UNIVERSITÉ DE MONTRÉAL

JUST-IN-TIME DETECTION OF PROTECTION-IMPACTING CHANGES ON WORDPRESS AND MEDIAWIKI

MOHAMED AMINE BARRAK
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
NOVEMBRE 2018

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

JUST-IN-TIME DETECTION OF PROTECTION-IMPACTING CHANGES ON WORDPRESS AND MEDIAWIKI

présenté par: BARRAK Mohamed Amine
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de:

M. MULLINS John, Ph. D., président
M. KHOMH Foutse, Ph. D., membre et directeur de recherche
M. ANTONIOL Giuliano, Ph. D., membre et codirecteur de recherche
M. FOKAEFS Marios-Eleftherios, Ph. D., membre

DEDICATION

To my parents,
For pushing me to my fullest potential,
And supporting me all these years.

ACKNOWLEDGEMENTS

At the end of this work, I would like to thank everyone who contributed to the success of my thesis and all the people (professors and students) in the software engineering team of the Department of Computer Engineering where I spent most of my time during my thesis.

Firstly, I would like to express my deepest gratitude to my supervisor, Dr. Foutse Khomh, for his great guidance, encouragement and patience that he provided along my master's studies.

Secondly, I also would like to thank Dr. Ettore Merlo for the collaboration he provide with our lab "SWAT" and especially Dr. Marc-André Laverdière for his time and the technical support that he gives to success the main objective of this thesis.

In addition, I would like to thank my co-supervisor Dr. Giuliano Antoniol and the heads of the university mission of Tunisia in Montreal for their financial support in a part of my thesis.

My most sincere thanks are extended to the committee members, Dr. John Mullins, Dr. Foutse Khomh, Dr. Giuliano Antoniol, and Dr. Marios-Eleftherios Fokaefs, for their valuable feedback on this thesis.

Finally, I take this opportunity to express my acknowledgement and my deepest respect to all the teachers I take courses with them in Polytechnique Montréal, as well as to all those who contributed directly or indirectly to the realization of this work.

RÉSUMÉ

Les mécanismes de contrôle d'accès basés sur les rôles accordés et les privilèges prédéfinis limitent l'accès des utilisateurs aux ressources sensibles à la sécurité dans un système logiciel multi-utilisateurs. Des modifications non intentionnelles des privilèges protégés peuvent survenir lors de l'évolution d'un système, ce qui peut entraîner des vulnérabilités de sécurité et par la suite menacer les données confidentielles des utilisateurs et causer d'autres graves problèmes. Dans ce mémoire, nous avons utilisé la technique "Pattern Traversal Flow Analysis" pour identifier les différences de protection introduite dans les systèmes WordPress et MediaWiki. Nous avons analysé l'évolution des privilèges protégés dans 211 et 193 versions respectivement de WordPress et Mediawiki, et nous avons constaté qu'environ 60% des commits affectent les privilèges protégés dans les deux projets étudiés. Nous nous référons au commits causant un changement protégé comme commits (PIC). Pour aider les développeurs à identifier les commits PIC en temps réel, c'est à dire dès leur soumission dans le repertoire de code, nous extrayons une série de métriques à partir des logs de commits et du code source, ensuite, nous construisons des modèles statistiques. L'évaluation de ces modèles a révélé qu'ils pouvaient atteindre une précision allant jusqu'à 73,8 % et un rappel de 98,8 % dans WordPress, et pour MediaWiki, une précision de 77,2 % et un rappel allant jusqu'à 97,8 %. Parmi les métriques examinés, changement de lignes de code, correction de bogues, expérience des auteurs, et complexité du code entre deux versions sont les facteurs prédictifs les plus importants de ces modèles. Nous avons effectué une analyse qualitative des faux positifs et des faux négatifs et avons observé que le détecteur des commits PIC doit ignorer les commits de documentation uniquement et les modifications de code non accompagnées de commentaires.

Les entreprises de développement logiciel peuvent utiliser notre approche et les modèles proposés dans ce mémoire, pour identifier les modifications non intentionnelles des privilèges protégés dès leur apparition, afin d'empêcher l'introduction de vulnérabilités dans leurs systèmes.

ABSTRACT

Access control mechanisms based on roles and privileges restrict the access of users to security sensitive resources in a multi-user software system. Unintentional privilege protection changes may occur during the evolution of a system, which may introduce security vulnerabilities, threatening user's confidential data, and causing other severe problems. In this thesis, we use the Pattern Traversal Flow Analysis technique to identify definite protection differences in WordPress and MediaWiki systems. We analyse the evolution of privilege protections across 211 and 193 releases from respectively WordPress and Mediawiki, and observe that around 60% of commits affect privileges protections in both projects. We refer to these commits as protection-impacting change (PIC) commits. To help developers identify PIC commits just-in-time, i.e., as soon as they are introduced in the code base, we extract a series of metrics from commit logs and source code, and build statistical models. The evaluation of these models revealed that they can achieve a precision up to 73.8% and a recall up to 98.8% in WordPress and for MediaWiki, a precision up to 77.2% and recall up to 97.8%. Among the metrics examined, commit churn, bug fixing, author experiences and code complexity between two releases are the most important predictors in the models. We performed a qualitative analysis of false positives and false negatives and observe that PIC commits detectors should ignore documentation-only commits and process code changes without the comments.

Software organizations can use our proposed approach and models, to identify unintentional privilege protection changes as soon as they are introduced, in order to prevent the introduction of vulnerabilities in their systems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDIXES

LIST OF SYMBOLS AND ABBREVIATIONS

LOC	Lines of Code
SCM	Source Code Management
OSS	Open Source Software
MSR	Mining Software Repositories
RBAC	Role-Based Access Control
CFG	Control Flow Graph
DPD	Definite Protection Difference
PIC	Protection-Impacting Change
PTFA	Pattern-Traversal Flow Analysis
PSP	Property Satisfaction Profiles
SQA	Software Quality Assurance

CHAPTER 1 INTRODUCTION

Web applications became a necessary component of internet development that carry daily services to millions of users to obtain information, perform financial transactions, provide them a fabulous time to socialize and communicate. Furthermore, many companies have transferred a huge portion of their services and applications on the web. Consequently, people from different places can share their information progressively and make a collaboration through web applications (Cho et Kim, 2004). On the other hand, web applications are one of the popular targets for attackers. Therefore, vulnerabilities that make an attacker able to control a web application and have access to its data pose a significant threat (Wassermann et Su, 2007). In this thesis, we focus on access control vulnerabilities in which we used machine learning techniques to predict just-in-time suspected commits that represent the vulnerability of the system. Access control, also known as authorization, is a key security mechanism in software applications, which mediates all accesses and resources according to a predefined policy.

However, in the role-based access control (RBAC) (Sandhu *et al.*, 1996), specific roles and privileges are assigned to the users of the applications and checks to ensure that all access conforms to the application's policy. It is a methodical framework for provisioning a group of roles and privileges. Allocation of the roles facilitates the access of different categories of users such as employees, contractors, and external users by incorporating policies and rules necessary to give the proper access to each of them (Rouse, 2018).

Figure 1.1 presents the functionality mechanism of the RBAC system. Users have to be granted access, to be able to get the assets. The ability to get the asset is passing through the rules which are controlling access. The rule is the permission to have access to the asset. It includes roles. The assignment of the roles can have user association in different ways. Then, the attributes determine the different association with users and rules are used to figure roles associations. Moreover, a user can be declared to have a specific associated role. Finally, all users should be associated with their attributes. In the RBAC model, what makes the controlling and maintaining access easier is the non requirement to maintain a direct relationship for each user with a specific role. The roles are acting as an abstraction layer containing a group of robust rules protecting the assets or different protected resources (Sander, 2009; Idenhaus, 2017).

Although implementing RBAC methodology appears simple, in practice it is difficult to enforce it efficiently. Based on the RBAC model, each role is associated with the authorizations

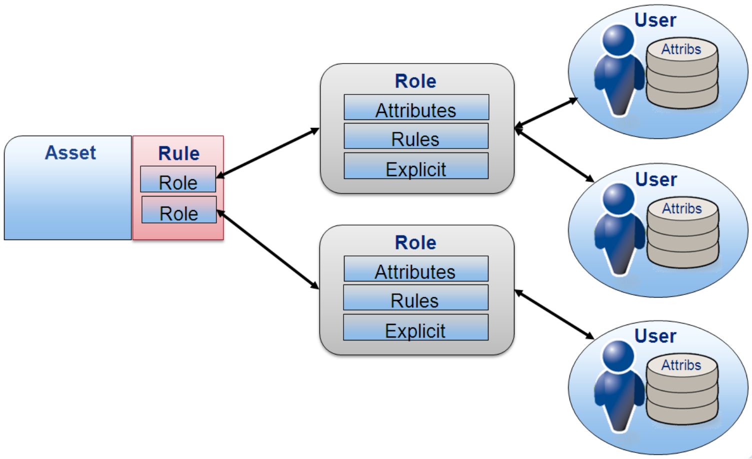


Figure 1.1 RBAC : Role Based Access Control (Idenhaus, 2017)

linked to the different responsibilities. To manage these roles, companies that handle several applications have to make continuous maintenance and even to manage hundreds or thousands of roles across applications. In such case, companies that use RBAC's role-based models are confronted with role explosion, which increases the risk of developers accidentally breaking access control rules when modifying the code (Harel, 2017).

