain the gap between the medians and means in Tables 7.2 and 7.3.

For both systems, release pairs with a large number of protection impacting changes ($allPIC \geq$ 10,000) also have a large volume of code changes. For WordPress, these release pairs have a mean of over 31 KLOC and 16 KLOC added and deleted lines, respectively. The relative proportion of protection-impacting change in these release pairs is not very high, with a mean $relAllPIC$ of 38.75%. For MediaWiki, the mean code changes for these release pairs is about 108 KLOC and 61 KLOC, respectively for added and deleted code. The relative protection-impacting change measure is relatively small, with a mean $relAllPIC$ of 21.21%. We also observe that all these release pairs have a minor or major release number (i.e. a .0 release) for Ver_b. We discuss these observations further in Section 7.6.1



(a) WordPress



(b) MediaWiki

Figure 7.15 Distribution of Protection-Impacting Lines of Code per Release Pair for Security-Impacted Release Pairs

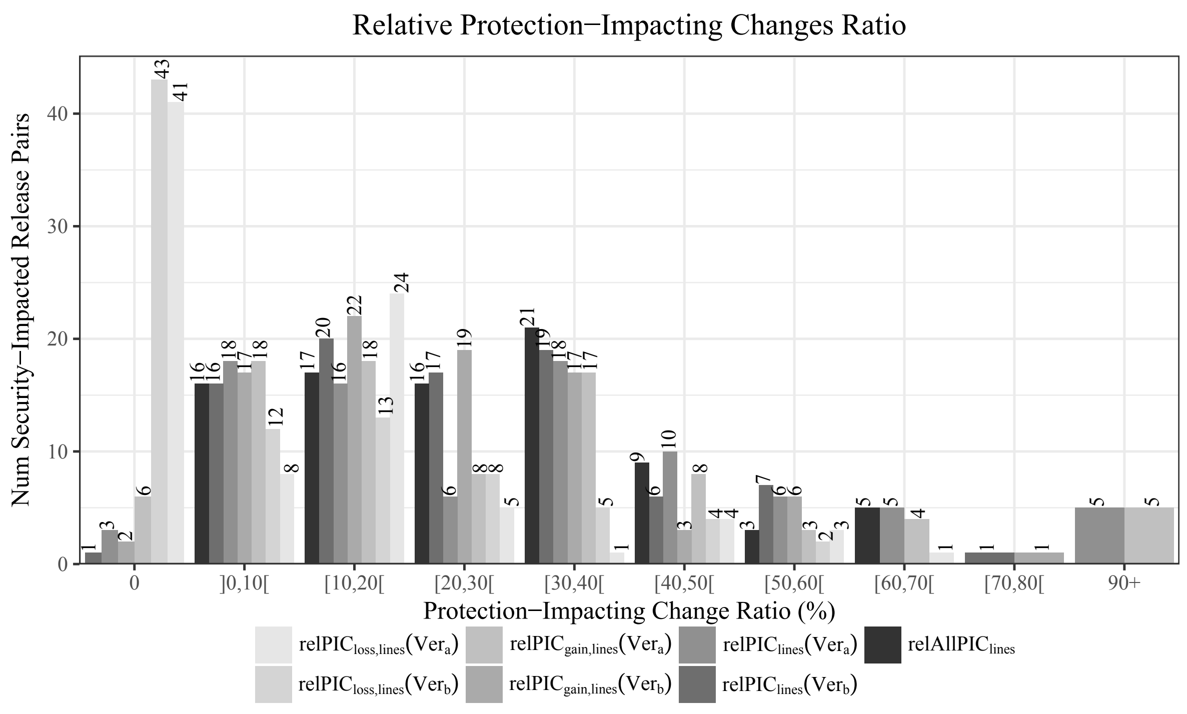
We present an histogram of the relative distributions $relPIC_{loss}$, $relPIC_{gain}$, $relPIC$ and $relAllPIC$ for security-impacted release pairs in Figure 7.16. The distributions for WordPress are in Figure 7.16a, and those of MediaWiki are in Figure 7.16b. For WordPress, we first observe the large spike at zero in the distribution of $relPIC_{loss}$. This is explained by the presence of 46 release pairs have no loss-affected vertices whatsoever.

We also observe that $relAllPIC_{lines}$ is somewhat skewed towards the left, especially for MediaWiki. We find that protection-impacting changes represent less than 10% of code changes ($relAllPIC_{lines} < 0.1$) in 16/87 (18%) and 18/42 (43%) release pairs, respectively of WordPress and MediaWiki. We also observe that respectively 42/87 (48%) and 32/42 (76%) security-impacted release pairs have less than 25% of changed lines of code considered as protection-impacting ($relAllPIC_{lines} < 0.25$). We also see that the distribution has for WordPress has 8/87 (9%) security-impacted release pairs above 50%. However, no release pairs of MediaWiki have $relAllPIC_{lines} > 0.5$. Thus, protection-impacting changes are generally few compared to the total number of code changes between versions.

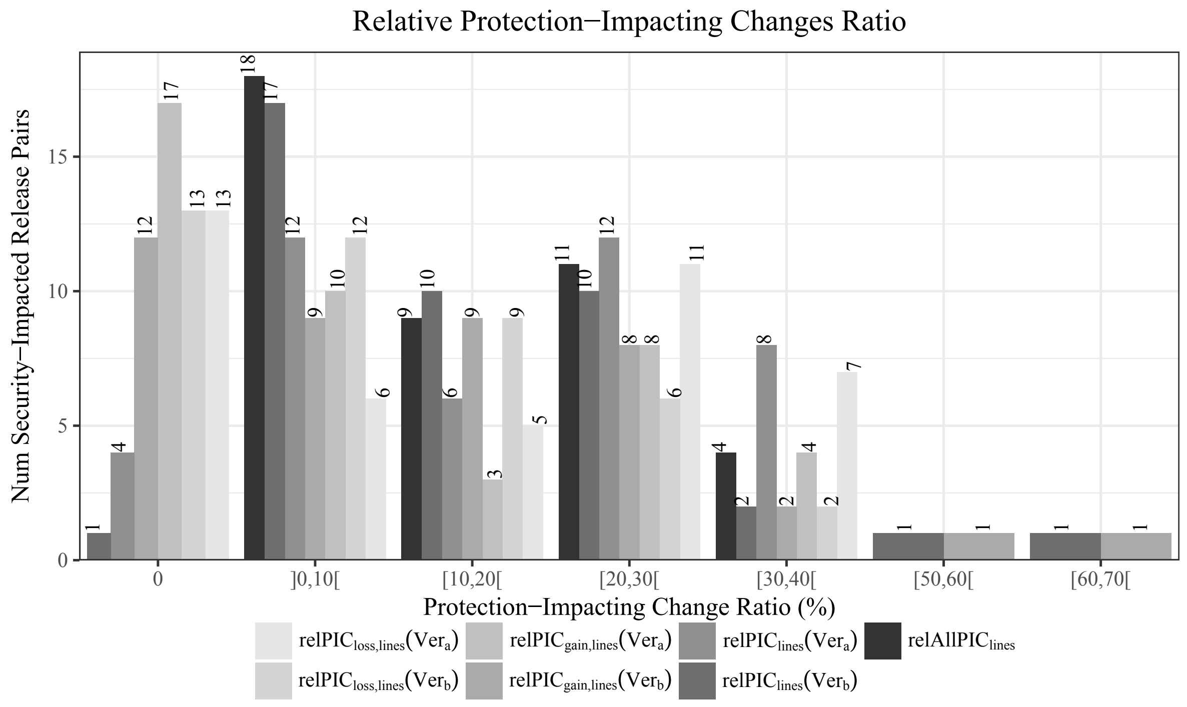
For WordPress, two distributions have large outliers, $relPIC_{gain}$ and $relPIC$ for Ver_a. These occur for release pairs with a small number of changes (<100 LOC) and appear to be very targeted. The difference in shape between Figures 7.15 and 7.16 is explained by the large volume of changes. For instance, the releases which have over 5000 protection-impacting changes ($allPIC_{lines} > 5000$) show a large amount of code changes. In these release pairs, the largest measure of $relAllPIC_{lines}$ is about 54% for WordPress and 20% for MediaWiki.

The median of $relAllPIC_{lines}$ (27.41% for WordPress and 13.85% for MediaWiki) is very encouraging. We discuss this result further in Section 7.6.1.

For WordPress, the median protection-impacting changed lines of code, for security-impacted release pairs, is 47 LOC. This corresponds to about 27% of total code changes.
For MediaWiki, the median protection-impacting changed lines of code, for security-impacted release pairs, is 13,540 LOC. This corresponds to about 14% of total code changes.



(a) WordPress



(b) MediaWiki

Figure 7.16 Distribution of Protection-Impacting Lines of Code per Release Pair for Security-Impacted Release Pairs

7.5.2 RQ2 : PICs containing all DPD root causes

To answer **RQ2** "How often PICs contain all DPD root causes in a release pair?", we reverted protection-impacting changes for the security-affected release pairs of our survey. By doing so, we determine how many release pairs' definite protection differences is exclusively explained by DPD root causes whose code changes belong to paths.

Evaluation Using Reversal Strategies

We reverted protection-impacting changes using complete reversal and individual reversal. We summarize the computation time and the success rate in Table 7.4.

We performed complete reversal, meaning that we reverted all protection-impacting changes in security-impacted release pairs. We created reverted versions and computed definite privilege protection for all reverted release pairs for WordPress and MediaWiki. We summarize our results in Table 7.4a.

Protection-impacting changes contain all DPDroot causes in 87% to 93% of cases.

In addition, we also performed individual reversal. We reverted all protection-impacting changes for every definite protection difference in one release pair of each system. To do so, we computed protection-impacting changes and created a reverted version for each definite protection difference. Then, we determined if the definite protection difference disappeared. All reverts were successful. We summarize the details of the release pairs studied, as well as the processing time, in Table 7.4b. This indicates that our approach is suitable for finding the root causes of individual definite protection differences as well as those of multiple definite protection differences.

Our approach is suitable for finding the root causes of individual definite protection differences as well as those of multiple definite protection differences.

These results are encouraging, and we discuss them further in Section 7.6.2.

7.6 Discussion

When definite protection differences are detected and are due to DPD root causes whose code changes belong to paths, developers take advantage of protection-impacting changes to focus their review efforts.

Table 7.4 Results of Reverting Definite Protection Differences

	WordPress	**MediaWiki**
Release Pairs	87	42
Total execution time (hours)	6.63	7.09
Average execution time (minutes)	4.62	10.13
Success rate	76/87 (87%)	39/42 (93%)

(a) Complete Reversals

	WordPress	**MediaWiki**
Release Pair	4.7.1 vs 4.7.2	1.28.2 vs 1.29.0
Definite Protection Differences	87	46
Total execution time (hours)	11.19	15.06
Average execution time (minutes)	7.72	19.64
Success rate	87/87 (100%)	46/46 (100%)

(b) Individual Reversals

We have seen that protection-impacting changes are a small percentage of code changes in security-affected release pairs, with a median of 27.41% for WordPress and 13.85% for MediaWiki. Using reversal, we also determined that 87% to 93% of release pairs are affected by DPD root causes whose code changes belong to paths. Our approach gives an opportunity to reduce review efforts, though further research is needed to investigate and confirm the impact on developers.

We discuss RQ1 in Section 7.6.1 and then RQ2 in Section 7.6.2. Afterwards, in Section 7.6.3, we discuss the applications as well as further research opportunities.

7.6.1 RQ1: Reduction of Code Changes to Evaluate

Asking our first research question,"*Is the size of protection-impacting changes smaller than the set of code changes?*", we found that protection-impacting changes reduces the number of code changes to evaluate. For protection-impacted release pairs in WordPress, we found that protection-impacting changes are a median of 27.41% of code changes per release pair. In the case of MediaWiki, the median relative protection-impacting changes per release pairs is 13.85%. When definite protection differences are detected, developers can focus their review efforts on the 14%-27% of code changes that are protection-impacting. Over the lifetime of the project, protection-impacting changes are an average of roughly 3% to 11% of code changes. Considering these results, our approach gives an opportunity to reduce review efforts, though further research is needed to confirm this.

We observed that 8 release pairs of WordPress and 24 release pairs of MediaWiki had more than 10,000 protection-impacting changed lines. We also saw that the mean relative measure of protection-impacting changes for these releases is ranging from about 21% to 39%. We also saw that this situation occurs only for release pairs leading to new minor and major (i.e. .0) releases. Even though this large volume of changes is relatively small compared to the total changes, it is likely that their verification and validation would be time-consuming. The regular use of our approach during the development of .0 releases would be especially desirable.

7.6.2 RQ2 : PICs containing all DPD root causes

Our second research question was "*How often PICs contain all DPD root causes in a release pair?*". We evaluated that root causes are a subset of protection-impacting changes in two different ways. In our first evaluation strategy, we reverted all protection-impacting changes at the file level for each release pair of our survey. we found that protection-impacting changes contained DPD root causes in 87% and 93% of release pairs, respectively for WordPress and MediaWiki.

In our second evaluation strategy, we likewise reverted protection-impacting changes. However, we computed and reverted protection-impacting changes individually for each definite protection differences in the latest security-affected release pair of each system. For WordPress, the release pair (4.7.1 vs 4.7.2) had 87 definite protection differences. For MediaWiki, the (1.28.2 vs 1.29.0) release pair had 46 definite protection differences. All individual reversals were successful.

7.6.3 Further Discussions

We focused on DPD root causes whose code changes belong to paths, but also observed definite protection differences caused by changes that occur somewhere else than paths. For instance, code changes that affect interprocedural edge resolution causes may cause definite protection differences.

We show an example in Figure 7.17 in Unified Diff format. In this case, `$a` is a known instance of interface `iFace`. Please imagine that a call graph construction algorithm creates an edge between the call site and all implementations of `iFace::foo`. In this case, the code change for class `B` will cause definite protection differences. However, the code change causing this does not belong to any path between v_0 and the definite protection differences.

In addition to the kind of impact based on class hierarchy changes and reflective calls, we also found definite protection differences caused by conservative approximations in our PTFA engine. For instance, it treats apparently dead code as unprotected entry points. This decision has the consequence of injecting unprotected paths that are not DPD root causes whose code changes belong to paths.

```
  function foo(iFace $a){}
    if (current_user_can('p')){
      $a->foo();
    }
  }

  interface iFace{
    function foo()
  }

  class A implements iFace{
    function foo(){echo "Definitely Protected";}
  }

  //This class now implements iFace
- class B {
+ class B implements iFace{
    function foo(){echo "Gain-Affected";}
  }
```

Figure 7.17 Example of definite protection difference Caused by Interprocedural Edge Resolution. Added and Deleted Code is Respectively Annotated With '+' and '-'

An absence of definite protection differences imply that the privilege protection for that code is just as valid or invalid for version Ver_b as it was for Ver_a.

While our approach does not require one, it may benefit from an oracle that classifies definite protection differences as beneficial, harmful or irrelevant. With such an oracle, we could compute protection-impacting changes only for harmful security changes. We could also report on how many code changes affected security, and on the distribution of protection-impacting changes populations. Suitable oracles include a formal security specification, a classification of security-impacted lines by developers, or a vulnerability oracle. Our approach can be used between any two versions of the application, making it suitable as an early warning system for security issues. For instance, developers could use definite protection difference analysis after each commit. Or they could integrate our approach in automated build systems. A build system could treat definite protection differences as regressions and calculate protection-impacting changes whenever such regressions occur.

Our approach could also be integrated with other formal methods for policy verification. Furthermore, formal verification methods may generate a counter-example that shows a path leading to the violations. Protection-impacting changes could be integrated with these counter-examples, which may make the counter-example more informative. We also envision integrated development environments (IDE) allowing developers to determine definite protection differences and protection-impacting changes interactively.

Developers using on-demand analysis may also find it easier to evaluate and fix vulnerabilities. For instance, they could compute protection-impacting changes for the vulnerable code segment. Then, she may determine which code changes are responsible for the vulnerability and which correction is needed. Additionally, our approach would allow developers to compare the effective definite protection differences against their expectations, making it useful for validating security patches. Developers may find useful know whether new code is definitely protected by a given privilege, or why it is lacking a privilege in its definite protection set.

Our approach may be suited for selecting regression tests. Ideally, one would only need to run the tests executing protection-impacting changes when re-validating the RBAC implementation.

Although our results are promising, further research towards identifying DPD root causes is desirable. Fine-grained (e.g. commit-level) protection-impacting changes surveys should also be performed. The psychological acceptability of our results should be confirmed using human studies.

Further research efforts should also be invested in improving $vertexMap$ construction. Improvements may take the form of from a finer-grained (e.g. token-based) or tree-based differencing methods. The latter relies on comparing abstract syntax trees [52].

As we mentioned in Section 7.2.2, definite protection differences can disappear by reversal or by alternative code changes. We would like to investigate automated repair algorithms, which would generate code changes making definite protection differences disappear.

As we discussed earlier, our approach currently only considers changed path root causes. Our analysis already detects protection-impacting edges that do not correspond to code changes.

However, our heuristics dismiss these edges when reporting protection-impacting changes at the line level. We would like to design additional heuristics in future research. These heuristics would identify the cause of these dismissed protection-impacting edges.

In addition, we would like to devise a decision support system based on our approach (Figure 7.18). This system would compute protection-impacting changes and perform a reversal. The latter step would confirm if the protection-impacting changes all belong to changed paths. In the event of a failed reversal, the PIC Simpliflier would determine the subset of PIC_{loss} and PIC_{gain} that do not correspond to code changes (i.e. the PIC Subset). Using such a decision support system, developers would be informed if the DPD root causes are changed path root causes. When that is the case, they may restrict their re-validation to protection-impacting changes. However, whenever the DPD root causes are not changed path root causes, this decision support system would inform the developer of edges requiring deeper analysis.