Pinto et Stuttard (2011) studied a large number of applications between 2007 and 2011 and reported that almost 71% of all web applications suffer from broken access controls. An example of broken access control is a Fintech application which investigates whether a user is allowed to transfer money from a PayPal account, without validating that this PayPal account belongs to the user.

Most of the recently announced cases of cyber-attacks and data violation have all been attributed to errors, bugs and different forms of weakness recognized in the software development process. Several billions of dollars lost resulted from these errors (Li *et al.*, 2017a; Telang et Wattal, 2005). Even more, most of the exploitable vulnerabilities that hackers use to attack Web sites or corporate servers are usually results of programming errors Vijayan (2009). Piancó *et al.* (2016) found that the frequency of changes can reveal functions more prone to have vulnerabilities, which opens doors for us to discuss if the level of protection changes which represents a vulnerability to the system.

During the evolution of an application, developers often modify the code enforcing access control to add more functions and resources. However, unexpected changes to the roles and privileges may cause security vulnerabilities in the application.

The consequences of unintentional protection changes can also be destructive. An attacker can use the vulnerabilities on the privilege protections to view, change or delete sensitive content; execute unauthorized functions; or even take control of the administration of the application. Therefore, it is very important to analyze the code changes carefully, which affect the privilege protections in an application (Laverdière et Merlo, 2017).

In this thesis, we propose a just-in-time analysis of protection-impacting change (PIC) commits, which are commits that affect privilege protections in an application. We build a statistical model to classify the commits submitted to the source code repository. A just-in-time detection of PICs is very desirable for many reasons. First and foremost, it allows corrections in an earlier stage of the software development process, when the cost of correction is low. Second, it facilitates the identification of the root cause of the issue. In comparison, the accumulation of many weeks or months of code changes may result in complicated protection-impacting changes. This would make identification of the root cause and corrections much harder. And finally, an analysis based on statistical models is expected to be faster.

In Figure 1.2, we present our proposed just-in-time prediction model. Before pushing a commit into the repository, the developer could run the proposed model to have an idea about the risk of breaking a protection If the risk is higher than a certain threshold (decided by the development team), the developer has to do refactoring on the changes and submit it for a thorough review.

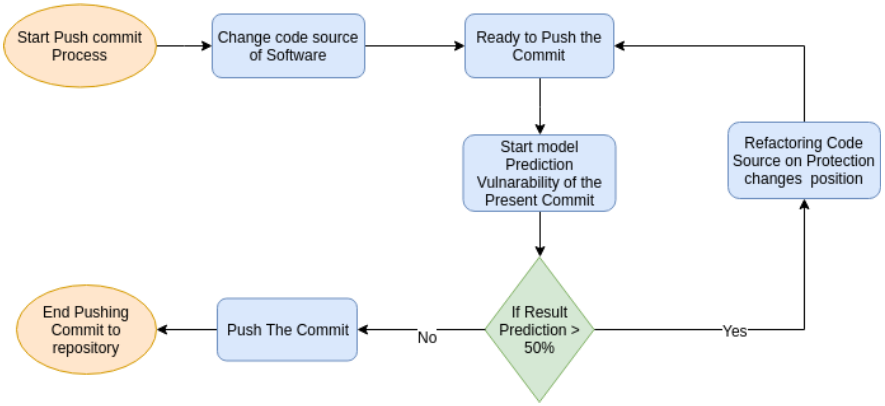

Start Push commit Process
Change code source of Software
Ready to Push the Commit
Start model Prediction Vulnarability of the Present Commit
Refactoring Code Source on Protection changes position
If Result Prediction > 50%
Yes
No
Push The Commit
End Pushing Commit to repository

Figure 1.2 Just In Time prediction processing of Pushing Commits

Previous studies on definite protection differences used an automated Pattern-Traversal Flow Analysis (PTFA) which is a static analysis and operated at a release granularity (Laverdière et Merlo, 2017; Laverdière et Merlo, 2017b; Laverdière et Merlo, 2018). These analyses required about 10 and 17 minutes on average per release pair for WordPress and MediaWiki, respectively.

1.1 Research Objectives

In this thesis, we use a dataset of protection-impacting changes computed using Pattern Traversal Flow Analysis (Gauthier et Merlo, 2012) in 211 release pairs of WordPress and 193 release pairs of MediaWiki. We build machine learning models on commits that touched on protection impacting changes and that affected the attribution of privileges in the code, then we perform a qualitative analysis to determine the characteristics of misclassified commits. Figure 1.3 presents an overview of our methodology. First, we gather releases from both studied projects including WordPress and MediaWiki. Then, we compose releases of each project into pairs to study the protection impacting changes (PIC) that occurred between each pairs. We used the tool presented in (Laverdière et Merlo, 2018) which reports the lines that have the PIC. Those lines are considered as the oracle of protection impacting changes lines. Finally, using a Python script, we were able to find the responsible commit that introduced the PIC lines. We compute commit log metrics and then used the Understand software analysis tool (scitool, 2017) to compute complexity of the code metrics. Moreover, we used computed metrics as input for the machine learning algorithms, to create predictive models capable of detecting suspicious commit, that could cause vulnerabilities issues. Then, we analyze the effectiveness of our models by conducting a qualitative analysis studying the misclassified (false negative and true negative) commits. Finally, using the obtained results, we answer the following research questions:

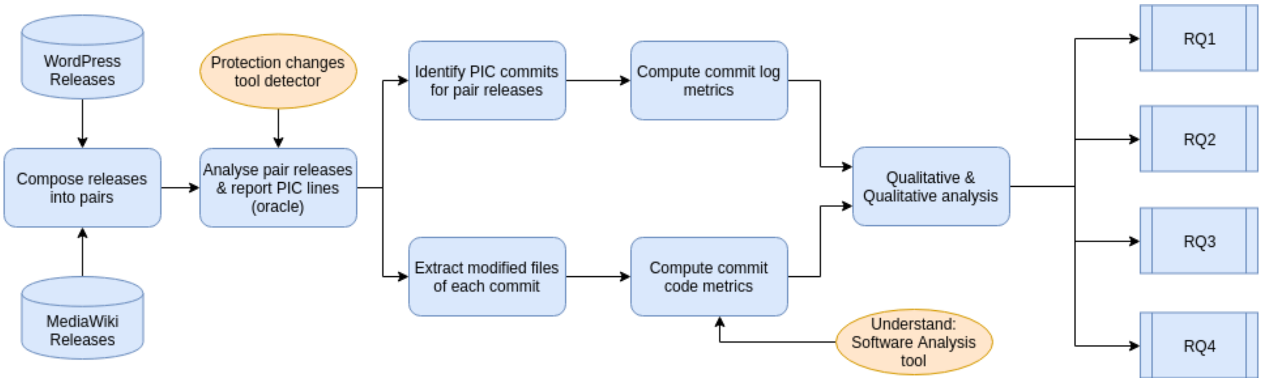

WordPress Releases
Protection changes tool detector
Identify PIC commits for pair releases
Compute commit log metrics
Compose releases into pairs
Analyse pair releases & report PIC lines (oracle)
Qualitative & Qualitative analysis
Extract modified files of each commit
Compute commit code metrics
MediaWiki Releases
Understand: Software Analysis tool
RQ1
RQ2
RQ3
RQ4

Figure 1.3 Overview of our Research Approach

RQ1: *What is the proportion of protection-impacting changes in Wordpress and MediaWiki?* We analyzed protection-impacting changes in two repositories including WordPress and MediaWiki. We used the Role Based Access Control (RBAC) (Sandhu *et al.*, 1996) approach to determine PIC lines in the different releases. We found that privilege protections were impacted by changes in 58% (123 / 211) of WordPress studied releases pairs and 77% (149 / 193) of MediaWiki studied releases pairs. We performed an empirical study to identify PIC commits occurrences from the source code repositories of the studied systems. We found that PIC commits account for 62% and 59% of commits in WordPress and MediaWiki, respectively.

RQ2: *What are the characteristics of protection-impacting changes?*

By examining the characteristics of PIC commits and other commits (i.e., commits that did not affect privilege protections), we observed that, PIC commits are submitted by developers with higher experience, in general. Besides, developers are likely to change fewer files, there are less inserted and deleted lines in PIC commits. They tend to implement more complex source code in PIC commits with the high number of functions, the number of declarative statements, high nested level of control functions, the more cyclomatic complexity of nested functions and more comment ratio. The risk of faults is higher in PIC commits.

RQ3: *To which extent can we predict protection-impacting changes?*

We used GLM, Naive Bayes, C5.0, and Random Forest algorithms to predict whether a commit is a PIC. Our predictive models reach the precision of 73.8% and recall up to 87.6% in WordPress. In MediaWiki, we obtained the precision of 77.2% and recall of 96%. Software organizations can apply our proposed techniques to identify PIC early on (i.e., as soon as they are introduced in the code repository) before they are exploited by malicious users.

RQ4: *Why do automatic machine learning models misclassify some protection-impacting changes?*

After a qualitative analysis, we observed that false positive and false negative PICs are due to the commits related to documentation or those that changed a version field (for WordPress, in `version.php` and in `DefaultSettings.php` for MediaWiki). Some other wrongly classified commits featured changes in the embedded HTML, JavaScript or CSS code.

1.2 Thesis Plan

The thesis is organized as follows. In chapter2, we present the background information about Pattern Traversal Flow Analysis and the detection of protection-impacting changes. Chapter3 discusses the related works. Chapter4 explains the design of our case study. Then, we answer the defined research questions and present the results in chapter5. Chapter6 discusses the threats to validity of our study. And finally, we conclude our thesis in chapter7.

CHAPTER 2 BACKGROUND

In this chapter, we provide background information about Pattern Traversal Flow Analysis and explain how we detect definite protection differences in this thesis.

2.1 Pattern Traversal Flow Analysis

Pattern Traversal Flow Analysis (PTFA) (Gauthier et Merlo, 2012; Francois et Ettore, 2012; Letarte et Merlo, 2009) is an automated whole-program static analysis. This model-checking approach verifies Boolean properties. Thus, PTFA verifies the property satisfaction that a predicate is true on all paths reaching a statement s. In our case, we use PTFA for code patterns pertaining to privilege verification for privilege $priv \in Privileges$. PTFA verifies the property satisfaction over the application's Control Flow Graph (CFG) $CFG = (V, E)$, where V is the set of vertices and E is the set of edges.

A PTFA engine creates model checking automata from the CFG and computes the graph reachability from the starting node v_0. PTFA automata are predicate-context-sensitive, meaning that security contexts are distinguished in the interprocedural analysis, and that equivalent contexts are merged. PTFA models have up to four states for each CFG vertex and privilege. These states encode the property satisfaction of the local and calling contexts and may be understood as being either protected or unprotected states, with regards to privilege $priv$. The PTFA model construction algorithm builds a model with many unreachable states and transitions. To facilitate reasoning, we simplify the models by retaining only reachable states and transitions. This model is called the *reachable PTFA model.*

2.2 Definite Protection Differences

When a privilege $priv$ is verified on all paths leading to CFG vertex v, we consider that v is *definitely protected* for $priv$. We can determine definite protection using the existence of states in reachable PTFA models. If v is definitely protected for privilege $priv$, then at least one protected state for v_i and no unprotected state for v_i exists in the model for $priv$. If an unprotected state is reachable in the automaton for $priv$, then $priv$ is not verified in at least one path to v_i.

Definite Protection Differences. When comparing two versions of an application (Ver_a and Ver_b), Definite Protection Differences (DPD) may occur for code that is shared by the two versions (Laverdière et Merlo, 2017b). When a statement s is common to releases Ver_a and

Ver$_b$, a DPD occurs when the definite privilege protection for s differs between *Ver*$_a$ and *Ver*$_b$.

Statement s is *loss-affected* when s is definitely protected by privilege *priv* in *Ver*$_a$, but is no longer so in *Ver*$_b$ – meaning that there exists at least one unprotected path to s in *Ver*$_b$. Conversely, s is *gain-affected* when s is not definitely protected by privilege *priv* in *Ver*$_a$, but is so in *Ver*$_b$. We use the term *security-affected* to refer to vertices that are either gain-affected or loss-affected.

2.3 Protection-Impacting Changes

Protection-Impacting Changes (PIC) Laverdière et Merlo (2018) are determined using code changes and graph reachability in reachable PTFA models.

$$deletedState(q_{i,j,k}) \doteq i \notin dom(vertexMap) \;\lor\; q_{vertexMap(i),j,k} \notin Q_b \tag{2.1}$$

$$addedState(q_{i,j,k}) \doteq i \notin image(vertexMap) \;\lor\; q_{bMap(i),j,k} \notin Q_a \tag{2.2}$$

$$\begin{aligned}&deletedEdges \doteq \\ &\left\{ \left(q_{i_1,j_1,k_1} \;,\; q_{i_2,j_2,k_2} \right) \in T_a \;\middle|\; \begin{array}{l} deletedState(q_{i_1,j_1,k_1}) \;\lor\; deletedState(q_{i_2,j_2,k_2}) \;\lor \\ \left(\; q_{vertexMap(i_1),j_1,k_1} \;,\; q_{vertexMap(i_2),j_2,k_2} \; \right) \notin T_b \end{array} \right\}\end{aligned} \tag{2.3}$$

$$\begin{aligned}&addedEdges \doteq \\ &\left\{ \left(\; q_{i_1,j_1,k_1} \;,\; q_{i_2,j_2,k_2} \; \right) \in T_b \;\middle|\; \begin{array}{l} addedState(q_{i_1,j_1,k_1}) \;\lor\; addedState(q_{i_2,j_2,k_2}) \;\lor \\ \left(\; q_{bMap(i_1),j_1,k_1} \;,\; q_{bMap(i_2),j_2,k_2} \; \right) \notin T_a \end{array} \right\}\end{aligned} \tag{2.4}$$

$$\begin{aligned}&partialPIC(changes, i, k) \doteq \; changes \cap \\ &(ReachableEdges(q_{i,0,k}) \cup ReachableEdges(q_{i,1,k}))\end{aligned} \tag{2.5}$$

$$\begin{aligned}&PIC(v_a, v_b) \doteq \\ &\left(\begin{array}{l} partialPIC(deletedEdges, v_a, 1) \cup partialPIC(deletedEdges, v_a, 0) \;, \\ partialPIC(addedEdges, v_b, 0) \cup partialPIC(addedEdges, v_b, 1) \end{array} \right)\end{aligned} \tag{2.6}$$

Code changes are reflected in PTFA models as added and deleted edges. $deletedEdges$ is the set of deleted transitions in the reachable PTFA model for Ver_a. Similarly, $addedEdges$ is the set of added transitions in the reachable PTFA model for Ver_b. Each set contains all transitions which connect one or more deleted or added state. It also contains all transitions for which no corresponding edge is present in the PTFA model of the other version. These equations depend on the following symbols. Q_i is the set of states in the reachable PTFA model for version Ver_i. T_i is the set of transitions in the PTFA model for version Ver_i. PTFA states are represented as $q_{i,j,k}$, where i is the CFG vertex identifier, j is the property satisfaction flag in the calling context, and k is the property satisfaction flag in the local context. The predicate $deletedState(q_{i,j,k})$ (Equation 2.1) is true whenever the state $q_{i,j,k} \in Ver_a$ has no corresponding state in Ver_b. Likewise, the predicate $addedState$ (Equation 2.2) is true whenever the state $q_{i,j,k} \in Ver_b$ has no corresponding state in Ver_a. This correspondence depends on the injective function $vertexMap$, which associates vertices in Ver_a to their corresponding vertices in Ver_b. The function $bMap$ is its reverse function. The symbols dom and $image$ correspond to the function's domain and image, respectively. In addition, protection-impacting changes belong to paths between the start of the program and security-affected vertices, on appropriately protected paths. For loss-affected code, the protection-impacting changes are the $deletedEdges$ 2.3 belonging to positively-protected paths to v_a and the $addedEdges$ 2.4 belonging to negatively-protected paths to v_b. For gain-affected code, the protection-impacting changes are similar, but with the protectedness reversed. For the sake of simplicity, and due to space constraints, we combine the definitions of protection-impacting changes for all security-affected code into the definition of PIC (Equation 2.6). In Equation , $ReachachableEdges(q_{i,j,k})$ is a function returning all edges between the initial state and the state $q_{i,j,k}$ in the reachable PTFA model. The $partialPIC$ function 2.5 combines changed edges (i.e. $addedEdges$ or $deletedEdges$) with reachable edges for either Ver_a or Ver_b. Finally in Equation 2.6, the vertices v_a and v_b respectively belong to the CFG of Ver_a and Ver_b and correspond to each other (i.e. $v_b = vertexMap(v_a)$).

2.4 Reporting Protection-Impacting Lines

Because it would be non-trivial to mine a software system in relation to edges in a control flow graph, we project our results over source code lines. This projection would also be easier to understand by end-users.

The results of a naive projection are likely to be problematic for end-users. Edges in $addedEdges$ and $deletedEdges$ may connect states belonging to unchanged lines of code. Users are likely to classify these as false positives, which would hurt psychological accept-ability. As such, we map PIC edges to source code locations, and retain only the locations that belong to changed code. To simplify the equations, we treat all code changes as either added or deleted code, similar to the `git` output. We define our line-projection functions in Equation 2.7. Function $PIC_{l,a}$ returns all protection-impacting code in Ver_a (i.e. deleted code). And $PIC_{l,b}$ does the same for version $Ver_b(i.e.foraddedcode)$. The line-projection functions rely on the projection functions $\pi_1(x)$ and $\pi_2(x)$, which respectively return the first and second index of the ordered pair x.

$$\begin{aligned} PIC_{l,a}(v_a, v_b) &\doteq \left(\bigcup_{(q_1,\ q_2) \in \pi_1(PIC(v_a,v_b))} \{srcLoc(q_1), srcLoc(q_2)\} \right) \cap deleted \\ PIC_{l,b}(v_a, v_b) &\doteq \left(\bigcup_{(q_1,\ q_2) \in \pi_2(PIC(v_a,v_b))} \{srcLoc(q_1), srcLoc(q_2)\} \right) \cap \ added \end{aligned} \tag{2.7}$$

2.4.1 Choice of pair releases and changes identification

Both of chosen projects are using a Software Configuration Management (SCM). One of the key features of modern SCM is the support of parallel lines of development known as branches. A branch is a virtual workspace forked from a particular state of the source code. It provides isolation from other changes where a developer or team of developers can make changes to the code in the branch without affecting others working outside the branch. There are conventional models of SCM development chosen by developers to fix their strategy of releases generation in a successive baseline (Walrad et Strom, 2002). Our strategy of choosing pairs is to avoid comparing branches in parallel, that are wildly different in their point of development interest. Using git repository graph, we compare new major and minor versions with the nearest releases to the points of bifurcation where a fork branch was created. We chose also series of releases in the same branch where developers test quality and fix bugs to prepare for a production release (Walrad et Strom, 2002).