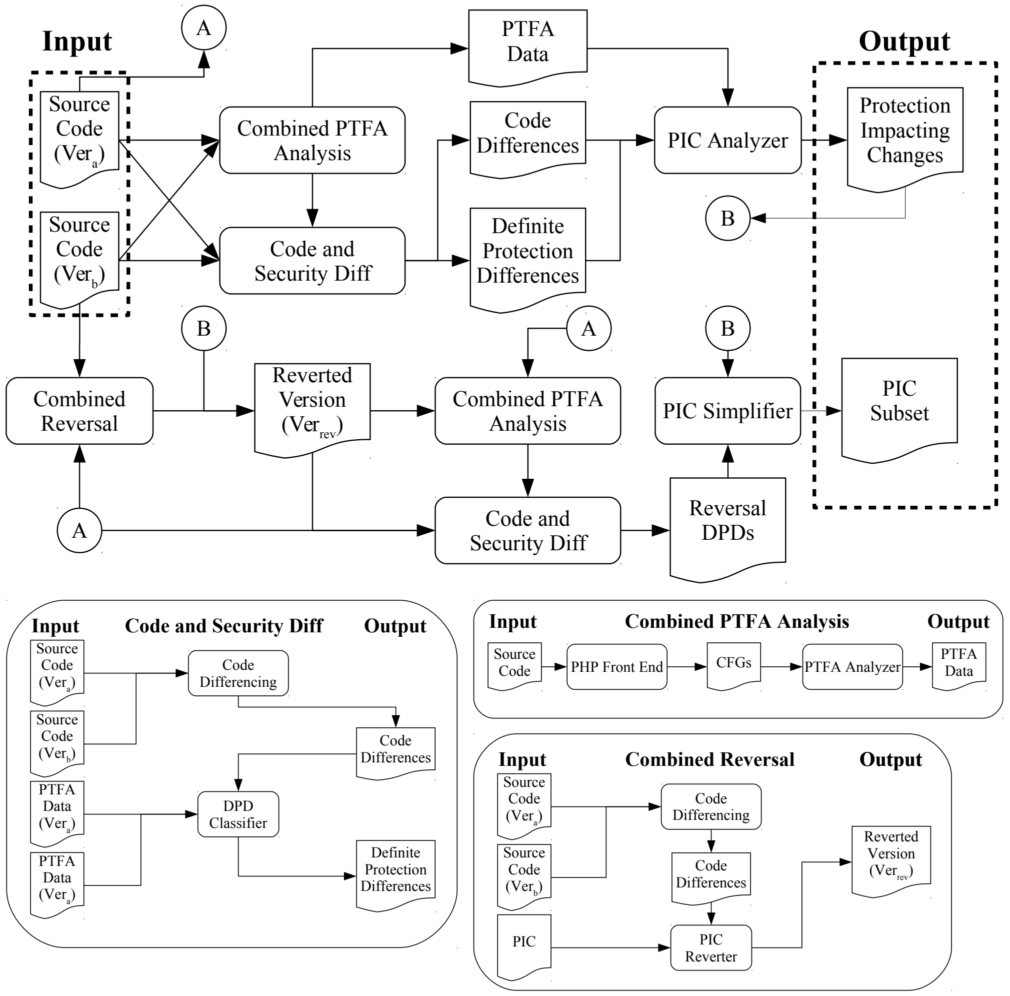


Figure 7.18 Decision Support System for Future Research

7.7 Related Work

Letarte et al. [112] also surveyed the evolution of RBAC systems using PTFA. Their survey was over 31 phpBB releases, which only handled a binary distinction between administrator and unprivileged users. In our study, we detect protection-impacting changes, handle a richer protection scheme, performed a larger survey and considered DPD root causes.

We previously surveyed WordPress using PTFA. In a first study [101], we defined definite protection differences and produced counter-examples for loss-affected code. We also proposed a classification of definite protection differences [103] and reported on the demography of the classification categories. In contrast, this paper defines and surveys protection-impacting changes and considers DPD root causes. It is also not aimed a the classification and distribution of these differences. We previously defined protection-impacting changes [100] and reported a survey of protection-impacting changes of 210 release pairs of WordPress. We also reported a manual evaluation of protection-impacting changes that found that protection-impacting changes caused real definite protection differences in 75% of cases. In this study, we survey an additional system for protection-impacting changes. We also evaluated if protection-impacting changes contains root causes by reverting changes.

Some researchers studied software evolution from the perspective of access control systems. Montrieux et al. published a position paper [134] showing that general software evolution approaches, aimed at consistency checking, were not sufficient for the evolution of RBAC. They do not address code-level RBAC implementation, however.

Hwang et al. published one of the first evolution studies of access control systems [80]. They studied the SELinux policies for the NCSU Virtual Computing Lab. They found that growth was linear and that the policies were changing frequently. This study did not consider implementation of a policy in code, however.

Formal methods exists to analyze RBAC policies. Margrave [55] is a tool leveraging formal methods to support access control evolution. It operates over policies specified using the eXtensible Access Control Markup Language (XACML) [138]. Margrave translates XACML policies to decision diagrams and differences the policies. However, Margrave does not take the implementation of the RBAC policy in consideration.

Other approaches use formal static analysis methods for RBAC systems. One system, RoleCast [167], automatically detects privilege protection checks in PHP and JSP applications. RoleCast uses inferences to identify critical variables controlling the reachability of security sensitive statements. Then, RoleCast infers roles and determines if checks are performed consistently. When RoleCast reports a potential vulnerability, it mentions many details, including the security sensitive statement, its calling context, and which security variables are erroneously verified. Our contribution is very different. Our analysis is comparative, meaning that we report differences in definite privilege protection. Also, we rely on *a priori* knowledge of code patterns used for RBAC privilege checks. We determine protection-impacting changes with graph reachability and code differences, thereby connecting definite protection differences to their potential causes. In addition, our study considers DPD root causes.

An extension tool of RoleCast, is Fix Me Up [168]. It statically finds access-control errors of omission and proposes candidate repairs. The approach generates an access control template, detects deviations from this template and changes them to conform. Their algorithm relies on slicing, a more complex algorithm than ours, and does not detect definite protection differences.

Another approach intended to detect violations is XACML2ASP from Ahn et al. [4]. XACML2ASP uses Answer Set Programming (ASP) to verify, compare and query XACML policies. They leverage ASP solvers to do so. Their method detects policy violations whenever XACML policies are enriched with constraints such as separation of duty. Their approach does not consider implementation of the policy in code, however.

Analyzing RBAC policy evolution is also useful for testing. Hwang et al. [81] use semantic differencing of XACML policies for test selection. This has a dynamic component, as it requires running the full test suite at least once. Le Traon et al. [108] used testing to detect hidden and implicit security mechanisms. They override the policy decision point (PDP) and compare the test results. In comparison, our approach is entirely static and identifies potential root causes for protection changes.

Huang et al. [76] developed the selective access control regression test tool (SACRT). Their test selection method also differences XACML policy files. They validated their approach on four test systems from the financial industry. Our approach is different since we do not require a formal policy and we consider definite protection differences, protection-impacting changes and DPD root causes.

It is also possible to generate test cases for systems using RBAC. Pretschner et al. [150] created a tool that generates test cases for an RBAC policy. They used combinatorial testing of the roles, permissions, and contexts hierarchies. The tests are refined using constraints and generate tests in Java. Our approach is different, as we do not consider test generation. Also, their study does not look into definite protection differences, protection-impacting changes and DPD root causes.

Model checking is also useful to identify candidate causes of errors. Ball et al. [20] proposed an method which identifies transitions in an error trace that are in no correct trace and injects halt statements to produce additional error traces. The authors leveraged information from a model checker to determine the cause of an error. Their approach uses both correct and incorrect traces and identifies transitions in an error trace that are in no correct trace. Then, they inject halt statements at each cause's location and produce additional error traces. Another model checking approach also informs the user of a small set of candidate causes for the error [66]. In this study, they used a distance metric that considers causal dependence to do so. They also express counter-example generation as a minimization problem, on both the length and semantic axes. While both approaches are also based on model checking, ours is concerned with privilege protection and depends on code changes.

Bugginings [166] identifies the cause of a bug in the context of software evolution by differencing dependence graphs. Bugginings did not consider RBAC security. Finally, Bugginings [166] identifies the cause of a bug in the context of software evolution. It does not rely on model checking. Their approach relies on differencing dependence graphs and may investigate multiple versions. In contrast, our approach targets only definite protection differences, We also determine differences at the CFG level and rely on PTFA backwards reachability.

7.8 Threats to Validity

Threats to internal validity refer to confounding variables that may influence our results. Our results depend on the accuracy of the PTFA engine we used. Because PHP applications like WordPress rely on many dynamic features, the engine relies on sound but conservative approximations that may lead to spurious paths. In turn, spurious paths may lead to the identification of spurious protection-impacting changes. As a consequence, the real set of protection-impacting changes may be even smaller than reported. We did not calculate the spurious path rate since a previous study reported the spurious path rate for PTFA on WordPress of 10.96 ± 3.18% (95% confidence level) [101]. Our results also depend on the differencing algorithm we used. We extracted line-level differences between releases using GNU `diff`. Since there may be many CFG vertices (and thus PTFA states) on the same line of code, we over-estimate *addedEdges* and *deletedEdges*. However, this should only affect our results minimally, since we present our results at a line granularity. Other differencing algorithms, such as those supplied by versioning systems could also be used and may produce slightly different results.

Developers sometimes rely on variables and dynamic expressions to determine which privilege to verify. Our tool also conservatively approximates privilege names verified, and we may report definite protection differences where none occurred. Practically, the real rate of definite protection differences may be to be lower than we report. To counter this, we would like to improve our tool's accuracy by leveraging string analysis or use runtime invariants.

Our evaluation depends in part on the method we chose for reverting protection-impacting changes. We revert at file granularity, which is coarse. Reverting changes at a finer granularity (e.g. function level) could also be used and may produce slightly different results. **Threats to external validity** relate to the generalizability of our results. We did not have a vulnerability oracle for our study, though our approach may use one. Consequently, we cannot determine a specific distribution of protection-impacting changes for vulnerabilities. To counter this issue, studies using a vulnerability oracle (e.g. a testbench with known vulnerabilities) should be performed.

Another threat to generalizability is that our study is on two open source content management system implemented in PHP. We may obtain different results when studying other systems, whether they are written in PHP or other languages. While the PTFA engine we used only handles PHP at the moment, our approach itself is reproducible and language-independent. To avoid that our conclusions depend on the change history of this single system, studies that include other systems should be performed.

7.9 Conclusion and Future Work

Our general objective is to reduce RBAC re-validation effort. We proposed an automated static analysis over privilege protection.

Our analysis finds definite protection differences and determine the *protection-impacting changes* that caused a subset of them. Protection-impacting changes are a subset of the code changes and contains those DPD root causes whose code changes belong to paths.

We detailed three main contributions. First, we described a method to automatically identify *protection-impacting changes* during software evolution. Second, we reported on a longitudinal survey of the prevalence of protection-impacting changes in two popular PHP web-based content management systems. This survey considered 210 release pairs of WordPress, and 192 release pairs of Mediawiki. Third, we reported on a longitudinal survey of the prevalence of DPD root causes whose code changes belong to paths in WordPress and MediaWiki.

We have seen that 87/210 (41%) WordPress release pairs and 42/192 (22%) MediaWiki release pairs were security-impacted. For these release pairs, protection-impacting changes are a small percentage of code changes, with a median of about 27% for WordPress and 14% for MediaWiki. Over all release pairs in our survey, protection-impacting changes are an even smaller percentage of code changes. While the median is not meaningful, their mean is about 11% and 3% of code changes respectively for WordPress and MediaWiki.

By reverting protection-impacting changes, we found that DPD root causes whose code changes belong to paths explain all definite protection differences in 87% of WordPress release pairs and 93% of MediaWiki release pairs. Our first research question was "*Is the size of protection-impacting changes smaller than the set of code changes?*". We conclude affirmatively. Using protection-impacting changes, developers would only need to validate between a median of about 14% to 27% of code changes in security-impacted release pairs. Our conclusion stands despite outliers pushing the average up somewhat (about 15%) to 26%.

RQ1 *Is the size of protection-impacting changes smaller than the set of code changes?*
Using protection-impacting changes, developers would only need to validate between a median of approximatively 14-27% and a mean of about 15-26% of code changes in security-impacted release pairs.

Our second research question was "*How often PICs contain all DPD root causes in a release pair?*". We conclude that the vast majority of release pairs meet this criteria. Using two reversal approaches, we confirmed that DPD root causes whose code changes belong to paths explained all definite protection differences in 87% to 93% of cases.

Our results indicate that developers are likely to benefit from using protection-impacting change analysis. They shrink the amount of code changes that must be evaluated up to 27% and could be used in 87% to 93% of the times.

Future research includes the investigation of the distribution of *protection-impacting changes* corresponding to vulnerabilities, whenever a security oracle is available. This evaluation would allow us to quantify how many protection-impacting changes caused or fixed vulnerabilities. Further research could also be devoted to investigating the interaction with developers and the visualization [180] of *protection-impacting changes*. An approach about determining explanatory counter-examples [101] could also be used, combined with visualization techniques to supply information to developers. As mentioned earlier, further research may include the assessment of the real impact of the proposed techniques to application security verification and validation in collaboration with real human developers. Such studies would measure the developers validation effort during an application evolution and assess the advantages of the proposed approach. Finally, we would also like to extend the proposed analysis to other systems and other programming languages.

CHAPTER 8 GENERAL DISCUSSION

Through five research objectives, our project found ways to support developers re-validating their RBAC implementation.

We presented a novel concept, evolutive analysis of RBAC implementations. This approach assumes that the code is just as valid as it was in the previous version when no definite protection differences are detected.

Earlier contributions compared stand-alone RBAC policies – potentially, but not explicitly, of different versions. However, these approaches were not suited for re-validating RBAC implementations. Other earlier contributions analyzed the RBAC implementation of one version. Despite these limitations, non-evolutive RBAC analyses may help developers for the initial validation of their RBAC implementation.

We do not repeat our earlier discussions. However, we summarize important points in relation to our research objectives in Section 8.1. Then, we discuss the use of vulnerability oracles in Section 8.2. Afterwards, we consider the industrial applications of our method in Section 8.3, and its limitations in Section 8.4. We then show a few examples from security vulnerabilities in Section 8.5. Then, we discuss the implementation of our toolchain, followed with reflection on language support in Sections 8.6 and 8.7, respectively.

8.1 Discussion on the Research Objectives

Our first research objective was "*Determine which parts of the program have different definite privilege protection, compared to the last version*". To achieve this objective, we defined definite protection differences. In our survey of WordPress, we found that code changes between releases have a small impact on definite privilege protection. When a pair of versions is free of DPDs, there are no regressions in the RBAC implementation, within the limits of our analysis. When a pair of versions has security-impacted code, our analysis is likely to focus developers' efforts, as 99.7% of code is free of DPDs.

Our second research objective was "*Determine how the protection differs*". We proposed a set-theoretic classification over definite privilege protection sets. We designed our classification to distinguish between different cases of overlap because there might be association between privileges in symmetric overlaps. Definiting this kind of category up front supports potential research efforts in determining these associations.

In our survey of WordPress, we found that complete gains, complete losses and substitution were the most common categories. The classification of DPDs may help developers understand the nature of the changes and their security impacts. Additionally, the dominant classification categories are very black-and-white. In the first main categories, definite protection for a statement changes from unprotected to protected (or *vice versa*). In the next main category, a statement's definite protection changes entirely from one privilege to another. This kind of straightforward information may be easier to reason about.

Our classification may also help them prioritize review efforts. For instance, depending on the application's security policy, a complete loss of privilege protection may be more troublesome than a monotonic gain. In such a case, developers are likely to investigate the former first.

Our third research objective was "*Given a change in definite protection, justify the outcome.*" We achieved this objective by computing explanatory counter-examples for loss-affected code. In order to facilitate the evaluation of explanatory counter-examples by developers, we additionally used graph transformations to shorten them. We found that most privilege protection loss examples are fewer than 100 states long, which is short. We also note that most examples are contained in the same file, but are not contained in the same function.

We expect that privilege protection loss examples will help developers revalidate their RBAC implementation. The counter-example is an unprotected path that is responsible for a loss of protection. There may be other paths that also explain the DPD. However, due to the universally quantified predicates of definite privilege protection, the existence of a single unprotected path is sufficient for definite privilege protection to be lost. The path presented should be straightforward for developers to evaluate. It also hints at corrective actions. Since this was a preliminary study, we restricted our approach to loss-affected cases. However, the approach is generalizable and future research should be dedicated to it.

Our fourth research objective was "*Determine which code changes are responsible for protection differences.*" To do so, we computed protection-impacting changes. In our survey of WordPress and MediaWiki, we found that, in security-affected release pairs, a median of approximately 14% to 27% of code changes are protection-impacting.

When definite protection differences are detected, developers can focus their review efforts on the code changes that are protection-impacting.

Our last research question was "*Investigate the relationship between the code changes identified in **RO4** and root causes of protection differences.*" Using reversal, we also determined that PICs contain all root causes of DPDs in 87% to 93% of release pairs. Since this is a preliminary study, we chose to revert DPD root causes at file granularity because the reverted changes are guaranteed to be syntactically correct and the simplicy of the operation.

Within the limitations we discuss below, these findings guarantee that reviewing PICs are sufficient to determine the origin of DPDs. In other words, when re-validating their RBAC implementation, developers only need to review a median of 27% of their code changes, in at least 87% of the times.

We also reported the successful reversal of protection-impacting changes individually for each definite protection differences in the latest security-affected release pair of each system. These reversal results indicate that our approach is suited for analyses of varying degrees of granularity. In particular, this result demonstrates the viability of on-demand analysis for specifics DPDs.

We focused on DPD root causes whose code changes belong to paths, but also observed definite protection differences caused by changes that occur somewhere else. For instance, code changes that affect interprocedural edge resolution may cause definite protection differences. In addition to the kind of impact due to class hierarchy changes and reflective calls, we also found definite protection differences caused by conservative approximations in our PTFA engine. For instance, it treats apparently dead code as unprotected entry points. This decision has the consequence of injecting unprotected paths that do not directly intersect with the code changes.

8.2 Vulnerability Oracles

Our approach does not require an oracle that classifies definite protection differences as beneficial, harmful or irrelevant. However, such an oracle would be beneficial. Using it, we could compute explanatory counter-examples and protection-impacting changes only for harmful security changes. In addition, this oracle, combined with PICs, would allow us to report how many code changes were harmful. Finally, we would be able to report on the differences of populations for DPDs, explanatory counter-examples and PICs.

8.3 Secure Software Development Processes

Our contributions could be integrated in software development processes with relative ease.

One can compute DPDs between any two versions of the application. Consequently, a developer could also compute explanatory counter-examples and PICs at any granularity. Our approach is therefore suitable as an early warning system for security issues. For instance, developers could detect DPDs after each commit. They could also do so in automated build systems. A build system could treat definite protection differences as regressions and calculate protection-impacting changes whenever such regressions occur.

Adding our approach to integrated development environments (IDE) would also be very helpful. This would allow developers to compute DPDs, explanatory counter-examples and PICs interactively.

Vulnerability reports may be easier to evaluate using on-demand analysis. For instance, they could compute protection-impacting changes for the vulnerable code segment. Then, they may determine which code changes are responsible for the vulnerability and which correction is needed. Thus, our approach may also facilitate the creation of vulnerability patches. Additionally, our approach may validate security patches, by comparing the effective definite protection differences against developers' expectations.

In addition, we earlier suggested a decision support system based on our approach. This system would automatically determine if the DPD root causes are changed path root causes. Whenever the DPD root causes are not changed path root causes, this decision support system would pinpoint edges requiring deeper analysis.

8.4 Limitations of the Proposed Solution

The precision of our analysis is limited by the precision of its inputs, namely of the PTFA model and the code differences, which we elaborate on in Section 8.4.1. Furthermore, the results we report are dependent on the program size measure we use, which we discuss in Section 8.4.2. In addition, our approach is limited in the scope of code we analyze, which we describe in Section 8.4.3. Finally, our approach is limited by code patterns developers use. We identified them during our reviews and describe them in Section 8.4.4.

8.4.1 Precision

The PTFA analyzer used throughout this thesis uses conservative approximations of many PHP features, including polymorphism and reflection [60]. The algorithm we use for backwards reachability (c.f. Appendix F) is sound, but may spuriously consider edges in recursive functions as reachable. This seems not to be a problem in practice, however. If this becomes problematic, it may be possible to compute all paths in the problematic region. Since this analysis is likely to be expensive, future research should consider this, possibly using a context-sensitive analysis. Nonetheless, program analysis in presence of recursion suffers from some undecidability and uncomputability constraints [96]. Despite these constraints, it is an active area of research (e.g. [17, 42, 110]). Future tools may take advantage of such improvements, since the definition of protection-impacting changes is independent of the interprocedural reachability algorithm used.

The code differencing approach we use is based on line-level differences. This reflects the industry standard, reflected by a widespread use of `diff`. In addition, version controls systems such as `git` use line-level differences by default, even though a finer-grained option is available (i.e. `--word-diff`). Even though line-level differencing works well in practice, we may obtain better results using finer grained differencing (e.g. AST differencing [47, 52] or clone analysis).

Another limitation on the precision of our results is based on the conservative definition of PICs, which report all code changes that are on appropriately protected interprocedural paths to DPDs. Therefore, PIC analysis may identify code changes as protection-impacting, even though they have no apparent impact to the privilege protection (e.g. many libraries and included code that do not affect privilege protection). Future research should address this limitation.

8.4.2 Program Size and Change Size Metrics

In the first article (Chapter 4), we reported a relative measure that was the number of DPDs over the program size in terms of CFG vertices. This provides an apples-to-apples comparison, since DPDs are defined over CFG vertices. In the second article (Chapter 5), we reported counter-example length in terms of PTFA states.

Such measures may be counter-intuitive for developers. As such, we reported Lines of Code (LOC) metrics in the third article (Chapter 6). For the sake of completeness, we reported both PTFA-level and line-level metrics in the last article (Chapter 7).

LOC is a simple measure that is strongly correlated with other size measures and many metrics [63, 117]. As such, a relationship between a variable and LOC should also hold for other program size metrics. In addition, metrics that are strongly correlated with program size are more able to predict external features such as failures [63]. Gil and Lalouche even argue that “code size is the only ‘unique’ valid metric.” [63].

The Number of Tokens (NOT) is considered to be a more robust metric [63] than LOC. However, the difference in correlation between LOC and other size metrics (such as the number of tokens [NOT]) is not statistically significant. In addition, LOC is more reproducible. This factor is relevant when analyzing programs written in programming languages which lack a formal specification of their grammar, such as PHP. Front-ends may define different tokens, yielding variations in NOT measures. We expect that these variations may be minimal. Nonetheless, we prefer a more reproducible metric and thus opt for LOC.

A measure of program size that excludes comments and empty lines is the Non-Comment Lines of Code (NCLOC) [54]. The use of NCLOC is however generally discouraged. Since PICs do not include comments and lines of code, it may nonetheless be more appropriate to report relative measures (e.g. PIC_{lines}) using changed NCLOC instead of changed LOC. However, obtaining changed NCLOC is likely to require post-processing over the output of differencing tools. As such, this should be considered in future research. Nonetheless, we expect that the ratios we report would not vary greatly. In our observations, the code changes were dominated by non-comment and non-whitespace lines.

8.4.3 Privilege Protection and Code Analyzed

Our approach only considers definite protection. It is possible that our approach is useful for possible privilege protection as well.

Also, our approach is that it depends on the differences between versions Ver_a and Ver_b. As such, the choice of these versions is very important.

In addition, DPDs are, by definition, only available for code that belongs to both versions. Intuitively speaking, the privilege protection of new and deleted code may be important to developers. For instance, they may be interested in the code changes that caused definite protection (or lack thereof) in new code. Currently, developers only have definite protection information for new and added code. Protection-impacting changes should be extended to answer this need.

8.4.4 Complex Code Patterns

Along my predecessors [60], I observed some RBAC patterns in the applications examined. The use of these patterns may be problematic, as they do not confer definite protection, and thus no DPDs. These patterns are *qualified privilege protection checks*, *disjunctive checks*, and *variable protection checks*.

Qualified privilege protection checks depend on other values. Such security checks create security invariants that developers rely on. We show an example in Figure 8.1a, where the security of a block relies on the earlier qualified verification of privilege `p1`.

The other two patterns are similar. Variable protection checks verify the privilege stored in a variable, which may have many possible values. We show an example in Figure 8.1b, where either privilege p1 or p2 is verified. Disjunctive checks, in a similar vein, verify that at least one privilege is verified. We show an example in Figure 8.1c, where privilege p1 or p2 is verified, but where it is possible that the user has both. PTFA analysis currently only tracks constant and conjunctive predicates.

```
if(('val' == $status)
   &&!isAllowed('p1'))
  return ERROR;
//...
if('val' == $status){
// Protected by p1
}
```

(a) Example of Qualified Privilege Check

```
if (...) $cap = 'p1';
else $cap = 'p2';

if (isAllowed($cap))
  die();
//Protected by p1 OR p2
```

(b) Example of Variable Privilege Check

```
if (!isAllowed('p1') &&
    !isAllowed('p2'))
  die();
//Protected by p1 OR p2
```

(c) Example of Disjunctive Privilege Check

Figure 8.1 Example of Privilege Checks Which Do Not Confer Definite Privilege Protection

These observations open many research directions. First, what are all the patterns of RBAC checks in RBAC implementations, and what is their demography? Second, which ones are RBAC implementation anti-patterns? Third, if they are anti-patterns, how can we automatically transform the code that use RBAC implementation anti-patterns into a form that only uses good patterns? Fourth, how can we determine definite protection in the presence of RBAC implementation anti-patterns? Fifth, how can we *efficiently* do so?

Regarding the fifth question, variations on PTFA are possible for disjunctive predicate checks. A first option would be to consider them as a new privilege (e.g. $priv_1 \lor priv_2$). Another option is to record grant edges in these cases as being possible, rather than certain, as was proposed by Ouellet et al. [141]. This latter option would require expanding our approach for non-definitive privilege protection.

8.5 Examples

We now consider three examples from vulnerability fixes. We represent the patches in Unified Diff format and we highlight the changed lines of code that our tool identifies as protection-impacting in green.

In our first example, we look again at CVE-2015-5623, which we first discussed in Section 1.1.5. In Figure 8.2, we observe that the functional change in line 26 is not protection-impacting. It is important to note that abrupt termination statements (in this example, return and exit) are not always protection-impacting. This is because these statements create an edge that connects directly to the end of the function. As such, they may not always belong to the appropriately protected paths between q_0 and DPDs.

```
 1     // In wp-admin/includes/dashboard.php
 2     wp_network_dashboard_right_now() {
 3     function wp_dashboard_quick_press( $error_msg = false ) {
 4     global $post_ID;
 5
 6   +  if ( ! current_user_can( 'edit_posts')){
 7   +     return;
 8   +  }
 9     // ...
10     //In wp-admin/post.php
11     if ( ! wp_verify_nonce( $nonce, 'add-post' ) )
12       $error_msg = __( '...' );
13
14   - if ( ! current_user_can( 'edit_posts'))
15   + if ( ! current_user_can( 'edit_posts')){
16   -   $error_msg = __( '...');
17   +     exit;
18   + }
19     //...
20     //In wp-includes/capabilities.php
21     case 'edit_post':
22     case 'edit_page':
23     $post = get_post( $args[0] );
24   - if ( empty( $post ) )
25   + if ( empty( $post ) ) {
26   +     $caps[] = 'do_not_allow';
27     break;
28   + }
29     //...
```

Figure 8.2 Reformatted Patch for CVE-2015-5623. Represented in Unified Diff Format. Protection-Impacting Lines of Code Highlighted in Green.

Our second example involves a complex code pattern. The patch to the vulnerability documented by CVE-2006-6016 (Figure 8.3). This vulnerability affected WordPress in releases prior to 2.0.5 and allowed remote authenticated users to access any user's metadata.

We observe that the vulnerability was caused by improper handling of a lack of privileges. The developers only recorded error messages to display, but did not prevent the information to be retrieved and displayed (lines 14 and 21). The patch resolves the issue with calls to `die`, which stops the script's execution (lines 15 and 23). The code elided corresponding to line 18 is definitely protected with regards to privilege `edit_users` after the patch is applied.

In addition, the developer added a parameter validity check (lines 1 to 5). This change does not appear related to the vulnerability and is plausibly an act of defensive programming.

In this case, the abrupt termination statements (`die`) are never protection-impacting. This occurs because these statements create an edge that connects directly to the end of the program execution. As such, they never belong to an appropriately protected paths between q_0 and DPDs.

```
 1  + $user_id = (int)$user_id;
 2  +
 3  + if ( $user_id )
 4  +       die(__('Invalid␣user␣ID.'));
 5  +
 6    switch ($action) {
 7    case 'switchposts':
 8      check_admin_referer();
 9      /* TODO: Switch all posts from one user to another user */
10    break;
11    case 'update':
12      //...
13      if (!current_user_can('edit_users'))
14  -     $errors['head']=__('...');
15  +     die(__('...'));
16      else
17        $errors = edit_user($user_id);
18      //...
19    default:
20    //...
21  -   if (!current_user_can('edit_users')) $errors['head'] = __('...');
22  +   if (!current_user_can('edit_users'))
23  +     die__('...');
24    ?>
```

Figure 8.3 Reformatted Security Patch for CVE-2006-6016. Changes represented in Unified Diff format. Protection-Impacting Lines in Green.

One of the branches of the switch statement (line 7) is a placeholder for a future feature. The consequence of this code pattern is that the code after the switch statement is not definitely protected, and thus we would not detect DPDs there. In practice, the code after the patch is definitely protected with privilege `edit_users` as long as the `$action` variable contains a string other than `'switchposts'`. This may be an invariant in that version. Because little code is gain-affected in this example, the change shown at lines 21 and 23 is not considered protection-impacting.

Our third example also includes complex code patterns. It is the patch fixing the vulnerability documented in CVE-2007-1893 (Figure 8.4). This vulnerability in WordPress versions prior to 2.1.3 allowed remote a privilege escalation. Users with the contributor role were able to publish a post, something that was restricted to more privileged users.

All the privilege checks in the patch are either disjunctive or qualified. In addition, the unqualified privilege check at line 9 was moved and made qualified at line 16. As a result, the code at lines 12 onwards are loss-affected.

```
 1  + if ( (’publish’ == $post_status) && !current_user_can(’publish_posts
        ’) )
 2  +    return new IXR_Error(401, ’Sorry,␣you␣do␣not␣have␣the␣right␣to␣
        publish␣this␣post.’);
 3  +
 4    $post_title = xmlrpc_getposttitle($content);
 5    $post_category = xmlrpc_getpostcategory($content);
 6    $post_content = xmlrpc_removepostdata($content);
 7  //...
 8    set_current_user(0, $user_login);
 9  - if ( !current_user_can(’edit_post’, $post_ID))
10  -    return new IXR_Error(401, ’...’);
11  -
12  - $postdata = wp_get_single_post($post_ID, ARRAY_A);
13  - extract($postdata);
14  - $this->escape($postdata);
15  //...
16  + if ( ( ’post’== $post_type )&& !current_user_can(’edit_post’, $post_ID))
17  +    return new IXR_Error(401, ’...’);
18  +
19  +  $postdata = wp_get_single_post($post_ID, ARRAY_A);
20  +  extract($postdata);
21  +  $this->escape($postdata);
22  //...
23  + if ( (’publish’== $post_status)){
24  +   if ( ( ’page’== $post_type )&& !current_user_can(’publish_pages’))
25  +      return new IXR_Error(401, ’...’);
26  +   else if ( !current_user_can(’publish_posts’))
27  +      return new IXR_Error(401, ’...’);
28  +          }
29    //...
30  }
```

Figure 8.4 Reformatted Security Patch for CVE-2007-1893. Changes represented in Unified Diff format. Protection-Impacting Lines in Green.

8.6 Implementation Notes

We summarize the software components described in this thesis in Figure 8.5. The component in light grey (the PHP front-end) was originally developed by fellow lab members and maintained by myself. It was partly described in the literature [60, 61] and we provide more details in Section 8.6.1. The components in dark grey have been implemented by fellow lab members and used as-is. One of them (the PTFA engine) was described in the literature [57, 60] and we provide more details in Sections 8.6.2 and 8.6.3.

What I implemented, the white components, use output of the PTFA analyzer and program differences to perform additional analyses. Please note that Figure 8.5 does not include all utilities and scripts that facilitate automation in order to avoid overcrowding. The new Java source code, without tests, adds to 23 KLOC. The python scripts add up to 4 KLOC.

The components follow the pipes-and-filters architecture, in line with the Unix philosophy. Thus, each component has a textual output format and dependent components parse them. This architecture may not offer the best performance, but it facilitates troubleshooting, reuse of experimental data and enables future research projects.

8.6.1 The PHP Front-End

The PHP front-end is based on an open-source grammar[1] and maintained by Thierry Lavoie, François Gauthier and eventually myself. The front-end also includes visitors that output ASTs and CFGs. It also integrates with a Datalog engine for propagating access control checks stored in variables [60, 61].

The grammar handles all PHP versions between 4.0 and 5.6. Please note that the majority of PHP 7 code is handled. Because the PHP grammar is not formally specified and fluctuates from one version to the other, our grammar supports a superset of the actual PHP 5.6 language. In other words, it would not be possible to use a recent version of the official PHP interpreter as a front-end, as old code sometimes violate the current grammar and is thus rejected by the PHP interpreter.

My maintenance duties mostly consisted at updating the grammar when we found edge cases that caused compilation errors. These fixes were non-trivial, as it often required a lot of trial-and-error to reverse-engineer the grammar rule that the PHP interpreter expects. This sometimes identified a production rule based on an erroneous prior reverse-engineering. In all cases, it was challenging to modify the grammar without regressions.

[1]The original grammar is credited to *Satyam*, but their full name is unknown. There is no apparent relationship with the IT company Mahindra Satyam, formerly known as Satyam Computer Services.

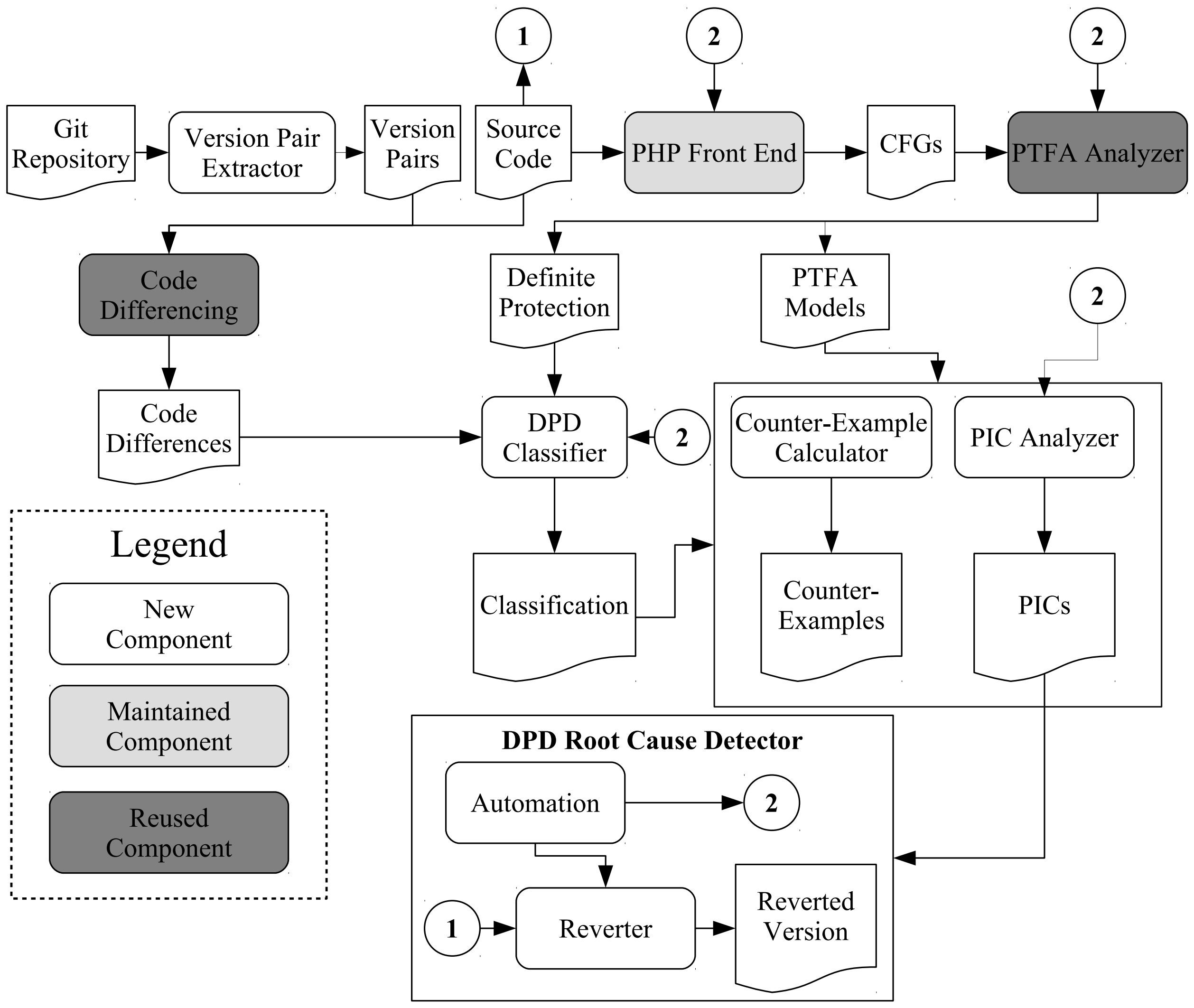


Figure 8.5 Contributed Software Ecosystem

8.6.2 The ACMA PTFA Analyzer

François Gauthier developed the PTFA engine we used, named ACMA [57]. Mathieu Mérineau maintained it and added support tools. ACMA is described in more details in Gauthier's PhD thesis [60]. While I have not directly modified ACMA, I added a processing step over its output files. ACMA outputs reachable PTFA models in a verbose XML format. Consequently, the output of a single analyzed version requires many gigabytes of storage, up to about 200 Gb for a recent version of MediaWiki. Out of necessity, I investigated multiple compression formats (e.g. Zip, BZip2, XZ) and settled on the Efficient XML Interchange (EXI) format [178]. This XML-specific format provides a higher compression ratio than the other formats for our data set.