CHAPTER 3 CRITICAL LITERATURE REVIEW

Our research project concerns two areas of software engineering research including *mining software repositories (MSR)*, in particular just-in-time prediction, and *Secure Software Engineering*, specifically, protection impacting analysis and vulnerability analysis of web applications. This chapter provides a critical literature review of these topics.

3.1 Protection Impacting Analysis

There were few studies on privilege protection changes. They all used PTFA static analysis and were conducted by comparing releases. Letarte *et al.* (2011) conducted a longitudinal study of privilege protection over 31 phpBB releases. They extracted the intra-procedural Control Flow Graph (CFG) in all functions of the system, then they used graph rewrite rules to extract the several founded patterns of security models. The authors defined the security patterns as Property Satisfaction Profiles (PSP) which represent a fulfillment estimation of properties processed on the extracted models. Then, they examined the growth of the changes made in the computed PSP on security models in the different releases of the chosen Web application. The results show that the model evolution analysis permits to distinguish changes in security levels between successive releases. The proposed approach requires almost 1 minute execution time for parsing, extraction of security models PSP for each version. This application only used a binary distinction between the administrator and unprivileged users.

Laverdière et Merlo (2017a) defined definite protection differences (previously named privilege protection changes) for richer protection schemes than Letarte et al. (Letarte *et al.*, 2011). They conducted longitudinal studies over 147 release pairs of WordPress on the presence of privilege protection changes and their classification. The authors identified privilege protection changes from an inter-procedural control flow graph (CFG) when they differ between two releases. Then, they examined the code and decided the security privilege protection models of Web applications written in PHP utilizing Pattern Traversal Flow Analysis (PTFA). The proposed methodology needs almost 7 minutes per release pair. The results show that 0.1% of the privilege protection changes are related to the program size and 70.01% of privilege protection changes represent possible security violations. Even more, 0.30% of the code is affected by security changes.

Then in (Laverdière et Merlo, 2017), the authors used Pattern Traversal Flow Analysis (PTFA) to statically examine code inferred formal models. They implemented the ana-

lyzer and classifier in Java to handle PHP code using a JavaCC grammar. The objective was to compute the counter-examples of definite protection properties and privilege protection losses. The authors studied 147 release pairs of WordPress to compute privilege protections and changes occurred between releases. They computed counter-examples for a total of 14,116 privilege protection losses and they discovered spread in 31 release pairs. The analysis requires 13 minutes to detect privilege protection losses between two releases.
Laverdière et Merlo (2018) defined protection impacting changes using PTFA. They conducted a survey of 210 release pairs of WordPress. The obtained results show that only 41% of the release pairs present protection-impacting changes while the other releases do not contain changes that touch privilege protection. Even more, protection impacting changes represent more than the quarter of the total changed lines of code of the impacted release pairs. Overall, they found that across the investigated releases of WordPress that the amount of protection-impacting changes is with 10.89% of changed lines of code. The authors conclude that using protection-impacting changes, developers would only spend the time to validate a median of the quarter of code changes in security-affected release pairs.

Previous studies operate their analysis at the release pairs analysis granularity and report the PIC lines and take many minutes to complete. In our paper, we identified responsible commits that induced PIC lines and built predictive models for protection-impacting changes that operate at the commit granularity.

Protection impacting change analysis is conceptually similar to approaches that identify the cause of bugs during evolution, such as Buginnings (Sinha *et al.*, 2010). This tool identifies bug-introducing code changes. It computes differences between dependence graphs and may investigate multiple versions. The authors built a mechanism capable of identifying bug-introducing code changes. They analyzed dynamically the effects of bug-fix code changes on program dependencies. Considering the semantics of the code changes, they could identify which version of the program the bug was first induced in the code. The proposed approach is based on computing text differences by analyzing the impacts of changes on the data dependencies, control dependencies, and all relations to compute added, deleted, and modified statements between two versions in the system. Then navigating in reverse starting from the second version backing in history to identify the version in which a statement was deleted or touched in the source code. The study includes intra-procedural and inter-procedural dependencies to identify the bug region. In the intra-procedural case, they constructed the program-dependence graph for each changed procedure and they created a program-dependence graph (PDG) for each procedure. The PDGs contain nodes that represent statements in the procedure and edges that represent data and control dependencies surrounding the statements. In the inter-procedural case, they constructed a system-dependence graph (SDG) that represent

a set of PDGs that are connected at call sites via parameter and call edges. Each call site has as input the variables that may be referenced or changed in the called procedure and give as output the variables that may be modified by the called procedure. Finally, their approach is conducted to find the chronologically last version where the bug region was created, it means the last version that contains any dependence or statement in the bug region that did not exist in previous version.
A major difference between these tools and our work is that they rely on identifying bug appearance in version granularity, whereas our approach is based on commit granularity with predictive analysis.

3.2 Just-in-time Prediction

Traditional defect prediction techniques often use metrics from bug reports to identify fault-prone modules or severity of bugs. Such defect prediction techniques can help software developers to prevent defects to some extent. However, those techniques do not help developers to handle defects once introduced in the system. In other words, before a defect is definitely resolved, users have to deal with issues such as bad user experience, data loss, and–or privacy threats. Just-in-Time defect prediction techniques are designed to predict defects at the commit level; helping developers to locate and address defects right before a commit is submitted for integration in a version control system. The use of just-in-time systems is quite spread in the MSR literature. Kamei *et al.* (2013) used a variety of source code metrics based on software change such as the number of added lines, and the developer experience to predict defect prone commits. Their study was conducted on six open source and five commercial systems to improve Software Quality Assurance (SQA). The effectiveness of their models gives an average accuracy of 68 % and an average recall of 64 %. The authors used the SZZ algorithm (Śliwerski *et al.*, 2005) to combine information from the version archived with the bug tracking system. Moreover, they showed that developers only need 20 % of habitual effort to review changes when using their proposed model. However, they only identified 35 % of all defect-inducing changes, contrary to our method that achieves good recall (97-98%) and precision (72-73%).
Using just-in-Time defect prediction techniques, Fukushima *et al.* (2014) conducted an empirical study on 11 open source projects to evaluate the performance of Just In Time (JIT) cross-project models. The proposed technique helps developers avoid defects especially at the beginning of the project where they do not have enough data to run traditional models learned from the project history. They found that in most cases, JIT models in cross-project context perform better than within-project models. Moreover, they confirm that projects

with similar correlation values between predictors and the dependent metrics achieve better results in a cross-project context.
Misirli *et al.* (2015) studied high impact fix-inducing changes (HIFCs) on six large open source projects in order to determine the best indicators of fix-inducing changes. They proposed an approach that measures the implementation work that developers spend some time to fix the changes. They include a series of code and process factors at commit level such as the amount of churn, the number of files and the number of subsystems touched by developers during the fixing process of bug-inducing changes. The authors used the SZZ algorithm (Śliwerski *et al.*, 2005) to link the bug fix Id with the main changes which cause the bug. The proposed models are able to predict between 56 % and 77 % of HIFCs, with 16% of misclassification. They found that the added lines of code, the number of included authors that make a change, and the number of files modified during a change are the best indicators of HIFCs.

An et Khomh (2015a) studied the appearance of crash-prone in Mozilla system. They identified commits responsible for the crash by using the SZZ algorithm (Śliwerski *et al.*, 2005) to map the bugs to their related commits. They found that the proportion of bug commits accounted for 25.5% in the studied version control system. They also found that crash-inducing commits are the most touched commits even more they were submitted by less experienced developers. In addition, they found that crash-inducing commits fix a previous bug, and often, they lead to another bug. They studied also the changed type of crash-inducing commits *i.e.,*class, comment, control flow, etc ... and they found that crash-inducing commits contain more unique changed types. They created just-in-time predictive models using different algorithm *i.e.,*GLM, Naive Bayes, C5.0, and Random Forest and they achieve a precision of 61.4% and a recall of 95.0%. In their extension study (An *et al.*, 2018) they characterise commits that would lead to frequent crashes and they called it as highly impactful crash-inducing commits. They observe that bugs caused by highly impactful crash-inducing commits were less reopened by programmers and tend to be fixed by a single commit. After applying predictive algorithms they could attend a precision of 60.9% and a recall of 91.1% to predict highly impactful crash-inducing commits. They analyse the characteristics of misclassified commits, and they found that 22.8% of commits do not lead to crashes but still caused bugs and they observed that developers make a high proportion of renaming operations on the code of these commits and they explain that renaming operations could lead to crash a software system. This study is closely related to ours, they studied crash-inducing in the commit granularity. Then, they created just-in-time predicting models. After that, they studied the causes of misclassified commits. But, as far as we know, this is the first study that uses just-in-time prediction techniques on the identification of protection-impacting changes.