8.6.3 Line Differences

The code differencing tool, `mapdiff`, is a Python script developped by Ettore Merlo. This script extracts line mapping information from `diff`'s output. I created a driver program to interface `diff` and `mapdiff` and facilitate their use for multiple pairs of versions. Please note that we use `diff` with the `--ignore-space-change` and `--minimal` flags. These flags allow to match corresponding lines even if indentation changed, and generally provides a better mapping between versions.

8.7 Language Support

Multiple factors explain why supporting other programming languages requires significant development effort. Although the concept of CFG is common, its representation and data schema varies a lot across different tools [136].

First, the architecture of ACMA integrated call graph construction and PTFA analysis in one step. This design decision made ACMA PHP-specific, although PTFA is a language-independent approach. This factor, combined with scalability issues uncovered with recent versions of Moodle, lead to the decision to implement a new PTFA engine. This new PTFA engine is currently under development and offers better language-independence.

Second, PTFA analysis requires a CFG with *grant* edges, a non-standard edge type. As such, the output of a CFG extractor could not be used as-is, and a post-processing tool would be required to detect access control verifications and modify edges accordingly.

Third, the computation of a call graph is subject of various reasonable interpretations and that statically computing a precise call graph is undecidable [136]. The call graph is the graph which contains the interprocedural edges in the interprocedural CFG. In other words, different static analysis tools tend to produce different interprocedural CFGs. Since the tools' design decisions affects the CFG and thus the PTFA analysis, a very careful evaluation is required before choosing and integrating one.

Finally, a lack of portability in the front-ends' output mean that no universal importer is practically feasible. The Graph eXchange Language [70] and ProBe [115] data formats are intended to overcome this issue, but their adoption has been a limited success so far.

CHAPTER 9 CONCLUSION AND RECOMMENDATIONS

In our concluding remarks, we summarize our contributions, succinctly state how this thesis advanced knowledge, and discuss future research.

9.1 Summary of Contributions

We have seen that RBAC is a widespread form of access control, and an important part of web application security. We have also seen that validating and re-validating RBAC policy implementation is a difficult process. Every maintenance activity, whether related to security or not, potentially adds both intentional and unintentional security flaws. As such, developers need to revalidate their application regularly. Failure to do so may result in access control vulnerabilities whose impact may range from information leaks (possibly to unauthorized parties) to privilege escalation allowing all users to use administrative functionalities.

To address this situation, we presented the following thesis:

Thesis
Using static program analysis, we can automatically (1) identify differences in definite privilege protection between two versions of an application, (2) classify these differences in explanatory categories, (3) compute explanatory counter-examples justifying the differences, (4) determine which code changes are responsible for these differences and (5) conservatively identify all root causes for these differences in most cases.

We demonstrated this thesis with new definitions, program analyses, and empirical results. Specifically, we examined five research objectives.

Our first research objective was "*Determine which parts of the program have different definite privilege protection, compared to the last version.*" To fulfill this objective we defined definite protection differences (DPDs). This method compares two versions of a program and identifies a partial one-to-one mapping between their CFG vertices. Using this information and the vertices definite protection, we compare definite protection and determine which vertices are security-affected. When surveying 147 release pairs of WordPress, we found that, when a release pair is security-affected, only 0.30% of the code is security-affected on average. In our review of the results, we found that roughly 70% of DPDs may violate security assumptions.

Our second research objective was "*Determine how the protection differs.*" To fulfill this objective, we defined a set-theoretic classification of DPDs. We proposed 8 categories: complete loss, complete gain, monotonic loss, monotonic gain, substitution, symmetric overlap, asymmetric overlap and disjoint. To do so, we compared definite protection similarly as **RO1**. However, instead of performing a per-privilege comparison, we compared definite protection sets. When surveying 147 release pairs of WordPress, we found that the most prevalent categories were complete gains ($\approx 41\%$), complete loss ($\approx 18\%$) and substitution ($\approx 20\%$). Doing a manual review of all $\approx 34k$ classifications, we found that our classification is about 82% accurate.

Our third research objective was "*Given a change in definite protection, justify the outcome.*" To achieve this objective, we computed privilege protection loss examples for them. This computation involves finding the shortest interprocedural path in PTFA models. In order to keep the explanatory counter-example short, we proposed a graph transformation that summarizes interprocedural subpaths having no impact on definite protection. When surveying 147 release pairs of WordPress, we found that privilege protection loss examples have median length of 88 PTFA states. We also found that explanatory counter-examples are fairly local – the median explanatory counter-example is in a single file and crosses one function boundary. Our review has shown a spurious path rate of about $11 \pm 3\%$ at a 95% confidence level.

Our fourth research objective was "*Determine which code changes are responsible for protection differences.*" To do so, we defined protection-impacting changes (PICs). To compute PICs, we compare two versions of the program as in **RO1**. Using a mapping between CFG vertices, we determined added and deleted edges between PTFA models. We also compute edge reachability of appropriately protected paths. Combining this information, we obtain PICs. We also surveyed 210 release pairs of WordPress and 192 release pairs of MediaWiki. We found that 87/210 (41%) and 42/192 (22%) release pairs, respectively, were security-affected. Their respective relative median PICs is about 27% and 14% of code changes in security-affected release pairs. Over all release pairs, about 91% and 97% of their respective source code changes are not protection-impacting.

Our fifth research objective was "*Investigate the relationship between the code changes identified in **RO4** and root causes of protection differences*". To do so, we reverted all PICs in security-affected release pairs at a file granularity. We found that 87% to 93% of release pairs had no definite protection differences (DPDs) left after reversal.

Our results indicate that developers are likely to benefit from using our analyses. They shrink the number of code changes that must be evaluated up to 27% and contain all root causes in 87% to 93% of the cases. In the event that protection-impacting changes do not contain the root causes, developers can still use definite protection differences and protection losses examples to understand why a DPD occurs.

9.2 Advancement to Knowledge

In this thesis, we introduced a novel concept, an evolutive analysis of RBAC implementations. We defined many novel analyses: definite protection differences (DPDs), protection-impacting changes (PICs), a classification of DPDs and a method to determine if PICs contains root causes using reversal. Doing so required creating a series of processing steps and algorithms.

In addition, our surveys of two open-source applications brought novel findings. We discovered how often definite protection changes between versions, how much of the code is security-affected, how short could explanatory examples get, and how much of the code changes are protection-impacting. In addition, we discovered one category of root causes of DPDs, root causes whose code changes belongs to paths. Furthermore, we discovered that PICs contained these root causes in the vast majority of cases.

9.3 Recommendations for Developers

Our surveys were on open-source systems with no explicit security policy. As such, we could not tell if a difference in protection violated the policy. This practice is likely to complicate RBAC implementation validation. An explicit policy would remove confusion at that level, especially if the policy is specified with semi-formal or formal models.

While reviewing our results, we often observed that developers would verify privileges at the beginning of the script and abort execution early. While this is likely to improve performance, it causes a large portion of the script to be definitely protected. In addition, this definite protection is carried over to all called functions. This code pattern seems to go against the principle of least privilege, as parts of the program are executing with more privileges than strictly necessary. We recommend that developers only verify privileges for minimal code sections that need them.

We also observed many qualified, variable and disjunctive checks. The use of these code patterns diminish the efficacy of our approach. Using only unqualified and constant privilege checks would facilitate the use of our tools and possibly facilitate re-validations *ipso facto*.

Developers should compute and verify PICs often during the project. This would avoid large revalidation efforts before release. This recommendation is especially important for large new releases (i.e. .0 releases). Even though the rate of protection-impacting changes for these releases is relatively small (21% to 39%), it remains that the absolute amount of protection-impacting changes (> 10,000) is large. The regular use of our approach during the development of .0 releases would be especially desirable.

9.4 Future Research

Our approach is promising. In addition to the future research related to the limitations we mentioned earlier, there are many research opportunities to leverage protection-impacting changes in other fields.

The concept of protection-impacting change is analogous to that of the fault localization in automatic software repair (ASR). Fault localization [65, 87] identifies code statements that introduces bugs. In other words, it determines the core piece of code that introduced a fault. This important step in ASR results in a smaller search space, which accelerates and increases the likelihood of successful fault repair. It would be interesting to investigate automatic software repair for protection-impacting changes in future research.

We surveyed applications at the granularity of release pairs. However, our approach can work between any two versions. In addition, our methods may be combined using software repository mining approaches.

Our approach may be suited for selecting regression tests. We could identify a subset of the tests that execute security-affected code and, when PICs contains all root causes, restrict the tests to execute based on PICs.

Our approach may be suited for change impact analysis. Given a proposed software changes, we can determine if any DPD would occur.

Our approach centers on RBAC, but it may be applicable to other forms of access control as well. In addition, it may also be suited to verify applications that rely on a formal access control policy such as XACML. This is because PTFA only depends on the predicates that the front-end can identify. The moment an application uses stereotypical code patterns for authorization, there is a good chance that PTFA can analyze it. Furthermore, PTFA could be extended to consider the role hierarchies and constraints between roles that belong to more complex forms or RBAC.

Our approach has been developed in isolation from developers in the industry, though it has been guided by our industrial experience. As such, case studies of developers using DPDs, DPD classification, explanatory counter-examples and PICs should be performed. To obtain a better response, user-friendly tool support (e.g. visualization, IDE integration) should be implemented prior to these studies.

We computed the association between CFG vertices between versions (i.e. the *vertexMap* and *bMap* functions) using line-level differences. Differencing code at the line level is a common practice in the industry, as evidenced by the `diff` and `git` utilities. However, it would be preferable to use a finer-grained approach. We could do so using AST differencing [47, 52], graph matching [36] or clone analysis [104, 105, 155].

We have made a first contribution towards the identification of the root causes of DPDs. Nonetheless, much research is needed to precisely identify root causes.

The PTFA engine should be improved to support complex the code patterns we identified in Section 8.4.4. In addition, support for additional languages should be added. One interesting avenue of research would be the handling of aspect-oriented languages in the construction of PTFA models.

Another interesting avenue of research is integrating dynamic analysis. This could be done by detecting runtime invariants (e.g. by extending Daikon [50] to PHP). Another option would be to record dynamic call graphs. It would be possible to use the dynamic call graphs as-is, but at the risk of introducing false negatives. Alternatively, one could use dynamic call graph data to rank the DPDs and PICs according to their frequency or execution, similarly to PatchAdvisor [139]. This latter approach would especially be relevant for industrial studies.

Finally, protection-impacting changes could be adapted for similar problems. PICs is ultimately a general framework that combines code changes with interprocedural graph reachability. Other problems that rely on graph reachability, such as taint analysis, are likely to be easily adapted to this framework.

REFERENCES

[1] A. Abadi, R. Ettinger, and Y. A. Feldman, "Fine slicing : Theory and applications for computation extraction," in *Proc. Int'l. Conf. Fundamental Approaches to Software Eng. (FASE 2012)*, 2012, pp. 471–485.

[2] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *ACM SIGPLAN Notices*, vol. 25, no. 6, pp. 246–256, Jun. 1990.

[3] A. Ahmad and B. Whitworth, "Access control taxonomy for social networks," in *Proc. 7th Int'l Conf. Information Assurance and Security (IAS 2011)*, 2011, pp. 256 – 261.

[4] G.-J. Ahn, H. Hu, J. Lee, and Y. Meng, "Representing and reasoning about web access control policies," in *Proc. 34th IEEE Annual Comput. Software and Appl. Conf. (COMPSAC 2010)*, 2010, pp. 137–146.

[5] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition).* Boston, MA, USA: Addison-Wesley, 2006, ch. 12.

[6] ——, *Compilers: Principles, Techniques, and Tools (2nd Edition).* Boston, MA, USA: Addison-Wesley, 2006, ch. 9.

[7] M. H. Alalfi, J. R. Cordy, and T. R. Dean, "Recovering role-based access control security models from dynamic web applications," in *Proc. Intl' Conf. Web Eng. (ICWE 2012)*, 2012, pp. 121–136.

[8] ——, "Automated verification of role-based access control security models recovered from dynamic web applications," in *Proc. 14th IEEE Int'l Symp. Web Systems Evolution (WSE 2012)*, 2012.

[9] M. H. Alalfi, E. P. Antony, and J. R. Cordy, "An approach to clone detection in sequence diagrams and its application to security analysis," *J. Software & Systems Modeling*, Sep. 2016.

[10] M. Alkhalaf, S. R. Choudhary, M. Fazzini, T. Bultan, A. Orso, and C. Kruegel, "ViewPoints: differential string analysis for discovering client- and server-side input validation inconsistencies," in *Proc. Int'l Symp. Software Testing and Analysis (ISSTA 2012)*, 2012, pp. 56–66.

[11] M. Alkhalaf, A. Aydin, and T. Bultan, “Semantic differential repair for input validation and sanitization,” in *Proc. Int’l Symp. Software Testing and Analysis (ISSTA 2014)*, 2014, pp. 225–236.

[12] M. Almorsy, J. Grundy, and A. S. Ibrahim, “Supporting automated vulnerability analysis using formalized vulnerability signatures,” in *Proc. 27th IEEE/ACM Int’l Conf. Automated Software Eng. (ASE 2012)*, 2012, pp. 100–109.

[13] B. Arkin, S. Stender, and G. McGraw, “Software penetration testing,” *IEEE Security & Privacy*, vol. 3, no. 1, pp. 84–87, Jan 2005.

[14] R. Arnold and S. Bohner, “Impact analysis - towards a framework for comparison,” in *Proc. Conf. Software Maintenance (CSM’93)*, 1993, pp. 292–301.

[15] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel, “FlowDroid : precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps,” in *Proc. 35th ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI 2014)*, 2013, pp. 259–269.

[16] S. Arzt, S. Rasthofer, R. Hahn, and E. Bodden, “Using targeted symbolic execution for reducing false-positives in dataflow analysis,” in *Proc. 4th ACM SIGPLAN Int’l Workshop State Of the Art in Program Analysis (SOAP 2015)*, 2015, pp. 1–6.

[17] I. Asăvoae, F. de Boer, M. Bonsangue, D. Lucanu, and J. Rot, “Model checking recursive programs interacting via the heap,” *Sci. Comput. Programming*, vol. 100, pp. 61–83, mar 2015.

[18] A. Austin and L. Williams, “One technique is not enough: A comparison of vulnerability discovery techniques,” in *Proc. Int’l Symp. Empirical Software Eng. and Measurement (ESEM ’11)*, 2011, pp. 97–106.

[19] D. Baca, B. Carlsson, and L. Lundberg, “Evaluating the cost reduction of static code analysis for software security,” in *Proc. 3rd ACM SIGPLAN Workshop Programming Languages and Analysis for Security (PLAS ’08)*, 2008, pp. 79–88.

[20] T. Ball, M. Naik, and S. K. Rajamani, “From symptom to cause: localizing errors in counterexample traces,” in *Proc. 30th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL ’03)*, 2003, pp. 97–105.

[21] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, “Saner: Composing static and dynamic analysis to validate sanitization in web

applications," in *Proc. IEEE Symp. Security and Privacy (SP 20018)*, 2008, pp. 387–401.

[22] J. Bell and G. Kaiser, "Phosphor: illuminating dynamic data flow in commodity JVMS," in *Proc. ACM Int'l Conf. Object Oriented Programming Systems Languages Appl. (OOPSLA 2014)*, 2014, pp. 83–101.

[23] M. Bishop, *Computer Security : Art and Science.* Upper Saddle River, NJ, USA: Addison-Wesley, 2003, ch. 15.

[24] ——, *Computer Security : Art and Science.* Upper Saddle River, NJ, USA: Addison-Wesley, 2003, ch. 2.

[25] ——, *Computer Security : Art and Science.* Upper Saddle River, NJ, USA: Addison-Wesley, 2003, ch. 4.

[26] ——, *Computer Security : Art and Science.* Upper Saddle River, NJ, USA: Addison-Wesley, 2003, ch. 7.