3.3 Vulnerabilities in Web Application

Vulnerability analysis is an important topic given the risk posed by software vulnerabilities to users, *e.g.,* sensitive information loss. Vulnerabilities are reported continuously in specific databases such as Common Vulnerabilities and Exposures (CVE) [1], National Vulnerability Database (NVD) [2], Bugzilla [3], etc. These systems provide a reference for publicly known security information vulnerabilities and exposures that was reported in software products. Li *et al.* (2017b) analyses open source vulnerability databases and studied the characteristics of the most occurred vulnerabilities in the software systems. Several studies focused on predicting and contending against their appearance. The main causes for their appearance are related to limited programming skills or lack of security mindfulness on the part of developers (Jovanovic *et al.*, 2006).

Scholte *et al.* (2012) performed an empirical analysis on a web security vulnerabilities report from (CVE) to explore whether nowadays developers are aware of the seriousness of web security issues evaluations. The study focused on SQL injection and cross-site scripting vulnerabilities. They found that most of the vulnerabilities were not related to any modern attacks strategies. They explained that developers pay more attention to implement correction against SQL injection and fail to take countermeasures against cross-site scripting vulnerabilities. The results show that the duration of validating foundational cross-site scripting vulnerabilities are longer than those of SQL injection, which means that there still exist the effects of cross-site scripting vulnerabilities that were submitted many years ago. Popular applications are more exposed to the higher number of announced vulnerabilities since popular applications are more targeted by attackers as it has more impact on potential victims. Finally, they argue that web developers partly failed to protect their software, which opens the doors for more research to investigate the causes of vulnerabilities appearance in applications. In our study, we propose a tool model that could help developers to identify their new modification that could threaten their system in a very specific type of vulnerabilities protection impacting changes (PICs). This study talked about the effort taken by engineer to correct SQLI and cross-site scripting which is the same thing as PIC vulnerabilities. To correct and found them it takes effort as well, but in our case, we build a model that could find the vulnerably (PIC) and flag it before its injection on code source.

Shar et Tan (2012) conducted an empirical study to prevent SQL injection and XSS attacks in the Web applications. This attack is a type of injection, in which a malicious scripts are

[1] https://www.cvedetails.com/

[2] https://nvd.nist.gov/

[3] https://www.bugzilla.org/

injected in websites. The authors chose three open source PHP-based web applications, then they identified and classified different natures of input sources of vulnerabilities using the control flow graph (CFG). They used WEKA [4] to apply three different prediction algorithms to assign each software sensitive sink to a class of the target attribute which represent a binary assignation of vulnerable or not-vulnerable cases. As a result, their models predicted over 85% of the studied attacks.

Gupta *et al.* (2015) proposed an approach to predict XSS vulnerabilities in web applications. They studied an open source dataset from (Bertrand STIVALET, 2015) which includes two categories of safe and unsafe vulnerabilities. The authors explored XSS vulnerabilities in which malicious users inject code to steal sensitive information. They considered that user-input information in the website is important to identify context-sensitive security vulnerabilities, since an output statement refers to the user input via constant HTML string exchange and generate dynamic response depending on the user input. To build predictive models, they used the WEKA tool (Peter Reutemann Eibe Frank et Trigg, 2018). They achieved an F-measure of 90.6 % and Accuracy of 92.6 %, using bagging classifier.

Spanos *et al.* (2017) assessed the severity level of vulnerabilities from the textual description and examined the power of description to interpret the vulnerability severity. They considered the description of the text as an independent variable and converted it into numerical data to predict two kinds of independent metrics, including exploitable metrics, and impact metrics. Exploitable metrics reflect the degree of easiness to exploit vulnerabilities, and impact metrics reflect the effect of the successive exploitation of a vulnerability. Then, they employed mining techniques, text analysis and classification methods *e.g.,*decision trees, neural networks and used founded metrics to compute a score that determines the vulnerability severity. The results show that using textual description, could enable identifying vulnerability severity levels with high accuracy.

Chowdhury et Zulkernine (2011) used complexity, coupling, and cohesion metrics to predict vulnerabilities prone files in fifty-two releases of Mozilla Firefox. The authors extracted the vulnerabilities that occurred in the studied systems from Mozilla Foundation Security Advisories (MFSA). They used Bugzilla to track the responsible changes causing each reported vulnerability. After that, they computed CCC metrics (Complexity, Coupling, and Cohesion). Then, they applied different statistical techniques (C4.5 Decision Tree, Random Forests, Logistic Regression, and Naïve-Bayes). The results report a detection rate of the vulnerability-prone files equal to 75% . Shin *et al.* (2011) also used the code complexity, code churn, and developer activity metrics to discriminate affected vulnerable files from the

[4] https://www.cs.waikato.ac.nz/ml/weka/

safe ones. They found that complex code programs are more likely to contain vulnerabilities. Their model was able to predict 80% of vulnerable files.

There is a common point between previous studies and ours, both are predicting web security vulnerabilities. They have used their methodology to analyse programs already created, then they applied their models to looks for vulnerabilities *i.e.,*XSS, SQLI. However, their models can only be applied after the vulnerability occurred and already affected a large population of users, which differs with our analysis where we study the apparition of protection impacting change and predicting them before their merge on the code source in the commit level (just-in-time), In such case, correction of the vulnerability will be easierface to find and fix it, since it still fresh in developer's brain.

CHAPTER 4 METHODOLOGY AND DESIGN

In this chapter, we describe the methodology followed in this thesis.

4.1 Case Study Design

In the following diagram presented in Figure 4.1 shows the different steps we used to compute PICs lines that we consider as an oracle of our analysis. First, We prepare all chosen versions, then determine protection-impacting lines of code. Second, we compute the interprocedural control flow graphs from the PHP source code of each version using a PHP front end. Then, we compute the PTFA models and obtain definite privilege protections. Afterwards, we compare the source code of the releases using GNU Diff. Using these code differences and the aforementioned information, we obtain the definite protection differences and protection-impacting changes. Please note that GNU Diff does not consider file renames. Instead, it reports these as being fully deleted and fully added.

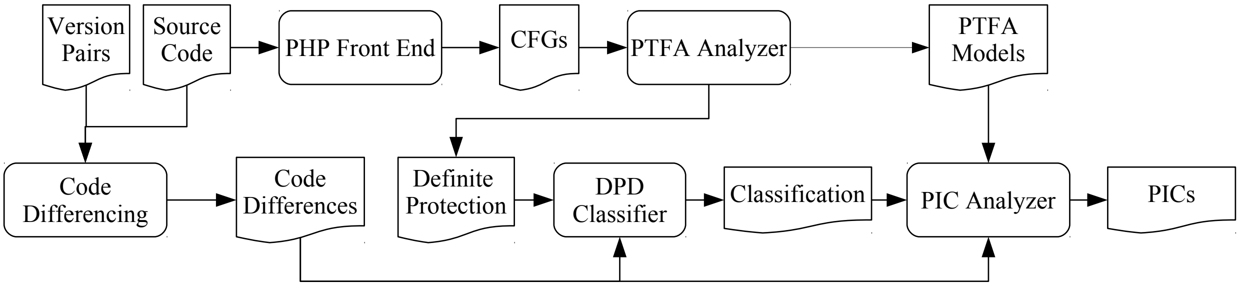


Figure 4.1 Processing Steps for Detecting Protection-Impacting Lines of Code Using PTFA

In this section, we describe the design of our case study which aims to answer the following research questions:

- RQ1: What is the proportion of protection-impacting changes in Wordpress and MediaWiki?
- RQ2: What are the characteristics of protection-impacting changes?
- RQ3: To which extent can we predict protection-impacting changes?
- RQ4: Why do automatic machine learning models misclassify some protection-impacting changes?

4.1.1 Collecting Data

Our study was performed on two open source projects: WordPress and MediaWiki. The analysis encompassed 211 release pairs of WordPress, from 2.0 to 4.7.3 and 193 release pairs of MediaWiki, from 1.5.0 to 1.29.2. We included all releases between the mentioned pairs. The full list of the studied release pairs are available in our data repository [1].

WordPress is a popular web-based content management system mainly implemented in PHP. It is a mature open source system with a long release history and its RBAC implementation and configuration are relatively simple. In terms of physical lines of code (LOC), WordPress's PHP code 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 a content management system from the MediaWiki Foundation. It is well-known thanks to its flagship user, Wikipedia. This application's PHP code ranges from roughly 149 KLOC to a peak of 1.35 MLOC. The combined HTML, JavaScript and CSS code is roughly between 4.5 KLOC and 188 KLOC.

Please note that, in this work, we only take into account the code changes written in PHP.

WordPress and MediaWiki maintains multiple releases in parallel and patches vulnerabilities for multiple versions simultaneously. As such, we organize release pairs in a tree according to their semantic versioning (Preston-Werner, 2013) and release date information. Each edge of that tree is a *release pair*. This approach was used in other RBAC evolution case studies before (Laverdière et Merlo, 2017; Laverdière et Merlo, 2017b; Laverdière et Merlo, 2018). In total, we found protection-impacting changes in 123 (58%) out of the 211 subject release pairs of WordPress and 149 (77%) out of the 193 subject release pairs of MediaWiki.

Since users typically deploy official releases, we performed the static code analysis on these releases only. We downloaded all subject releases from the official website of WordPress[2]. Since we downloaded the release archives, we have analyzed them as-is. There were two releases that were not published by WordPress on their site. In those cases, we extracted a path from their SVN repository and applied it on the previous release.

To ease the analysis, we performed our data mining against git repositories of both systems. For WordPress, we used a git-ified clone of the official WordPress SVN repository, hosted by GitHub[3]. For MediaWiki, we used the official git repository [4].

[1] https://github.com/AmineBarrak/PIC_Analysis

[2] `https://wordpress.org/download/release-archive/`

[3] `https://github.com/WordPress/WordPress`

[4] `https://gerrit.wikimedia.org/r/p/mediawiki/core.git`

Please note that there are discrepancies between the repository and the releases, as there are additional files in the release (e.g. extensions). As such, we ignored protection-impacting changes in these additional files.

4.1.2 Identifying Protection-Impacting Commits

Definite protection differences are due to code changes. These code changes are inserted through commits in the version control system. Our oracle is derived from the PICs reported by the PTFA-based tool. Protection-impacting lines of code can either be deleted or added code.

We identify protection-impacting commits as follows. Our PIC detection tool provides a list of protection-impacting lines of code for each release pair. These PICs are organized by file, line, and the type of definite protection difference (loss or gain).

If the tool detects a series of protection-impacting changes between a pair of releases (V_N and $V_{N'}$), it will output the following information for each of them:

```
N N' gain/loss F L
```

where F denotes the name of the file in which the privilege protection change occurred, while L denotes the specific line of the change and B the bifurcation point between V_N and $V_{N'}$.

Between V_N and $V_{N'}$, there often exist multiple commits. To find out the corresponding commit in which a privilege protection change occurred (either added or removed), we assume that the most recent commit before $V_{N'}$ commit that modified line L in the file F is the commit that introduced this privilege protection change. If the first commit in $V_{N'}$ is C', we apply the following command to identify the commit(s) that contains privilege protection changes:

```
git blame -L B..C' - F
```

Otherwise, we assume that a removed privilege protection line is based on the V_N version, we apply the following command to report commit that deleted a line:

```
git blame -reverse -L B..C' - F
```

As our analysis dataset, we consider only commits responsible for the reported lines that were modified in files between V_N and $V_{N'}$ which include Protection Impacting commits.

4.1.3 Computing Metrics

To capture the characteristics of protection-impacting changes, we compute the 16 metrics described in Table 4.2. We group the metrics in the two following categories:

Commit log metrics

We extract the following commit-related metrics to capture the structure of committed code. First, metrics related to date (*i.e.,* week day, month day and month), because protection-impacting changes may occur at specific dates (Castelluccio *et al.*, 2017). Second, we compute author experience to examine how commits or changes merged by less experienced developers impact privilege protections. Third, metrics related to commits size (*i.e.,* number of changed files, number of added and deleted lines), because Walden *et al.* (2009) found that there is usually a correlation between code change size and security issues. Fourth, we computed message size metrics, as Alali et al. Alali *et al.* (2008) found that commit message size is an indicator for maintenance activities.

We also identify commits related to bug fixing changes following the heuristic proposed by Sliwerski et al. (Śliwerski *et al.*, 2005). Using this information we compute the Boolean metric *Is bug fix* for each commit.

Code complexity metrics

For each studied commit, we use the Mercurial log command to extract all of its changed PHP files. Then, we apply the source code analysis tool *Understand* from Scitools [5] in order to collect complexity metrics. *Understand* provides a command line tool that helps to create a large number of project to analyse and to automate processing commits and metrics generation. We create a bash script to automate the extraction. We obtain seven code complexity metrics from Understand for the files in each subject commit, similar to An and Khomh (An et Khomh, 2015b). These metrics are lines of code (LOC), Cyclomatic complexity (also known as McCabe Cyclomatic complexity) which captures the occurrence of decision points in the code, number of functions, maximum nesting which is the level of controlling constructs in a function, number of declarative statements, number of blank lines, and ratio of comment lines over all lines in a file. For each commit (containing multiple files), we took the average of the metric values obtained for each file.

[5], https://scitools.com/

Table 4.1 Commit Log Metrics

Attribute	**Explanation and Rationale**
Week day	Day of week (from Mon to Sun). Code committed on certain week days may be less carefully written (*e.g.,* Friday) Śliwerski *et al.* (2005); Anbalagan et Vouk (2009).
Month day	Day in month (1-31). Code performed on certain days could be less delicately written (*i.e.,* end of months, before and during public holidays).
Month	Month of year (1-12). Code performed in some seasons may be less delicately written (*i.e.,* Christmas holidays, summer).
Message size	Words in a commit message. In **RQ2**, we found that PIC commits are correlated with longer commit messages.
Author experience	The number of prior submitted commits. In **RQ2**, we found that PIC commits tend to be submitted by less experienced developers.
Number of changed files *	Number of changed files in a commit. In **RQ2**, we found that commits with more changed files tend to have PIC apparition.
Number of added lines	Number of inserted lines in a commit. In **RQ2**, we found that commits with more added lines tend to have PIC apparition.
Number of deleted lines	Number of deleted lines in a commit. In **RQ2**, we found that commits with more deleted lines tend to have PIC apparition.
Is bug fix	Whether a commit aimed to fix a bug. In **RQ2**, we found that PIC commits are correlated with bug fixing code.

Table 4.2 Code Complexity Metrics

Attribute	**Explanation and Rationale**
LOC	Mean number of lines of code in all PHP files of a commit. In **RQ2**, we found that PIC commits have higher code churn (i.e., added/deleted lines).
Number of functions	Mean number of functions of all files in a commit. In **RQ2** we found that big functions may be difficult to understand or modify, and lead to PIC.
Cyclomatic complexity	Mean cyclomatic complexity of the functions in the all files of a commit. In **RQ2** we found Complex code is hard to maintain and may cause crashes.
Max nesting	Mean maximum level of nested functions in all files in a commit. In **RQ2** we found that highest nested functions correlate with PIC, even a high level of nesting increases complexity.
Number of declarative statements	Mean Number of declarative statements in a commit. In **RQ2** we found that highest declarative statements correlate with PIC
Number of blank lines *	Mean number of blank PHP lines of all files in a commit. In **RQ2** we found that, the more blank lines, the highest it leads to PIC commits
Comment ratio	Mean ratio of comment lines to code lines of all files in a commit. Codes with lower ratio of comments may be hard to understand, and may result in PIC

CHAPTER 5 CASE STUDY RESULTS

In this chapter, we report and discuss answers to our research questions. For each research question, we present the motivation, the approach followed to answer the question, and the obtained results.

RQ1: What is the proportion of protection-impacting changes in Wordpress and MediaWiki?

Motivation.

This question is preliminary to the remaining questions. It aims to examine the distribution of PICs in Wordpress and MediaWiki. The result of this question will help web software managers to realize the prevalence of PIC in projects. Allowing them to adjust their interventions when modifying the code that introduced security protection-impacting changes.

Approach. To compute the proportion of protection-impacting changes in the studied projects we conducted the PTFA analysis of PIC on releases pairs as described in Section 4.1.2. We identified the modified PIC lines and then searched for the commit that introduced the line. Finally, we computed the proportion of commits containing a PIC and the proportion of commits that do not contain any PIC.

Findings. Among the modifications of php files that occurred over 211 and 193 pair releases of WordPress and MediaWiki, we found 25069 commits in WordPress and 35864 commits in MediaWiki. Through our analysis of protection-impacting change detection (described in Section 4.1.2) we identified 62% (15700 / 25069) of PICs commits in WordPress and 59% (21335 / 35864) of PICs commits in MediaWiki. Figure 5.1 illustrates the proportion of protection-impacting changes commits and other commits.

Finding a PIC in a commit does not mean that all changed lines inside the commit represent a protection impacting changes.

Protection impacting change commits account for 62% in WordPress and 59% on MediaWiki, from the total number of studied releases.

Almost one out of every two commits is likely to contain PICs, which are at risk of introducing vulnerabilities (Woodraska *et al.*, 2011). Therefore, software developers should strive to catch PIC commits as soon as possible, e.g., when they are submitted into the version control system.

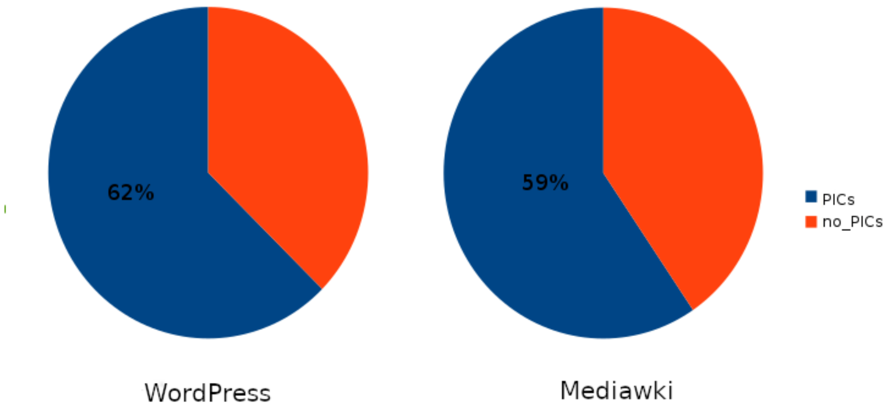


Figure 5.1 Proportion of Protection-Impacting changes commits in WordPress and MediaWiki

In the rest of this section, we will investigate the characteristics of commits that change privileges (i.e., PIC) and examine how to effectively predict them early.