[27] P. Bisht, T. Hinrichs, N. Skrupsky, and V. N. Venkatakrishnan, "WAPTEC: whitebox analysis of web applications for parameter tampering exploit construction," in *Proc. 18th ACM Conf. Comput. Commun. Security (CCS 2011)*, 2011, pp. 575–586.

[28] I. Bocić and T. Bultan, "Finding access control bugs in web applications with CanCheck," in *Proc. 31st IEEE/ACM Int'l Conf. Automated Software Eng. (ASE 2016)*, 2016, pp. 155–166.

[29] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *Proc. 33rd Int'l Conf. Software Eng. (ICSE 2011)*, 2011, pp. 241–250.

[30] R. Boren. Changeset 15097. [Online]. Available: https://core.trac.wordpress.org/changeset/15097 (Accessed 7 Dec 2017).

[31] T. Bourdier, H. Cirstea, M. Jaume, and H. Kirchner, "Formal specification and validation of security policies," in *Proc. Int'l Symp. Foundations and Practice of Security (FPS 2012)*, 2012, pp. 148–163.

[32] R. Bragg, *CISSP Training Guide.* Indianapolis, IN, USA: Pearson IT Certification, 2002, pp. 130–131.

[33] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, “Towards a taxonomy of software change,” *J. Software Maintenance and Evolution : Research and Practice*, vol. 17, no. 5, pp. 309–332, 2005.

[34] D. Cappelli, A. Moore, and R. Trzeciak, *The CERT Guide to Insider Threats: How to Prevent, Detect, and Respond to Information Technology Crimes (Theft, Sabotage, Fraud)*. Upper Saddle River, NJ, USA: Addison-Wesley Professional, 2012.

[35] S. Chaki, A. Groce, and O. Strichman, “Explaining abstract counterexamples,” in *Proc. 12th ACM SIGSOFT Int’l Symp. Foundations of Software Eng. (FSE 12)*, 2004, pp. 73–82.

[36] P. P. Chan and C. Collberg, “A method to evaluate CFG comparison algorithms,” in *Proc. 14th Int’l Conf. Quality Software (QSIC 2014)*, 2014, pp. 95–104.

[37] W. Chang, B. Streiff, and C. Lin, “Efficient and extensible security enforcement using dynamic data flow analysis,” in *Proc. 15th ACM Conf. Comput. Commun. Security (CCS 2008)*, 2008, pp. 39–50.

[38] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan, “Types of software evolution and software maintenance,” *J. Software Maintenance and Evolution: Research and Practice*, vol. 13, no. 1, pp. 3–30, 2001.

[39] H. Chen and D. Wagner, “MOPS: an infrastructure for examining security properties of software,” in *Proc. 9th ACM Conf. Comput. Commun. Security (CCS 2002)*, 2002, pp. 235–244.

[40] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, “TaintTrace: Efficient flow tracing with dynamic binary rewriting,” in *11th IEEE Symp. Comput. and Commun. (ISCC 2006)*, 2006, pp. 749–754.

[41] A. S. Christensen, A. Møller, and M. I. Schwartzbach, “Precise analysis of string expressions,” in *Proc. 10th Int. Conf. on Static Analysis (SAS 2003)*, 2003, pp. 1–18.

[42] R. Chugh, “IsoLATE: A type system for self-recursion,” in *Proc. Europ. Symp. Programming Languages and Systems (ESOP 2015)*, 2015, pp. 257–282.

[43] J. Clause, W. Li, and A. Orso, “Dytan : A generic dynamic taint analysis framework,” in *Proc. ACM/SIGSOFT Int’l Symp. Software Testing and Analysis (ISSTA 2007)*, 2007, pp. 196–206.

[44] Definition of 'root cause'. Collins English Dictionary. [Online]. Available: https://www.collinsdictionary.com/dictionary/english/root-cause (Accessed 7 Dec 2017).

[45] L. D'antoni, M. Veanes, B. Livshits, and D. Molnar, "Fast : A transducer-based language for tree manipulation," *ACM Trans. Programming Languages and Systems*, vol. 38, no. 1, pp. 1–32, oct 2015.

[46] M. K. Davidsen and J. Krogstie, "A longitudinal study of development and maintenance," *Information and Software Technol.*, vol. 52, no. 7, pp. 707–719, jul 2010.

[47] G. Dotzler and M. Philippsen, "Move-optimized source code tree differencing," in *Proc. 31st IEEE/ACM Int'l Conf. Automated Software Eng. (ASE 2016)*, 2016, pp. 660 – 671.

[48] D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "Specifying and reasoning about dynamic access-control policies," in *Proc. Int'l Joint Conf. Automated Reasoning (IJCAR 2006)*, 2006, pp. 632–646.

[49] A. Edmundson, B. Holtkamp, E. Rivera, M. Finifter, A. Mettler, and D. Wagner, "An empirical study on the effectiveness of security code review," in *Proc. 5th Int'l Symp. Eng. Secure Software and Systems (ESSoS 2013)*, 2013, pp. 197–212.

[50] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Sci. Comput, Programming*, vol. 69, no. 1-3, pp. 35–45, dec 2007.

[51] M. E. Fagan, "Advances in software inspections," *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 7, pp. 744–751, July 1986.

[52] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proc. 29th ACM/IEEE Int'l Conf. on Automated Software Eng. (ASE '14)*, 2014, pp. 313–324.

[53] M. Felderer and E. Fourneret, "A systematic classification of security regression testing approaches," *Int'l J. Software Tools for Technol. Transfer*, vol. 17, no. 3, p. 305–319, Jun. 2015.

[54] N. E. Fenton and S. L. Pfleeger, *Software Metrics : A Rigorous & Practical Approach*, 2nd ed. Boston, MA, USA: PWS Publishing Co., 1997, ch. 7, pp. 246–254.

[55] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, "Verification and change-impact analysis of access-control policies," in *Proc. 27th Int'l Conf. Software Eng. (ICSE 2005)*, 2005, pp. 196–205.

[56] V. Ganesh, M. Minnes, A. Solar-Lezama, and M. Rinard, "Word equations with length constraints: what's decidable?" in *Proc. Haifa Verification Conference (HVC 2012)*, 2012, pp. 209–226.

[57] F. Gauthier and E. Merlo, "Fast detection of access control vulnerabilities in PHP applications," in *Proc. 19th Working Conf. Reverse Eng. (WCRE '12)*, 2012, pp. 247–256.

[58] F. Gauthier, T. Lavoie, and E. Merlo, "Uncovering access control weaknesses and flaws with security-discordant software clones," in *Proc. 29th Annual Comput. Security Appl. Conf. (ACSAC 2013)*, 2013, pp. 209–218.

[59] F. Gauthier, E. Merlo, E. Stroulia, and D. Turner, "Supporting maintenance and evolution of access control models in web applications," in *Proc. 30th IEEE Int'l Conf. Software Maintenance and Evolution (ICSME '14)*, 2014, pp. 506–510.

[60] F. Gauthier, "Reverse-engineering and analysis of access control models in web applications," Ph.D. dissertation, École Polytechnique de Montréal, 2014.

[61] F. Gauthier and E. Merlo, "Alias-aware propagation of simple pattern-based properties in PHP applications," in *Proc. 12th IEEE Int'l Working Conf. Source Code Analysis and Manipulation (SCAM '12)*, 2012, pp. 44–53.

[62] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques," *ACM Comput. Surveys*, vol. 50, no. 4, pp. 1–36, aug 2017.

[63] Y. Gil and G. Lalouche, "On the correlation between size and metric validity," *Empirical Software Eng.*, vol. 22, no. 5, pp. 2585–2611, Jun. 2017.

[64] R. D. Gordon, "Measuring improvements in program clarity," *IEEE Trans. Softw. Eng.*, vol. SE-5, no. 2, pp. 79–90, March 1979.

[65] C. L. Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," *Software Quality J.*, vol. 21, no. 3, pp. 421–443, Jun. 2013.

[66] A. Groce and D. Kroening, "Making the most of BMC counterexamples," *Electron. Notes Theor. Comput. Sci.*, vol. 119, no. 2, pp. 67–81, Mar. 2005.

[67] W. Halfond, A. Orso, and P. Manolios, "WASP: Protecting web applications using positive tainting and syntax-aware evaluation," *IEEE Trans. Softw. Eng.*, vol. 34, no. 1, pp. 65–81, Jan. 2008.

[68] W. G. J. Halfond and A. Orso, “Preventing SQL injection attacks using AMNESIA,” in *Proc. 28th Int’l Conf. Software Eng. (ICSE 2006)*, 2006, pp. 795–798.

[69] Z. Han, M. Mérineau, F. Gauthier, E. Merlo, X. Li, and E. Stroulia, “Evolutionary analysis of access control models: A formal concept analysis method,” in *Proc. 25th IBM/ACM Int’l Conf. Comput. Science and Software Eng. (CASCON ’15)*, 2015, pp. 261–264.

[70] R. Holt, A. Schürr, S. E. Sim, and A. Winter. Graph eXchange Language. [Online]. Available: http://www.gupro.de/GXL/ (Accessed 25 April 2018).

[71] P. Hooimeijer and M. Veanes, “An evaluation of automata algorithms for string analysis,” Microsoft Research, Tech. Rep. MSR-TR-2010-90, 2010. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=133121 (Accessed 25 Apr 2018).

[72] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, “Fast and precise sanitizer analysis with BEK,” in *Proc. USENIX Security Symp. (USS ’11)*, 2011.

[73] S. Horwitz, T. Reps, and D. Binkley, “Interprocedural slicing using dependence graphs,” *ACM SIGPLAN Notices*, vol. 23, no. 7, pp. 35–46, Jul. 1988.

[74] H. Hu, G.-J. Ahn, and K. Kulkarni, “Discovery and resolution of anomalies in web access control policies,” *IEEE Trans. Dependable Secure Comput.*, vol. 10, no. 6, pp. 341–354, nov 2013.

[75] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone, “Guide to attribute based access control (ABAC) definition and considerations,” National Institute of Standards and Technol., Tech. Rep. 800-162, Jan. 2014.

[76] C. Huang, J. Sun, X. Wang, and Y. Si, “Selective regression test for access control system employing RBAC,” in *Proc. Int’l Conf. Information Security and Assurance (ISA’09)*, 2009, pp. 70–79.

[77] W. Huang, Y. Dong, A. Milanova, and J. Dolby, “Scalable and precise taint analysis for android,” in *Proc. Int’l Symp. Software Testing and Analysis (ISSTA 2015)*, 2015, pp. 106–117.

[78] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, “Securing web application code by static analysis and runtime protection,” in *Proc. 13th Int’l Conf. World Wide Web (WWW 2004)*, 2004, pp. 40–52.

[79] Y.-Y. Huang, K. Chen, and S.-L. Chiang, “Finding security vulnerabilities in java web applications with test generation and dynamic taint analysis,” in *Proc. 2nd Int’l Congress Comput. Appl. and Computational Sci.*, 2012, vol. 145, pp. 133–138.

[80] J. Hwang, D. Y. Lee, L. Williams, and M. Vouk, “Access control policy evolution: An empirical study,” in *Proc. 25th IEEE Int’l Symp. on Software Reliability Eng. (ISSRE 2014)*, 2014, pp. 245–254.

[81] 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 Int’l Conf. Automated Software Eng. (ASE 2012)*, 2012, pp. 266–269.

[82] (2013, Apr.) Frequently asked questions regarding IEEE permissions. IEEE. [Online]. Available: https://www.ieee.org/publications_standards/publications/rights/permissions_faq.pdf (Accessed 15 Oct 2017).

[83] *Software Engineering – Software Life Cycle Processes – Maintenance*, ISO/IEC Std. 14 764:2006, 2006.

[84] M.-A. Jashki, R. Zafarani, and E. Bagheri, “Towards a more efficient static software change impact analysis method,” in *Proc. of the 8th ACM SIGPLAN-SIGSOFT workshop Program analysis for software tools and Eng. (PASTE’08)*. ACM Press, 2008.

[85] M. Jaume, “Semantic comparison of security policies: From access control policies to flow properties,” in *Proc. IEEE Symp. Security and Privacy Workshops (SPW 2012)*, 2012, pp. 60 – 67.

[86] R. Jhala and R. Majumdar, “Software model checking,” *ACM Comput. Surveys*, vol. 41, no. 4, pp. 1–54, oct 2009.

[87] J. A. Jones and M. J. Harrold, “Empirical evaluation of the Tarantula automatic fault-localization technique,” in *Proc. 20th IEEE/ACM Int’l Conf. Automated Software Eng. (ASE 2005)*, 2005, pp. 273–282.

[88] N. Jovanovic, C. Kruegel, and E. Kirda, “Pixy: A static analysis tool for detecting web application vulnerabilities (short paper),” in *Proc. IEEE Symp. Security and Privacy (SP ’06)*, 2006, pp. 258–263.

[89] ——, “Pixy: A static analysis tool for detecting web application vulnerabilities (technical report),” Vienna University of Technology, Tech. Rep., 2006. [Online].

Available: http://www.seclab.tuwien.ac.at/papers/pixy_techreport.pdf (Accessed 25 Apr 2018).

[90] K. Kane and J. C. Browne, "On classifying access control implementations for distributed systems," in *Proc. 11th ACM Symp. Access Control Models and Technol. (SACMAT 2006)*, 2006, pp. 29–38.

[91] A. H. Karp, H. Haury, and M. H. Davis, "From ABAC to ZBAC: The evolution of access control models," HP Laboratories, Tech. Rep. HPL-2009-30, 2009. [Online]. Available: http://www.hpl.hp.com/techreports/2009/HPL-2009-30.pdf (Accessed 27 July 2016).

[92] U. P. Kedhker, A. Sanyal, and B. Karkare, *Data flow analysis: theory and practice.* Boca Raton, FL, USA: CRC Press, 2009, ch. 9.

[93] V. Kolovski, J. Hendler, and B. Parsia, "Analyzing web access control policies," in *Proc. 16th Int'l Conf. World Wide Web (WWW 2007)*, 2007, pp. 677–686.

[94] S. Kumar, A. Sanyal, and U. P. Khedker, "Value slice: A new slicing concept for scalable property checking," in *Proc. 21st Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015)*, 2015, pp. 101–115.

[95] M. S. Lam, M. Martin, B. Livshits, and J. Whaley, "Securing web applications with static and dynamic information flow tracking," in *Proc. ACM SIGPLAN Symp. on Partial evaluation and semantics-based program manipulation (PEPM 2008)*, 2008, pp. 3–12.

[96] W. Landi, "Undecidability of static analysis," *ACM Letters Programming Languages and Systems*, vol. 1, no. 4, pp. 323–337, dec 1992.

[97] C. E. Landwehr, A. R. Bull, J. P. Mcdermott, and W. S. Choi, "A taxonomy of computer program security flaws, with examples," Naval Research Laboratory, Technical Report, 1994. [Online]. Available: http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA465587 (Accessed 20 Jan 2016).

[98] T. Latvala, A. Biere, K. Heljanko, and T. Junttila, "Simple bounded LTL model checking," in *Proc. 5th Int'l Conf. on Formal Methods in Comput.-Aided Design (FMCAD 04)*, 2004, pp. 186–200.

[99] M.-A. Laverdiere, B. J. Berger, and E. Merlo, "Taint analysis of manual service compositions using cross-application call graphs," in *Proc. 22nd IEEE Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER 2015)*, 2015, pp. 585–589.

[100] M.-A. Laverdière and E. Merlo, "Detection of protection-impacting changes during software evolution," in *Proc. 25th IEEE Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER 2018)*, 2018, pp. 434–444.

[101] ——, "Computing counter-examples for privilege protection losses using security models," in *Proc. IEEE 24th Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER '17)*, 2017, pp. 240–249.

[102] ——, "RBAC protection-impacting changes : A case study of PHP application evolution," *Submitted to IEEE Trans. Soft. Eng.*, 2018.

[103] ——, "Classification and distribution of RBAC privilege protection changes in wordpress evolution," in *To appear in Proc. 15th Int'l Conf. Privacy, Security and Trust (PST 2017)*, 2017.

[104] T. Lavoie and E. Merlo, "How much really changes? a case study of firefox version evolution using a clone detector," in *Proc. 7th Int'l Workshop Software Clones (IWSC 2013)*, 2013, pp. 83–89.

[105] ——, "Performance impact of lazy deletion in metric trees for incremental clone analysis," in *Proc. 9th IEEE Int'l Workshop Software Clones (IWSC 2015)*, 2015, pp. 15–18.

[106] H. T. Le, C. D. Nguyen, L. Briand, and B. Hourte, "Automated inference of access control policies for web applications," in *Proc. 20th ACM Symp. Access Control Models and Technol. (SACMAT 2015)*, 2015.

[107] W. Le and S. D. Pattison, "Patch verification via multiversion interprocedural control flow graphs," in *Proc. 36th Int'l Conf. Software Eng. (ICSE 2014)*, 2014, pp. 1047–1058.

[108] Y. Le Traon, T. Mouelhi, A. Pretschner, and B. Baudry, "Test-driven assessment of access control in legacy applications," in *Proc. Int'l Conf. Software Testing, Verification, and Validation (ICST 2008)*, 2008, pp. 238–247.

[109] M.-G. Lee and T. Jefferson, "An empirical study of software maintenance of a web-based Java application," in *Proc. 21st IEEE Int'l Conf. Software Maintenance (ICSM 2005)*, 2005, pp. 571–576.

[110] J. Lerch, J. Spath, E. Bodden, and M. Mezini, "Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths," in *Proc. 30th IEEE/ACM Int'l Conf. Automated Software Eng. (ASE 2015)*, 2015, pp. 619 – 629.

[111] D. Letarte and E. Merlo, "Extraction of inter-procedural simple role privilege models from PHP code," in *Proc. 16th IEEE Working Conf. Reverse Eng. (WCRE '09)*, 2009, pp. 187–191.

[112] D. Letarte, F. Gauthier, and E. Merlo, "Security model evolution of PHP web applications," in *Proc. 4th IEEE Int'l Conf. Software Testing, Verification and Validation (ICST '11)*, 2011, pp. 289–298.

[113] S. Letovsky and E. Soloway, "Delocalized plans and program comprehension," *IEEE Software*, vol. 3, no. 3, pp. 41–49, May 1986.