RQ2: What are the characteristics of protection-impacting changes?

Motivation. In **RQ1**, we found that almost one in two commits contain protection-impacting changes. If developers fail to detect such changes and ensure the safety of the commit before integration into the code base, users risk suffering from security vulnerabilities. In this research question, we set out to investigate the characteristics of protection-impacting changes. The answer to this research question can help software practitioners to better differentiate protection-impacting changes from other changes.

Approach. We use the metrics shown in Tables 4.2 and statistically compare the 12 numerical variables in the following order: message size, authors experience, number of changed files, number of added lines, number of deleted lines, line of code, number of functions, cyclomatic complexity, maximum nesting, number of declarative statement, number of blank lines and comment ratio. We do so while partitioning the commits between PICs and non-PICs. If a commit contains more than one changed file, we compute the mean value of each metric on these files. For each of the 12 metrics (m_i), we formulate the following null hypothesis: H_i^0: *there is no difference between the values of* m_i *for the commits that contain at least one PIC and those that do not contain any*, where $i \in \{1, \ldots, 12\}$.

We use the Mann-Whitney U test (Hollander *et al.*, 2013) to accept or reject the 12 null hypotheses. This is a non-parametric statistical test, which measures whether two independent distributions have equally large values. We use a 95% confidence level (*i.e.,* $\alpha = 0.05$) to accept or reject these hypotheses. Since we perform more than one comparison on the same dataset, to control the familywise error rate, we use the Bonferroni correction (Dmitrienko *et al.*, 2005). Concretely, we divide our α by the number of tests, *i.e.,* $\alpha = 0.05/12 = 0.004$.

Whenever we obtain a statistically significant difference between the metric values, we compute the Cliff's Delta effect size (Cliff, 1993), which measures the magnitude of the difference while controlling for the confounding factor of sample size (Coe, 2002). We assess the magnitude using the threshold provided in (Romano et D. Kromrey, 2006), *i.e.,*, $|d| < 0.147$ “negligible"(N), $|d| < 0.33$ “small"(S), $|d| < 0.474$ “medium"(M), otherwise “large"(L).

In addition to the 12 mentioned metrics, we investigate the distribution of the bug fixes commits and the weekend churn (Saturday and Sunday) according to the partionning of PIC and non-PIC dataset.

Findings.

Table 5.1 and 5.2 summarise differences between the characteristics of commits that introduced protection-impacting changes (PIC) and others *i.e.,* commits that did not alter protection privileges (non-PIC), corresponding to the projects WordPress and MediaWiki. We show the median value of PIC and non-PIC for each metric, as well as the *p*-value of the Mann Whitney U test and the Cliff's Delta effect size. We observe that the commit message size of PICs is significantly longer than non-PICs commits messages sizes in both projects. It is possible that PICs commits are more complex and consequently developers need extended comments to describe these changes. According to the results of both projects, PICs are submitted by developers with more experience. This result could be explained by the fact that not all programmers have the ability to change sensitive code areas containing privilege protection lines. Another interesting finding is the fact that PICs tend to have higher code complexity in terms of number of functions, cyclomatic complexity, maximum nesting, number of declarative statements and comment ratio. These results are reinforced by the obtained medium and large effect sizes. Finally, most of our studied PICs commits are bug fixing operations. Woodraska *et al.* (2011) found that bugs can turn into severe security vulnerabilities which is consistent with our results.

In light of results from Table 5.1 we reject all the null hypotheses H_i^0. In other words, for all metrics listed in Table 5.1, there exist statistically significant differences between PICs and non-PICs commits in varying proportions. Appendix .1 and .2 represent more details about the characteristics of protection impacting changes.

Table 5.1 Median value of the characteristics of PICs and non-PICs as well as p-value of Mann-Whitney U test and effect size for WordPress project

Metric	**PIC**	**non-PIC**	**P-value**	**Effect size**
Message Size	19.66	15.61	<2.2e-16	0.144 (N)
Author experience	1531.98	1343.94	0.033	0.016 (N)
Number of changed files	3.18	2.60	<2.2e-16	0.127 (N)
Number of inserted lines	71.03	86.19	<2.2e-16	0.205 (S)
Number of removed lines	47.961	75.66	<2.2e-16	0.197 (S)
LOC	666.21	591.15	<2.2e-16	0.092 (N)
Number of functions	28.91	26.20	<2.2e-16	0.067 (N)
Cyclomatic complexity	9.93	7.97	<2.2e-16	0.054 (N)
Max nesting	3.65	3.46	<6.3e-15	0.057 (N)
Number of declarative statements	42.13	37.73	<2.2e-16	0.069 (N)
Number of blank lines	160.13	139.25	<2.2e-16	0.105 (N)
Comment ratio	0.915	0.964	0.002	0.023 (N)
Is bug fix	58.7%	57.6%	–	–
Weekend churn	20%	21.7%	–	–

Overall, we found significant differences between PIC and the non-PIC commits on all studied characteristics. PICs commits are submitted by experienced developers. They contain longer commit messages and make complex changes in files.

RQ3: To which extent can we predict protection-impacting changes?

Motivation. On the one hand, leaving unintentional privilege protection changes in the source code may lead to security vulnerabilities that severely affect end users. On the other hand, identifying unintentional privilege protection changes from each new code change is a non-trivial task. Although the PTFA-based analysis can scan a PHP system quickly (about 10 and 17 minutes on average per release pair, respectively for WordPress and MediaWiki), it is still impractical to perform such analysis for each of the code changes because developers may submit hundreds of code changes daily for a large-scale system. One feasible way to remind developers a security warning in a real-time manner is to build just-in-time prediction models. In our case, such models can be trained using historical data and predict whether a new code change (such as a commit or a changed file) contains PIC(s) or not. Previous studies, including (Kamei *et al.*, 2013; Fukushima *et al.*, 2014), showed that just-in-time prediction models can help software practitioners better focus their efforts on debugging fault-inducing changes, which can reduce code reviewing and testing efforts as well as prevent them from delivering defects to end users. In this research question, we examine the possibility of

Table 5.2 Median value of the characteristics of PICs and non-PICs as well as *p*-value of Mann-Whitney U test and effect size for MediaWiki project

Metric	**PIC**	**non-PIC**	**P-value**	**Effect size**
Message Size	27.06	18.10	<2.2e-16	0.255 (S)
Author experience	1378.77	1012.47	<2.2e-16	0.093 (N)
Number of changed files	3.505	9.014	<2.2e-16	-0.055 (N)
Number of inserted lines	83.18	293.13	2.5*10e-4	-0.022 (N)
Number of removed lines	59.588	220.842	2.5*10e-3	-0.018 (N)
LOC	739.87	1399.06	<2.2e-16	-0.26 (S)
Number of functions	37.17	18.69	<2.2e-16	0.47 (M)
Cyclomatic complexity	2.47	2.23	<2.2e-16	0.056 (N)
Max nesting	3.31	1.66	<2.2e-16	0.48 (L)
Number of declarative statements	57.37	27.80	<2.2e-16	0.48 (L)
Number of blank lines	130.49	166.63	<2.2e-16	-0.134 (S)
Comment ratio	0.528	0.503	<2.2e-16	0.284 (S)
Is bug fix	29.4%	23.4%	–	–
Weekend churn	18.5%	17.3%	–	–

using just-in-time prediction models to identify unintentional privilege protection changes in real-time.

Approach. We use the metrics from Table 4.2 as independent variables to build statistical models. All of these metrics are extracted at commit level as it is required for a just-in-time prediction model. Our prediction target (*i.e.,* dependent variable) is whether a new commit contains at least one PIC or not. We apply four different machine learning algorithms: **General Linear Model (GLM)**, **Naive Bayes**, **decision tree**, and **Random Forest**. General Linear Model is an extension of multiple linear regression and enables the analysis of a binary classification problem, *i.e.,* in our case, whether a commit contains PIC(s). Although this algorithm is extensively used in classification analyses, it may not achieve a good fitness when there is no smooth linear decision boundary in the dataset. In this work, we use this algorithm as the baseline to assess the effectiveness of other models. Naive Bayes are a set of logistic regression algorithms based on the Bayes' theorem (Vapnik et Vapnik, 1998) with strong independence assumptions between the features. This algorithm often obtains, in practice, a good classification result (Rish, 2001). Compared to General Linear Model, decision tree does not assume a linear relationship between variables and it can also implicitly perform variable screening or feature selection. Thus, decision tree is expected to obtain a high prediction accuracy. Shihab *et al.* (2013) used the C4.5 decision tree algorithm to predict re-opened bugs and got a good results. In this study, we used C5.0 model, the improved version of C4.5, which can obtain a better accuracy, perform faster, and have less memory usage than C4.5 (Research, 2017). To further mitigate the biases and variance from the decision tree model, Leo Breiman and Adele Cutler introduced Random Forest (Breiman, 2001), which takes a majority voting of decision trees to generate classification (predicting often binary class labels) or regression (predicting numerical values) results. In previous just-in-time prediction studies, *e.g.,* (An *et al.*, 2017), Random Forest achieved the best prediction accuracy. In this study, we build 1,000 trees, each of which are with 5 randomly selected metrics.