[114] O. Lhoták and L. Hendren, "Relations as an abstraction for BDD-based program analysis," *ACM Trans. Programming Languages and Systems*, vol. 30, no. 4, pp. 1–63, Jul. 2008.

[115] O. Lhoták. ProBe: Data formats for recording program behaviour. [Online]. Available: https://plg.uwaterloo.ca/~olhotak/probe/ (Accessed 25 April 2018).

[116] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Software Testing, Verification and Reliability*, vol. 23, no. 8, pp. 613–646, Apr. 2012.

[117] H. Li and W. Cheung, "An empirical study of software metrics," *IEEE Trans. Softw. Eng.*, vol. SE-13, no. 6, pp. 697–708, jun 1987.

[118] X. Li and Y. Xue, "A survey on server-side approaches to securing web applications," *ACM Comput. Surveys*, vol. 46, no. 4, pp. 1–29, Mar. 2014.

[119] D. Lin, P. Rao, E. Bertino, N. Li, and J. Lobo, "EXAM: a comprehensive environment for the analysis of access control policies," *Int'l J. of Information Security*, vol. 9, no. 4, pp. 253–273, May 2010.

[120] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie, "AutoPaG: Towards automated software patch generation with source code root cause identification and repair," in *Proc. 2nd ACM Symp. Information, Comput. Commun. Security (ASIACCS '07)*, 2007, pp. 329–340.

[121] B. Livshits and S. Chong, "Towards fully automatic placement of security sanitizers and declassifiers," in *Proc. 40th annual ACM SIGPLAN-SIGACT symp. Principles of programming languages (POPL 2013)*, 2013, pp. 385–398.

[122] B. Livshits, M. Martin, and M. S. Lam, “Securifly: Runtime protection and recovery from web application vulnerabilities,” Stanford University, Tech. Rep., September 2006.

[123] V. B. Livshits and M. S. Lam, “Finding security errors in Java programs with static analysis,” Stanford University, Tech. Rep., Aug. 2005.

[124] M. Martin, B. Livshits, and M. S. Lam, “Finding application errors and security flaws using PQL,” in *Proc. 20th ACM SIGPLAN conf. Object-oriented programming, systems, languages, and Appl. (OOPSLA ’05)*, 2005, pp. 365–383.

[125] G. McGraw, *Software Security: Building Security in.* Upper Saddle River, NJ, USA: Addison-Wesley Professional, 2006.

[126] ——, *Software Security: Building Security in.* Upper Saddle River, NJ, USA: Addison-Wesley Professional, 2006, ch. 3.

[127] E. Merlo, D. Letarte, and G. Antoniol, “Automated protection of PHP applications against SQL-injection attacks,” in *Proc. 11th European Conf. Software Maintenance and Reengineering (CSMR 2007)*, 2007, pp. 191–202.

[128] Y. Minamide, “Static approximation of dynamically generated web pages,” in *Proc. 14th Int’l Conf. World Wide Web (WWW 2005)*, 2005, pp. 432–441.

[129] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu, “Taintpipe: Pipelined symbolic taint analysis,” in *Proc. 24th USENIX Security Symposium (USENIX Security 2015)*, 2015, pp. 65–80.

[130] J. Ming, D. Wu, J. Wang, G. Xiao, and P. Liu, “StraightTaint: decoupled offline symbolic taint analysis,” in *Proc. 31st IEEE/ACM Int’l Conf. Automated Software Eng. (ASE 2016)*, 2016.

[131] MITRE Corporation. (2015, 12) CWE-425: Direct Request (’Forced Browsing’). [Online]. Available: https://cwe.mitre.org/data/definitions/425.html (Accessed 18 Oct 2016).

[132] A. Mohosina and M. Zulkernine, “DESERVE: A framework for detecting program security vulnerability exploitations,” in *Proc. IEEE 6th Int’l Conf. Software Security and Reliability (SERE 2012)*, 2012, pp. 98–107.

[133] M. Monga, R. Paleari, and E. Passerini, “A hybrid analysis framework for detecting web application vulnerabilities,” in *ICSE Workshop Software Eng. for Secure Systems (SESS 2009)*, 2009, pp. 25–32.

[134] L. Montrieux, M. Wermelinger, and Y. Yu, “Challenges in model-based evolution and merging of access control policies,” in *Proc. 12th Int’l Workshop Principles of Software Evolution and 7th ERCIM Workshop Software Evolution (IWPSE-EVOL ’11)*, 2011, pp. 116–120.

[135] T. Mouelhi, Y. Le Traon, and B. Baudry, “Transforming and selecting functional test cases for security policy testing,” in *Proc. 2nd Int’l Conf. Software Testing Verification and Validation (ICST ’09)*, 2009, pp. 171 – 180.

[136] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan, “An empirical study of static call graph extractors,” *ACM Trans. Software Eng. and Methodology*, vol. 7, no. 2, pp. 158–191, Apr. 1998.

[137] T. B. Muske, A. Baid, and T. Sanas, “Review efforts reduction by partitioning of static analysis warnings,” in *Proc. 13th IEEE Int’l Working Conf. Source Code Analysis and Manipulation (SCAM 13)*, Sept 2013, pp. 106–115.

[138] *eXtensible Access Control Markup Language (XACML)*, OASIS Std., 2013. [Online]. Available: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf (Accessed 21 Dec 2016).

[139] J. Oberheide, E. Cooke, and F. Jahanian, “If it ain’t broke, don’t fix it: challenges and new directions for inferring the impact of software patches,” in *Proc. 12th USENIX Conf. Hot topics in operating systems (HotOS 2009)*, 2009.

[140] A. Orso, T. Apiwattanapong, and M. J. Harrold, “Leveraging field data for impact analysis and regression testing,” *ACM SIGSOFT Software Eng. Notes*, vol. 28, no. 5, p. 128, sep 2003.

[141] M. Ouellet, E. Merlo, N. Sozen, and M. Gagnon, “Locating features in dynamically configured avionics software,” in *Proc. 34rd Int’l Conf. Software Eng. (ICSE 2012)*. IEEE, jun 2012.

[142] OWASP Foundation. (2008) OWASP Testing Guide. [Online]. Available: http://www.owasp.org/images/5/56/OWASP_Testing_Guide_v3.pdf (Accessed 18 Oct 2016).

[143] ——. (2014, 9) OWASP testing guide v4. [Online]. Available: https://www.owasp.org/index.php/OWASP_Testing_Project (Accessed 25 Apr 2018).

[144] ——. (2015) OWASP application security verification standard 3.0. [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project (Accessed 06 Jan 2018).

[145] ——, "OWASP Top 10 – 2013," 2013. [Online]. Available: https://www.owasp.org/index.php/Top_10_2013 (Accessed 25 April 2018).

[146] ——. (2017) OWASP Top 10 – 2017. [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project (Accessed 25 Apr 2018).

[147] R. Padhye and U. P. Khedker, "Interprocedural data flow analysis in soot using value contexts," in *Proc. 2nd ACM SIGPLAN Int'l Workshop State Of the Art in Java Program analysis (SOAP 2013)*, 2013, pp. 31–36.

[148] H. Peine, M. Jawurek, and S. Mandel, "Security goal indicator trees: A model of software features that supports efficient security inspection," in *Proc. 11th IEEE High Assurance Systems Eng. Symp. (HASE '08)*, 2008, pp. 9–18.

[149] T. Preston-Werner. (2013, 06) Semantic Versioning 2.0.0. [Online]. Available: http://www.semver.org (Accessed 12 September 2016).

[150] A. Pretschner, T. Mouelhi, and Y. L. Traon, "Model-based tests for access control policies," in *Proc. 1st Int'l Conf. Software Testing, Verification, and Validation (ICST 2008)*, 2008, pp. 338–347.

[151] M. Priyadharshini, J. Yowan, and R. Baskaran, "Security enhancement in web services by detecting and correcting anomalies in XACML policies at design level," in *Proc. Int'l Symp. Security in Computing and Commun. (SSCC 2014)*, 2014, pp. 120–135.

[152] N. Qamar, Y. Ledru, and A. Idani, "Evaluating RBAC supported techniques and their validation and verification," in *Proc. 6th Int'l Conf. Availability, Reliability and Security (ARES '11)*, 2011, pp. 734–739.

[153] C. D. P. K. Ramli, H. R. Nielson, and F. Nielson, "XACML 3.0 in answer set programming," in *Proc. Int'l Symp. Logic-Based Program Synthesis and Transformation (LOPSTR 2012)*, 2013, pp. 89–105.

[154] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proc. 22nd ACM SIGPLAN-SIGACT Symp. Principles of programming languages (POPL 1995)*, 1995, pp. 49–61.

[155] C. K. Roy and J. R. Cordy, “A survey on software clone detection research,” Queen’s University, Tech. Rep. 007-541, Sep. 2007. [Online]. Available: http://research.cs.queensu.ca/TechReports/Reports/2007-541.pdf (Accessed 26 Apr 2018).

[156] P. Runeson and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” *Empirical Software Eng.*, vol. 14, no. 2, pp. 131–164, Dec. 2008.

[157] M. Sagiv, T. Reps, and S. Horwitz, “Precise interprocedural dataflow analysis with applications to constant propagation,” *Theoretical Comput. Sci.*, vol. 167, no. 1-2, pp. 131–170, 1996.

[158] J. Saltzer and M. Schroeder, “The protection of information in computer systems,” *Proc. of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[159] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren, “Automated repair of HTML generation errors in PHP applications using string constraint solving,” in *Proc. 34rd Int’l Conf. Software Eng. (ICSE 2012)*, 2012, pp. 277–287.

[160] R. Sandhu, D. Ferraiolo, and R. Kuhn, “The NIST model for role-based access control,” in *Proc. 5th ACM Workshop Role-based access control (RBAC 2000)*, 2000, pp. 47 – 63.

[161] P. Saxena, D. Molnar, and B. Livshits, “SCRIPTGARD : automatic context-sensitive sanitization for large-scale legacy web applications,” in *Proc. 18th ACM Conf. Comput. Commun. Security (CCS 2011)*, 2011, pp. 601–614.

[162] J. Schneider, “Tracking down root causes of defects in simulink models,” in *Proc. 29th ACM/IEEE Int’l Conf. Automated Software Eng. (ASE 2014)*, 2014, pp. 599–604.

[163] A. Sergeev and R. Matulevicius, “An approach to capture role-based access control models from spring web applications,” in *21st IEEE Int’l Enterprise Distributed Object Computing Conf. (EDOC 2017)*, 2017, pp. 159 – 164.

[164] Y. Shin, L. Williams, and T. Xie, “SQLUnitGen: Test case generation for SQL injection detection,” North Carolina State University, Tech. Rep. TR-2006-21, 2006. [Online]. Available: https://repository.lib.ncsu.edu/handle/1840.4/826 (Accessed 25 Apr 2018).

[165] R. Shirey. (2007, 08) Internet security glossary. Internet Engineering Task Force. [Online]. Available: https://www.ietf.org/rfc/rfc4949.txt (Accessed 21 Dec 2016).

[166] V. S. Sinha, S. Sinha, and S. Rao, "BUGINNINGS: Identifying the origins of a bug," in *Proc. 3rd India Software Eng. Conf. (ISEC '10)*, 2010, pp. 3–12.

[167] S. Son, K. S. McKinley, and V. Shmatikov, "RoleCast: Finding missing security checks when you do not know what checks are," in *Proc. 26th ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Appl. (OOPSLA 11)*, 2011, pp. 1069–1084.

[168] ——, "Fix Me Up: Repairing access-control bugs in web applications," in *Proc. 20th Network and Distributed System Security Symp. (NDSS 2013)*, 2013.

[169] M. Sridharan, S. J. Fink, and R. Bodik, "Thin slicing," *ACM SIGPLAN Notices*, vol. 42, no. 6, p. 112, Jun. 2007.

[170] A. Steinhauser and F. Gauthier, "JSPChecker : Static detection of context-sensitive cross-site scripting flaws in legacy web applications," in *Proc. ACM Workshop Programming Languages and Analysis for Security (PLAS 2016)*, 2016, pp. 57–68.

[171] M. Taghdiri, G. Snelting, and C. Sinz, "Information flow analysis via path condition refinement," in *Proc. 7th Int'l Workshop Formal Aspects in Security and Trust (FAST)*, 2010, pp. 65–79.

[172] H. Thompson, "Why security testing is hard," *IEEE Security & Privacy Magazine*, vol. 1, no. 4, pp. 83–86, Jul. 2003.

[173] F. Thung, D. Lo, and L. Jiang, "Automatic recovery of root causes from bug-fixing changes," in *Proc. 20th Working Conf. Reverse Eng. (WCRE 2013)*. IEEE, Oct. 2013, pp. 92–101.

[174] F. Tip, "A survey of program slicing techniques," *J. Programming Languages*, vol. 3, no. 3, pp. 121–189, 1995.

[175] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "TAJ: effective taint analysis of web applications," *SIGPLAN Not.*, vol. 44, no. 6, pp. 87–97, 2009.

[176] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "Andromeda: Accurate and scalable security analysis of web applications," in *Proc. 16th Int'l Conf. Fundamental Approaches to Software Eng. (FASE 2013)*, 2013, pp. 210–225.

[177] F. Turkmen, J. den Hartog, S. Ranise, and N. Zannone, "Analysis of XACML policies with SMT," in *Proc. Int'l Conf. Principles of Security and Trust (POST 2015)*, 2015, pp. 115–134.

[178] Efficient XML interchange (EXI) format 1.0 (second edition). W3C. [Online]. Available: https://www.w3.org/TR/exi/ (Accessed 24 April 2018).

[179] H. Wang and C. Wang, “Taxonomy of security considerations and software quality,” *Commun. ACM*, vol. 46, no. 6, pp. 75–78, Jun. 2003.

[180] Q. Wang, W. Wang, R. Brown, K. Driesen, B. Dufour, L. Hendren, and C. Verbrugge, “EVolve,” in *Proc. ACM Symposium Software Visualization (SoftVis ’03)*, 2003, pp. 37–46.

[181] G. Wassermann and Z. Su, “Sound and precise analysis of web applications for injection vulnerabilities,” in *Proc. ACM SIGPLAN Conf. Programming language design and implementation (PLDI 2007)*, 2007, pp. 32–41.

[182] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, “Permission evolution in the android ecosystem,” in *Proc. 28th Annual Comput. Security Appl. Conf. (ACSAC 2012)*, 2012, pp. 31–40.

[183] M. Weiser, “Program slicing,” in *Proc. 5th Int’l Conf. Software Eng. (ICSE 1981)*, 1981.

[184] ——, “Program slicing,” *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 352–357, Jul. 1984.

[185] Manual:user rights. Wikimedia Foundation. [Online]. Available: https://www.mediawiki.org/wiki/Manual:User_rights (Accessed 6 Dec 2017).

[186] MediaWiki releases. Wikimedia Foundation. [Online]. Available: https://releases.wikimedia.org/mediawiki (Accessed 25 Apr 2018).

[187] Codex: Roles and capabilities. WordPress. [Online]. Available: https://codex.wordpress.org/Roles_and_Capabilities (Accessed 6 Dec 2017).

[188] Release archive. WordPress. [Online]. Available: https://wordpress.org/download/release-archive/ (Accessed 25 Apr 2018).

[189] F. Yu, M. Alkhalaf, and T. Bultan, “Patching vulnerabilities with sanitization synthesis,” in *Proc. 33rd Int’l Conf. Software Eng. (ICSE 2011)*, 2011, pp. 251–260.

[190] F. Yu, C.-Y. Shueh, C.-H. Lin, Y.-F. Chen, B.-Y. Wang, and T. Bultan, “Optimal sanitization synthesis for web application vulnerability repair,” in *Proc. 25th Int’l Symp. Software Testing and Analysis (ISSTA 2016)*, 2016, pp. 189–200.

[191] N. Zhang, M. Ryan, and D. P. Guelev, “Evaluating access control policies through model checking,” in *Proc. Int’l Conf. Information Security (ISC 2005)*, 2005, pp. 446–460.

APPENDIX A Example

We now provide a fictitious example of definite protection differences in Figure A.1 in Unified Diff syntax. The access control policy specifies that `foo()` must always be executed with privilege 'p' verified. In the version Ver_a, the access control policy was implemented correctly.

In version Ver_b, a developer implemented a feature which did not require any privileges in `c.php`. Unaware of the access control policy, the developer mistakenly adds a call to `foo()` from an unprotected context. Another developer modifies `a.php` in an unrelated manner, adding a call to `bar()`. Both developers add their changes to the access control repository in separate commits.

When comparing Ver_a with Ver_b, the code in function `foo()` is loss-affected (`b.php` line 1) and the code in function `bar()` (`b.php` line 5) is gain-affected. For `foo()`, the protection-impacting changes are lines 1 and 2 in `c.php`. For `bar()`, the protection-impacting change is line 7 in `a.php`.

We show the changes between the CFGs or Ver_a and Ver_b in Figure A.2. The code added in `c.php` results in an added unprotected path, which we show in Figure A.2a. Adding the call to `bar` is shown in A.2b. Finally, the deletion of an `echo` statement in `a.php` is shown in Figure A.2c. We have greyed out the CFG vertices that are unchanged. In other words, the grey vertices belong to $vertexMap$, whereas the white ones do not. This makes it easier to visualize the contribution of added and deleted edges.

```
1    require_once ("b.php");
2
3    if (!current_user_can('p'))
4      die();
5  - echo "Authorized\n";
6    foo();
7  + bar();
```

Listing A.1 a.php

```
1  function foo(){
2  //Assumes privilege p
3  }
4
5  function bar(){
6  //No security assumptions
7  }
```

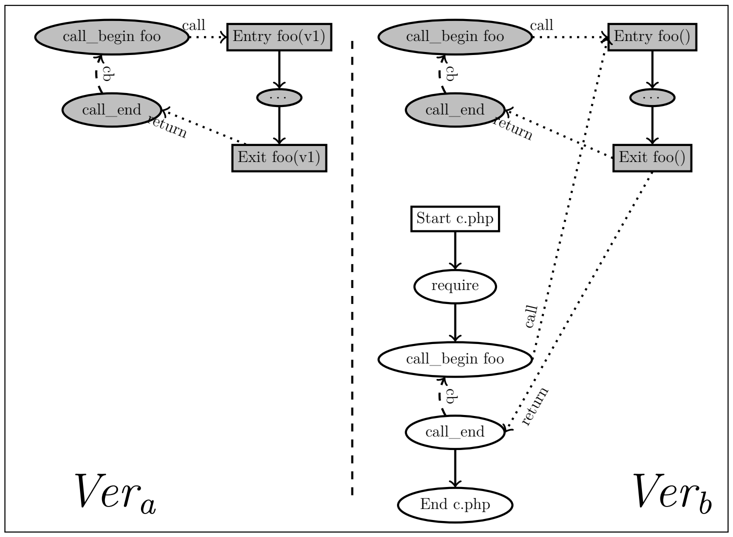
Listing A.2 b.php

```
1  + require_once("b.php");
2  + foo();
3  + //...
```

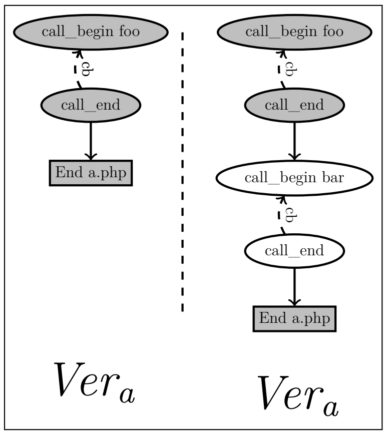
Listing A.3 c.php

Figure A.1 Example of Protection-Impacting Changes. + and - Represents Insertions and Deletions in Ver_b

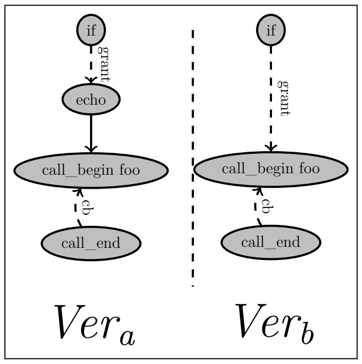
Please consider the code change shown in Figure A.2c. None of the code is new (all vertices are grey), however, the edges are different. We notice that the vertex for the `echo` statement, and its accompanying edges, is missing. Also, we observe a new edge connecting the `if` to the `call_begin` statement.



(a) New Path



(b) Addition to Path

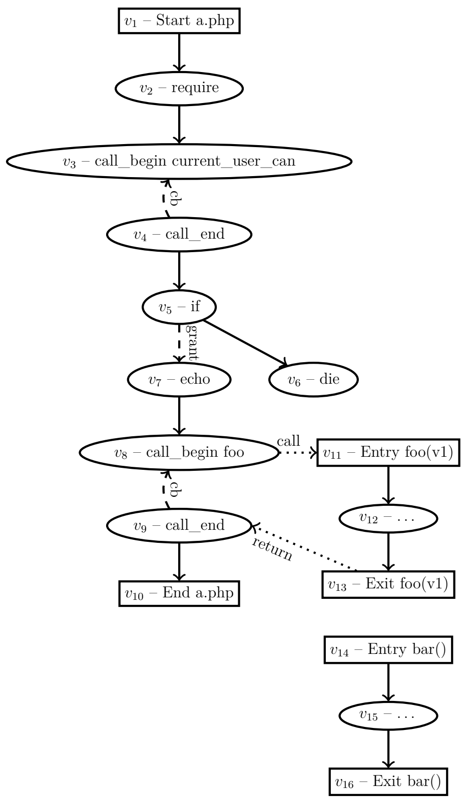


(c) Deletion

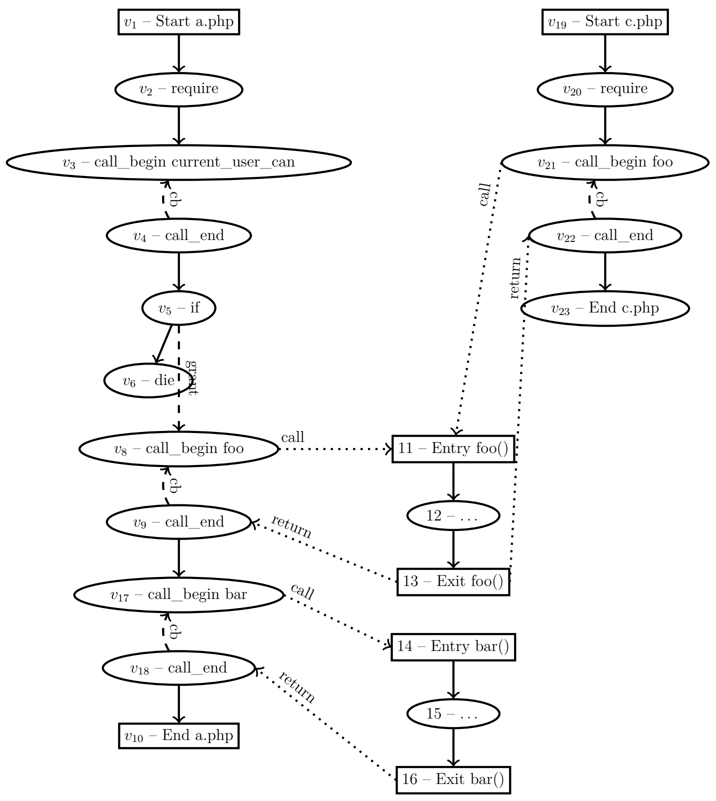
Figure A.2 Kinds of Changed Edges – CFG Subsets – Unchanged Vertices Greyed Out

We show the CFGs for our example in Figure A.3. Please note that, in this example, $vertexMap$ is the identity function.

We show the PTFA models for loss-impacted code in Figure A.4. We do likewise for gain-impacted code in Figure A.5. We highlight *addedEdges* and *deletedEdges* with red arrows. We drew edges belonging to the set of protection-impacting changes with thicker arrows. We highlight the applicable definite protection differences using a background pattern in



(a) CFG for Ver_a



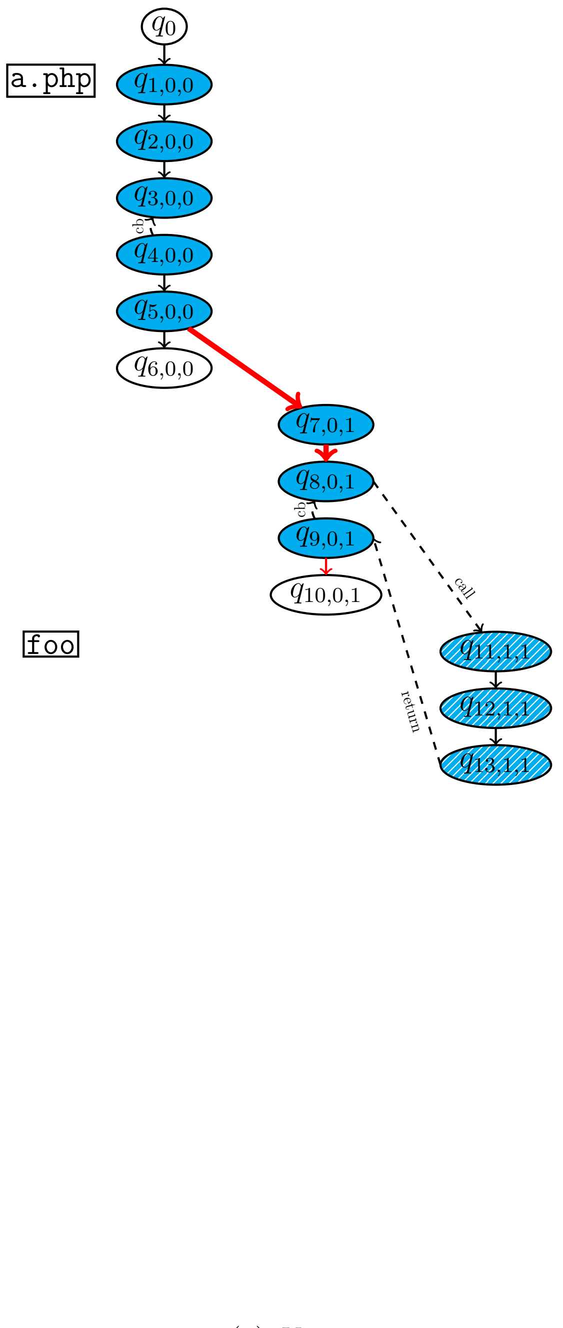
(b) CFG for Ver_b

Figure A.3 Simplified CFGs of the Example – $vertexMap$ is the identity function

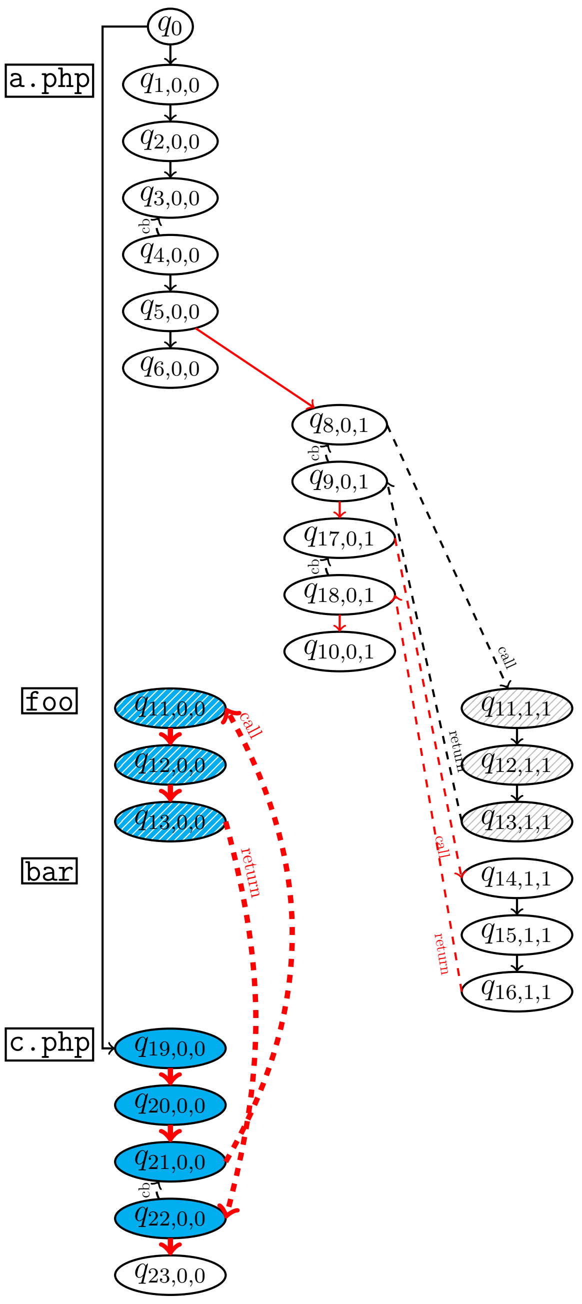
the states. We also coloured states that belong to appropriately protected paths to definite protection differences in blue. As we mentioned earlier, the i in PTFA state $q_{i,j,k}$ means that $q_{i,j,k}$ corresponds to CFG vertex v_i in Figure A.2.

The DPD root cause for loss-affected code in Figure A.4 is the paths caused by code added in `c.php`. The deleted edges in `a.php` are not DPD root causes, but are identified as protection-impacting changes. PIC_{loss} is thus a superset of the DPD root causes. The DPD root cause for gain-affected code in Figure A.5 is the added call to `bar` in `a.php`. PIC_{gain} is a superset of the DPD root cases in this case as well.

Some edges counter-intuitively belong to $addedEdges$ and $deletedEdges$. Since CFG vertex v_7 has no entry in $vertexMap$, the edges $(q_{5,0,0}\ ,\ q_{7,0,1})$, and $(q_{7,0,1}\ ,\ q_{8,0,1})$ are deleted. The edge $(q_{9,0,1}, q_{10,0,1})$ is deleted because code is inserted between the states. Therefore, the control flow of $q_{9,0,1}$ in Ver_b must go to v_17 instead. Similarly, the edge between $(q_{5,0,0}\ ,\ q_{8,0,1})$ is added because there is no corresponding edge in Ver_b. Because v_7 was removed, the control flow goes directly from v_5 to vertex v_8.



(a) *Ver*$_a$

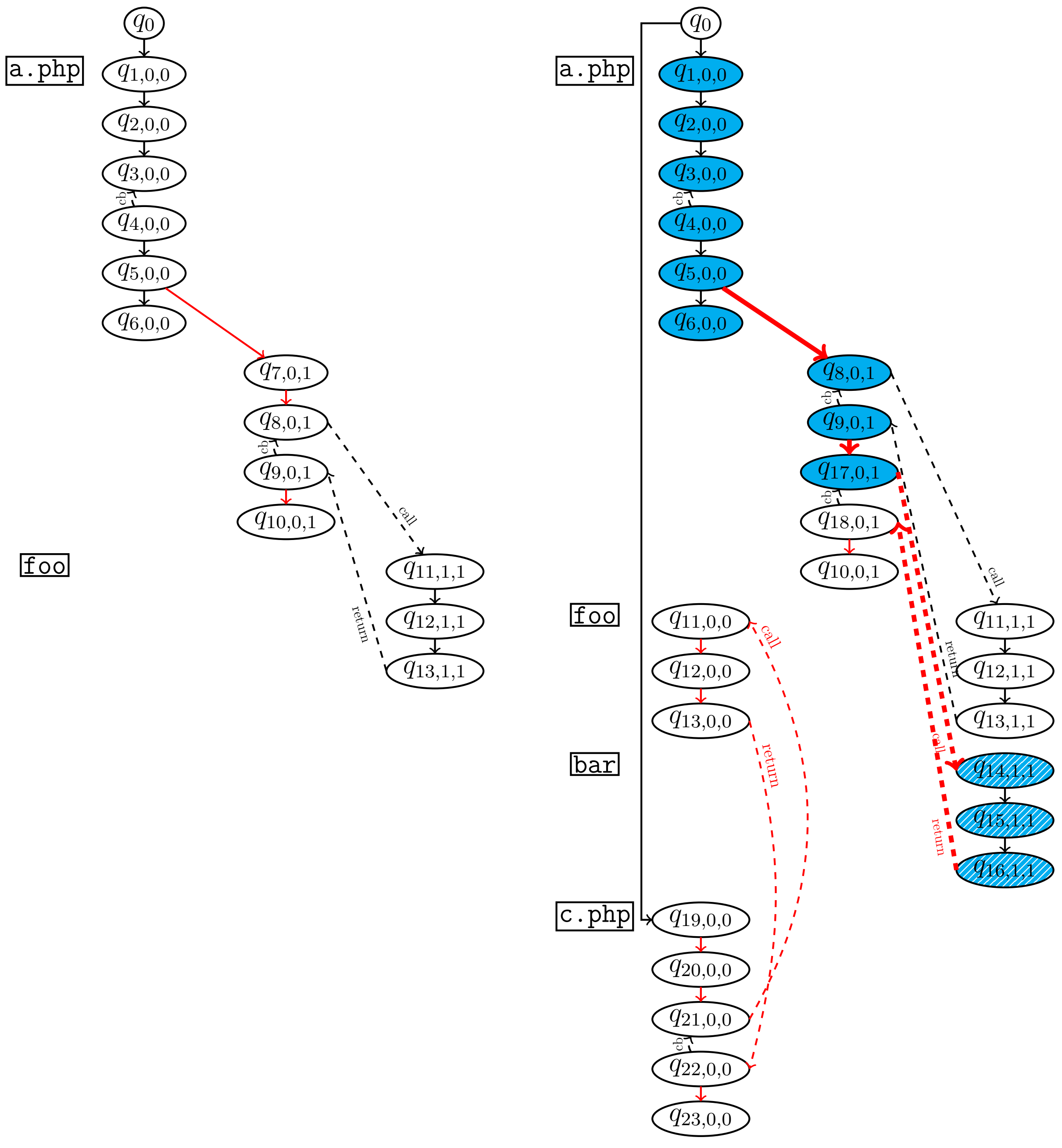


(b) *Ver*$_b$

Legend

▨ $q_{i,j,k}$	Definite Protection Difference	$\longrightarrow$	Added or deleted edge
● $q_{i,j,k}$	States in Appropriately Protected Path to Loss-Affected Code	$\Longrightarrow$	Protection-Impacting Change

Figure A.4 Simplified PTFA Models of Our Example – Protection-Impacting Changes for Loss-Impacted Vertices Highlighted.



(a) Ver_a (b) Ver_b

Legend

▨ $q_{i,j,k}$ Definite Protection Difference

● $q_{i,j,k}$ States in Appropriately Protected Path to Loss-Affected Code

$\longrightarrow$ Added or deleted edge

$\Longrightarrow$ Protection-Impacting Change

Figure A.5 Simplified PTFA Models of Our Example – Protection-Impacting Changes for Loss-Impacted Vertices Highlighted.

In our example, not all edges in $deletedEdges$ and $addedEdges$ are protection-impacting changes. For instance, the edge between $q_{9,0,1}$ and $q_{10,0,1}$ in Ver_a and the edge between $q_{10,0,1}$ and $q_{11,0,1}$ in Ver_b are not protection-impacting changes. However, there may be cases where protection-impacting changes will be ($deletedEdges$, $addedEdges$).

We observe that PIC_{gain} and PIC_{loss} are disjoint in our example. This occurs because protection-impacting changes take appropriately protected paths in consideration. Please consider the case in Figure A.4. The states $q_{11,1,1}$, $q_{12,1,1}$ and $q_{13,1,1}$ are reachable in both Ver_a and Ver_b. Protection-impacting changes consider the edges belong to paths to these states only in Ver_a. Consequently, the added edge ($q_{5,0,0}$, $q_{8,0,0}$) does not belong to PIC_{loss}. If we considered all paths to definite protection difference, then ($q_{5,0,0}$, $q_{8,0,0}$) would be protection-impacting.

APPENDIX B Obtaining Source Code for the Case Study

We downloaded the releases from WordPress and Mediawiki's websites [186, 188]. There were missing releases in the archive of both systems. As such, we had to reconstitute them. We did so using version control data to generate a patch, which we applied to the release archive of the earlier release. We confirmed with developers of both systems, on their IRC channels, that no additional code changes were introduced for these versions.

APPENDIX C VertexMap Construction Algorithm

In order to determine added and deleted code in CFGs and PTFA models, we create *vertex-Map*, an injective function between the CFG vertices of two releases (V_a and V_b). To do so, we use textual differencing and node metadata.

Prior to building *vertexMap*, we number the nodes in AST order. Since we are comparing identical lines of code, the order of the identifiers of the CFG vertices remains constant. After numbering the vertices, we execute BuildVertexMap (Figure C.1). For the sake of simplicity, we show the algorithm using $findOnLine$ as a function that finds all CFG vertices on a given file and line. The *sortById* function sorts the vertices according to the value of their identifier. Finally, function *zip* is from functional programming. Given two collections in input (e.g. x and y), it creates one tuple with one element of each collection (i.e. (x_i, y_i) for all positions i in x and y).

```
function BuildVertexMap($identicalLines, V_a, V_b$)
    $vertexMap \leftarrow \emptyset$
    for all $(file_a, line_a, file_b, line_b) \in identicalLines$
        $onLine_a \leftarrow findOnLine(file_a, line_a, V_a)$
        $onLine_b \leftarrow findOnLine(file_b, line_b, V_b)$
        $onLine_a \leftarrow sortById(onLine_a)$
        $onLine_b \leftarrow sortById(onLine_b)$
        $vertexMap \leftarrow vertexMap \cup zip(onLine_a, onLine_b)$
    return $vertexMap$
```

Figure C.1 *vertexMap* Construction Algorithm

APPENDIX D Release Tree Construction Algorithm

We build a tree of release pairs using their semantic versioning [149] and release date information. Semantic versioning is a *de facto* standard for numbering software releases and it is widely used in open source projects. Releases are tagged in the format $x.y.z$, where x is a major version number, y is a minor version number, and z is a patch number. Whenever y or z are missing, they are assumed to be zero. For instance, version 2 means 2.0.0 and release 2.3 means 2.3.0.

When we build the tree, we group together all releases with the same major and minor version and connect them with an edge in increasing order of bugfix version. When it comes to connect edges to .0 minor or major releases, we select a bugfix release in the previous minor release branch. The bugfix release is the highest numbered version whose release date is the same or earlier as the .0 release.

This behavior is provided by function BuildReleasePairTree, which returns the edges of the tree (Figure D.1). The root of the tree is implicit, as it is the only vertex with no predecessors. We rely on function $sortVersion$, which sorts according to the semantic versioning scheme. We also rely on function $extractMajorMinor$, which extracts the major and minor version numbers (i.e. it returns $x.y$ for an input of $x.y.z$). The function $groupByMajorMinor$ builds a multimap of releases, indexed by the major and minor version numbers. The *releaseDate* function returns the release date of a version. Finally, functions *head* and *last* respectively return the first and last element of an ordered data structure.

```
**function** BUILDRELEASEPAIRTREE ($releases$)
    $edges \leftarrow \emptyset$
    $majorMinorReleasesSorted \leftarrow versionSort(extractMajorMinor(releases))$
    $releasesInMajorMinor \leftarrow groupByMajorMinor(releases)$
    $previousMajorMinor \leftarrow Nil$
    **for all** $majorMinor \in majorMinorReleasesSorted$
        $bugfixesSorted \leftarrow versionSort(releasesInMajorMinor[majorMinor])$
        $previous \leftarrow Nil$
        // Create edges for bugfix releases in the same major.minor branch
        **for all** $bugfix \in bugfixesSorted$
            **if** $previous \neq Nil$
                $edges = edges \cup (previous, bugfix)$
            $previous \leftarrow bugfix$
        // Create edge from the previous major.minor branch
        **if** $previousMajorMinor \neq Nil$
            $dotZero \leftarrow head(bugfixesSorted)$
            $earlier \leftarrow \left\{ \begin{array}{l} v \in releasesInMajorMinor[previousMajorMinor] \;\mid \\ releaseDate(v) \leq releaseDate(dotZero) \end{array} \right\}$
            $earlierSorted \leftarrow versionSort(earlier)$
            $edges \leftarrow edges \cup (last(earlierSorted), dotZero)$
        $previousMajorMinor \leftarrow majorMinor$
  **return** $edges$
```

Figure D.1 Release Tree Construction Algorithm

APPENDIX E Revert Rules

We summarize the reversal rules and elaborate on their rationale.

In Table E.1, we represent a file present in Ver_a by f_a and a file present in Ver_b by f_b. Whenever f_a and f_b are both present in a decision rule, we mean that files f_a and f_b have the same relative path from the application's root directory. A ✓in column PIC means that protection-impacting changes are present in that file

For rule #1, the file f_a was deleted and had no protection-impacting changes. We omit this file from the reverted version.

For rule #2, the file f_a was deleted, but it had protection-impacting changes, meaning that some of its paths were necessary for privilege protection in Ver_a. Thus, we copy f_a to the reverted version.

For rule #3, the file f_b was added and it has no protection-impacting changes. As such, we copy f_b to the reverted version.

For rule #4, the file f_b was added and it contains protection-impacting changes. We omit this file from the reverted version.

For rule #5, the file f_a became f_b. Neither contain protection-impacting change, so we copy f_b to the reverted version.

For rule #6, the file f_a became f_b, and only f_a has protection-impacting changes. If the files are identical, picking either f_a or f_b yields the same result. If the files are different, then there is a deleted path in f_a that was important for privilege protection, and we need to put this path back in the reverted version. Thus, we copy f_a to the reverted version.

For rule #7, the file f_a became f_b, and only f_b has protection-impacting changes. Thus, f_b has added at least one path that affected privilege protection and we must remove this path. Therefore, we copy f_a to the reverted version.

Finally, for rule #8, f_a became f_b, and both have protection-impacting changes. The rationale for rules #6 and #7 apply. Thus, we copy f_a to the reverted version.

#	***Ver*** $_a$		***Ver*** $_b$		**File Copied**
	File	PIC	File	PIC	
1	f_a	–	–	–	–
2	f_a	✓	–	–	f_a
3	–	–	f_b	–	f_b
4	–	–	f_b	✓	–
5	f_a	–	f_b	–	f_b
6	f_a	✓	f_b	–	f_a
7	f_a	–	f_b	✓	f_a
8	f_a	✓	f_b	✓	f_a

Table E.1 Rules for Reverting Files with Protection-Impacting Changes

APPENDIX F Backwards Reachability Algorithm

We present our linear backwards racheability algorithm (BackwardsReachabilityEdges) in Figures F.1 and F.2. The first step, Traverse, performs a backwards traversal until it meets q_0. It does so by recording but without entering into called functions. When meeting a function start vertex, Traverse will continue its backwards analysis to its call sites. The second step, Fill, is similar to Traverse, but restricts itself to intraprocedural edges and transitively visits called functions. The *inter_pred* and *intra_pred* functions returns the edges. The evaluation of *bReachable* is performed when visiting a call_end vertex, and it finds all the interprocedural edges between the call begin and the call end.

This algorithm is of complexity $O(E)$, where E is the number of edges in the program. In the worst case, the algorithm will traverse all the edges in the program only once.

The *bReachable* function (Figure F.3) determines the relationships between call sites and functions in a reachable PTFA model. It also uses the *call_begin* function, which returns the CFG state that corresponds to the *call_end* state. The *entry* function returns the entry CFG vertex that corresponds to a function's exit state. *PTFARevEdges* is the set of reversed forward-reachable PTFA edges. Please note that this function is based the graph rewriting rules in PTFA [111].

```
function bPropagatableInterprocEdges ($ce \in V$)
    // Return all call and return edges connecting to this
    // call end that respects the bPropagatable equation
    $resultCalls \leftarrow \emptyset$
    $resultReturns \leftarrow \emptyset$
    for all $(exit, cb) \in inter\_pred(ce)$
        for all $cb \in call\_begin(ce)$
            for all $entry \in entries(exit)$
                if $bReachable(ce, cb, entry, exit)$
                    $resultCalls \leftarrow resultCalls \cup \{(cb, entry)\}$
                    $resultReturns \leftarrow resultReturns \cup \{(exit, ce)\}$
    return $(resultCalls, resultReturns)$
```

```
function Traverse ($changedNodeSet \subseteq V$)
    $worklist \leftarrow changedNodeSet$
    while $worklist \neq \emptyset$
        $curNode \leftarrow pop(worklist)$
        $bReachable \leftarrow bReachable \cup \{curNode\}$
        switch $curNode.type$
        case $call\_end$ //Visit function returning here in the next step
            $(calls, returns) \leftarrow bPropagatableInterprocEdges(curNode)$
            $pathEdges \leftarrow pathEdges \cup calls \cup returns$
            $toBeFilledSet \leftarrow toBeFilledSet \cup \{v_{from} \mid (v_{from}, v_{to}) \in returns\}$
            $theCbs \leftarrow \{v_{from} \mid (v_{from}, v_{to}) \in calls\}$
             //Create pseudo edges from call begin to call end
            for all $theCb \in theCbs$
                $pathEdges \leftarrow pathEdges \cup \{(theCb, curNode)\}$
                if $theCb \notin bReachable$
                    $push(worklist, theCb)$
        case $entry$ //Record and visit callers
            $pathEdges \leftarrow pathEdges \cup inter\_pred(curNode)$
            $unvisited \leftarrow \{v_{from} \mid (v_{from}, v_{to}) \in interproc\_pred(curNode) \wedge v_{from} \notin bReachable\}$
            $pushAll(worklist, unvisited)$
        default //Record and visit intraprocedural edges
            $pathEdges \leftarrow pathEdges \cup intraproc\_pred(curNode)$
            $unvisited \leftarrow \{v_{from} \mid (v_{from}, v_{to}) \in intra\_pred(curNode) \wedge v_{from} \notin bReachable\}$
            $pushAll(worklist, unvisited)$
```

Figure F.1 Backwards Reachability Algorithm

```
function FILL ($toFillNodeSet \subseteq V$)
  $worklist \leftarrow toFillNodeSet$
  while $worklist \neq \emptyset$
    $curNode \leftarrow pop(worklist)$
    $filled \leftarrow fill \cup \{curNode\}$
    switch $curNode.type$
    case $call\_end$
      $(calls, returns) \leftarrow bPropagatableInterprocEdges(curNode)$
      $pathEdges \leftarrow pathEdges \cup calls \cup returns$
      //Visit function returning here later
      $unvisited \leftarrow \{v_{from} \mid (v_{from}, v_{to}) \in returns \ \wedge \ v_{from} \notin filled \cup worklist\}$
      $pushAll(worklist, unvisited)$
      $theCbs \leftarrow \{v_{from} \mid (v_{from}, v_{to}) \in calls\}$
       //Create pseudo edges from call begin to call end
      for all $theCb \in theCbs$
        $pathEdges \leftarrow pathEdges \cup \{(theCb, curNode)\}$
        if $theCb \notin filled$
          $push(worklist, theCb)$
    default  //Record and visit intraprocedural edges
      $pathEdges \leftarrow pathEdges \cup intraproc\_pred(curNode)$
      $unvisited \leftarrow \{v_{from} \mid (v_{from}, v_{to}) \in intra\_pred(curNode) \wedge v_{from} \notin filled\}$
      $pushAll(worklist, unvisited)$

        function BACKWARDSREACHABILITYEDGES ($initNodes \subseteq V$)
          $toBeFilledNodeSet \leftarrow \emptyset$
          $pathEdges \leftarrow \emptyset$
          $traverse(initNodes)$
          $fill(toBeFilledSet)$
          return $pathEdges$
```

Figure F.2 Backwards Reachability Algorithm (continued)

$$((\mathit{call\_end}\ w,\ 0,\ 0) \in \mathit{bReachable}\ \wedge\ (\mathit{exit}\ z,\ 0,\ 0) \in Q\ \wedge\ (\mathit{call\_begin}(w),\ 0,\ 0) \in Q)$$
$$\Rightarrow \left(\begin{array}{l} (\mathit{call\_begin}(w),\ 0,\ 0) \in \mathit{bReachable} \wedge \\ ((\mathit{call\_end}\ w,\ 0,\ 0),\ (\mathit{exit}\ z,\ 0,\ 0)) \in \mathit{PTFARevEdges} \wedge \\ ((\mathit{entry}(z),\ 0,\ 0),\ (\mathit{call\_begin}(w),\ 0,\ 0)) \in \mathit{PTFARevEdges} \end{array} \right)$$

$$(\ (\mathit{call\_end}\ w,\ 0,\ 1) \in \mathit{bReachable} \wedge\ (\mathit{exit}\ z,\ 0,\ 1) \in Q\ \wedge\ (\mathit{call\_begin}(w),\ 0,\ 0) \in Q)$$
$$\Rightarrow \left(\begin{array}{l} (\mathit{call\_begin}(w),\ 0,\ 0) \in \mathit{bReachable} \wedge \\ ((\mathit{call\_end}\ w,\ 0,\ 1),\ (\mathit{exit}\ z,\ 0,\ 1) \in \mathit{PTFARevEdges}) \wedge \\ ((\mathit{entry}(z),\ 0,\ 0),\ (\mathit{call\_begin}(w),\ 0,\ 0)) \in \mathit{PTFARevEdges} \end{array} \right)$$

$$(\ (\mathit{call\_end}\ w,\ 0,\ 0) \in \mathit{bReachable} \wedge\ (\mathit{exit}\ z,\ 1,\ 0) \in Q\ \wedge\ (\mathit{call\_begin}(w),\ 0,\ 1) \in Q)$$
$$\Rightarrow \left(\begin{array}{l} (\mathit{call\_begin}(w),\ 0,\ 1) \in \mathit{bReachable} \wedge \\ ((\mathit{call\_end}\ w,\ 0,\ 0),\ (\mathit{exit}\ z,\ 1,\ 0)) \in \mathit{PTFARevEdges} \wedge \\ ((\mathit{entry}(z),\ 1,\ 1),\ (\mathit{call\_begin}(w),\ 0,\ 1) \in \mathit{PTFARevEdges}) \end{array} \right)$$

$$(\ (\mathit{call\_end}\ w,\ 0,\ 1) \in \mathit{bReachable} \wedge\ (\mathit{exit}\ z,\ 1,\ 1) \in Q\ \wedge\ (\mathit{call\_begin}(w),\ 0,\ 1) \in Q)$$
$$\Rightarrow \left(\begin{array}{l} (\mathit{call\_begin}(w),\ 0,\ 1) \in \mathit{bReachable} \wedge \\ ((\mathit{call\_end}\ w,\ 0,\ 1),\ (\mathit{exit}\ z,\ 1,\ 1)) \in \mathit{PTFARevEdges} \wedge \\ ((\mathit{entry}(z),\ 1,\ 1),\ (\mathit{call\_begin}(w),\ 0,\ 1)) \in \mathit{PTFARevEdges} \end{array} \right)$$

$$(\ (\mathit{call\_end}\ w,\ 1,\ 0) \in \mathit{bReachable} \wedge\ (\mathit{exit}\ z,\ 0,\ 0) \in Q\ \wedge\ (\mathit{call\_begin}(w),\ 1,\ 0) \in Q)$$
$$\Rightarrow \left(\begin{array}{l} (\mathit{call\_begin}(w),\ 1,\ 0) \in \mathit{bReachable} \wedge \\ ((\mathit{call\_end}\ w,\ 1,\ 0),\ (\mathit{exit}\ z,\ 0,\ 0)) \in \mathit{PTFARevEdges} \wedge \\ ((\mathit{entry}(z),\ 0,\ 0),\ (\mathit{call\_begin}(w),\ 1,\ 0)) \in \mathit{PTFARevEdges} \end{array} \right)$$

$$(\ (\mathit{call\_end}\ w,\ 1,\ 1) \in \mathit{bReachable} \wedge\ (\mathit{exit}\ z,\ 0,\ 1) \in Q\ \wedge\ (\mathit{call\_begin}(w),\ 1,\ 0) \in Q)$$
$$\Rightarrow \left(\begin{array}{l} (\mathit{call\_begin}(w),\ 1,\ 0) \in \mathit{bReachable} \wedge \\ ((\mathit{call\_end}\ w,\ 1,\ 1),\ (\mathit{exit}\ z,\ 0,\ 1)) \in \mathit{PTFARevEdges} \wedge \\ ((\mathit{entry}(z),\ 0,\ 0),\ (\mathit{call\_begin}(w),\ 1,\ 0)) \in \mathit{PTFARevEdges} \end{array} \right)$$

$$(\ (\mathit{call\_end}\ w,\ 1,\ 0) \in \mathit{bReachable} \wedge\ (\mathit{exit}\ z,\ 1,\ 0) \in Q\ \wedge\ (\mathit{call\_begin}(w),\ 1,\ 1) \in Q)$$
$$\Rightarrow \left(\begin{array}{l} (\mathit{call\_begin}(w),\ 1,\ 1) \in \mathit{bReachable} \wedge \\ ((\mathit{call\_end}\ w,\ 1,\ 0),\ (\mathit{exit}\ z,\ 1,\ 0)) \in \mathit{PTFARevEdges} \wedge \\ ((\mathit{entry}(z),\ 1,\ 1),\ (\mathit{call\_begin}(w),\ 1,\ 1)) \in \mathit{PTFARevEdges} \end{array} \right)$$

$$(\ (\mathit{call\_end}\ w,\ 1,\ 1) \in \mathit{bReachable} \wedge\ (\mathit{exit}\ z,\ 1,\ 1) \in Q\ \wedge\ (\mathit{call\_begin}(w),\ 1,\ 1) \in Q)$$
$$\Rightarrow \left(\begin{array}{l} (\mathit{call\_begin}(w),\ 1,\ 1) \in \mathit{bReachable} \wedge \\ ((\mathit{call\_end}\ w,\ 1,\ 1),\ (\mathit{exit}\ z,\ 1,\ 1)) \in \mathit{PTFARevEdges} \wedge \\ ((\mathit{entry}(z),\ 1,\ 1),\ (\mathit{call\_begin}(w),\ 1,\ 1)) \in \mathit{PTFARevEdges} \end{array} \right)$$

Figure F.3 Definitions for *bReachable*