To reduce the multicollinearity from the dataset, we use the Variance Inflation Factor (VIF) technique to remove correlated metrics before building the models. As recommended by (Rogerson, 2010), we remove the metrics whose VIF values are greater than or equal to 5. In Tables 4.2, the removed metrics are marked with *.

We apply ten-fold cross validation (Efron, 1983) to measure the fitness of the models using R script `https://github.com/swatlab/crash-inducing`. The efficiency of the proposed model is calculated based on general parameters as described in (Ding et Li, ????), We reported respectively the general accuracy, as well as the precision, recall to report , and F-measure on the commits or change files that contain PIC(s) for each model.

The simplest measure is the **accuracy**. It is defined as the ratio of the number of instances which are correctly classified to the total number of the instances in test set. Its equation is defined as follows,

$$Accuracy \doteq \frac{(TN + TP)}{(TN + FP + FN + TP)} \tag{5.1}$$

The measure, **precision** known as **positive predictive value**, is defined as the ratio of the number of correctly classified positive instances to the total number of instances that are classified into positive class. So the precision can be calculated by,

$$Precision \doteq \frac{TP}{TP + FP} \tag{5.2}$$

The measure, **recall** known as **sensitivity**, is defined as the ratio of the number of correctly classified positive instances to the total number of positive instances. So, recall can be calculated by,

$$Recall \doteq \frac{TP}{(TP + FN)} \tag{5.3}$$

Based on precision and recall, we can get a comprehensive measure **F-measure** which is the measure of a test's accuracy, which is defined as,

$$F - measure \doteq \frac{(2 * Recall * Precision)}{(Recall + Precision)} \tag{5.4}$$

In the above definitions, True Positive (TP) is the number of positive cases classified correctly, False Negative (FN) is the number of positive cases classified as negative, False Positive (FP) is the number of negative cases classified as positive, and True Negative (TN) is the number of negative cases classified correctly (Huang *et al.*, 2012).

In the cross validation, we randomly split the subject commits into ten disjoint sets. Nine of them are used as training data and the remaining one as testing data. We repeat this process for ten times and report mean results for accuracy, precision, recall, and F-measure. In the dataset, the commits containing PICs account for 62% and 59% corresponding to WordPress and Wikimeda, which can lead to biases and inaccuracy in the results (Kamei *et al.*, 2007). To deal with this, we perform a combination of over- and under-sampling using the R `ovun.sample` package. In the training sets, instances in the majority category (*i.e.,* commit or changed file without PICs) will be randomly deleted and the minority category (*i.e.,* commit or changed file with PICs) will also be randomly boosted, until the number of the instances in both categories achieve the same level. In addition to reporting the fitness of the models, we will rank the impact of the independent variables to identify the top predictors of the algorithm that obtains the best prediction results.

Findings. Table 5.3 shows the median accuracy, precision, recall, and F-measure for the four algorithms used to predict whether a commit contains protection-impacting changes in WordPress and MediaWiki systems. According to the results, our models can predict PIC commits in WordPress system with a precision up to 73.8% and a recall up to 98.3%. In MediaWiki we achieve a precision of 77.2% and a recall of 97.8%. Random Forest is the best prediction algorithm. It achieves the best F-measure when predicting PICs commits.

Our predictive models can achieve a precision of 73.8%, and a recall of 87.6% in Wordpress. In MediaWiki, models achieve a precision of 77.2%, and a recall of 96%. The Random Forest algorithm achieves the best prediction performance in both projects. Closeness is ranked as the best predictor in this algorithm. Software organizations can use the proposed predictive models to catch protection impacting change commits just in time as soon as they are submitted for integration in the repository, e.g., during code review.

Table 5.3 Accuracy, precision, recall and F-measure obtained from GLM, Naive Bayes, C5.0, and Random Forest when predicting protection-impacting changes in WordPress repository

Metric	**GLM**	**Bayes**	**C5.0**	**Random Forest**
Fitting models using WordPress repository				
Accuracy	64.2%	62.0%	68.8%	72.5%
PIC precision	64.8%	62.6%	71.0%	73.8%
PIC recall	94.1%	98.3 %	84.7%	87.6%
PIC F-measure	76.8%	76.4%	77.4%	79.8%
Fitting models using MediaWiki repository				
Accuracy	74.3%	62.7%	78.6%	80.9%
PIC precision	73.7%	61.7%	71.0%	77.2%
PIC recall	88.3%	97.8 %	84.7%	96.0%
PIC F-measure	80.3%	76.4%	83.9%	85.6%

RQ4: Why do automatic machine learning models misclassify some protection-impacting changes?

Motivation. In the previous research question, even though the models achieved a good performance, there is still a percentage of the clean commits (respectively changed files) that were classified by our models as commits (respectively changed files) with PICs, which we refer to as false positives. In addition, certain commits (respectively changed files) with PICs were wrongly classified as clean commits (respectively changed files); we refer to them as to false negatives. In this research question, we want to understand the reasons behind these false positives and negatives. The answer of this question may help us to discover further hidden factors that are related to unintentional protection impacting changes and to improve our current predictive models.

Approach. We performed a qualitative analysis of false positives and false negatives to search for the main causes of the wrong classification of the predictive model. In WordPress, using Random Forest, 509 commits were false negatives, and 239 commits were false positives. In MediaWiki, We obtained 580 false negative commits and 110 false positives commits with Random Forest (our best performing classifier).

To understand the characteristics of the misclassified commits, we randomly took a sample of them with a margin error of 10% and a confidence level of 95%.

For WordPress, our sample was of 83 / 509 false negatives and 69 / 239 false positives.

Findings. We summarize our observations in Table 5.4. Overall, we observed that many wrongly classified commits changed a version field (for WordPress, in `version.php` and in `DefaultSettings.php` for MediaWiki). If we leave out changes to version fields, some commits had only documentation changes. Some other commits featured changes in the embedded HTML, JavaScript or CSS code. While these changes were within a `.php` file, they are not PHP code *per se*. Changes to non-PHP code or to documentation cannot be protection-impacting by definition. We also observed that a minority of wrongly classified commits added control flow branches (conditions or loops), which may in turn have affected some of the observed metrics. Some commits were merge commits due to their strategy of continuous integration development, others were the addition of profiling information, and a few were clearly related to the RBAC implementation. There is 18 / 19 in WordPress and 19 in MediaWiki that were only about documentation or contained whitespaces only; these commits are clearly non-PIC but were predicted as PIC.

Table 5.4 Qualitative Observations Over Wrongly Classified Commits

Observation	**WordPress**		**MediaWiki**	
Samples	FP (69 / 239)	FN (83 / 509)	FP (52 / 109)	FN (84 / 579)
Modified a version field	23	13	1	0
Documentation or whitespace only	1	18	0	19
Changes to embedded HTML/JS/CSS	4	6	0	0
Branches added or deleted	13	20	8	17
Similar in/out	50	44	31	33
Merge commits	0	0	6	4
Profiling	0	0	2	0
RBAC-Related	0	0	2	0

These observations lead us to methodological improvements to consider in future research.

First, per-project rules defining code changes to ignore should be added. For instance, all changes to `version.php` in WordPress should be ignored. Future research in repository mining for PICs should ignore documentation-only commits altogether, and possibly strip comments from the code changes analyzed. In addition, a per-project whitelist of internal APIs known to have no impact on privilege protection (e.g. profiling calls) could be used to filter out changes further.

CHAPTER 6 THREATS TO VALIDITY

We now discuss in this chapter the threats to the validity of our study following the guidelines for case study research (Yin, 2002).

6.1 Threats to internal validity

Threats to internal validity are factors that may influence our independent variables and that were not taken into account. Our results depend on the accuracy of the PTFA engine that we used. This engine relies on sound but conservative approximations for dynamic features common to PHP applications. These approximations may lead to spurious paths and thus spurious protection-impacting changes. The reported spurious path rate for PTFA is 10.96 ± 3.18% (95% confidence level) (Laverdière et Merlo, 2017).

Our results also depend on the source differencing tools we used. We used GNU `diff` to extract line-level differences between releases. This causes some imprecision in the vertex mapping between releases, and some vertices may be inaccurately considered changed. However, this should not affect much our results, since the output of the security-impacting change detection tool, and the rest of the analysis is also at the line granularity. To validate the detected protection-impacting changes, we randomly sampled 100 commits that are considered as PICs. We analyzed the corresponding changed lines in these commits and observed that many commits are exact protection-impacting changes, which provides us confidence on the accuracy of our detection results.

6.2 Threats to conclusion validity

Threats to conclusion validity are concerned with the relationship between the treatments and the outcome. We paid attention to not violate the assumptions when performing statistical analyses. In **RQ2**, we only used non-parametric tests (including Mann-Whitney U test and Cliff's Delta effect size) that do not require making assumptions on the distribution of our dataset. To mitigate the familywise error rate in our null hypotheses, we used the Bonferroni correction to calculate an adjusted *p*-value for each subject characteristic. When building statistical models, we applied variance inflation factor (VIF) to remove multicollinearity among the independent variables.

6.3 Threats to external validity

Threats to external validity affect the generalizability of our results. Despite the fact that our approach may leverage vulnerability oracle, we did not have access to one for our study. Consequently, we cannot study protection-impacting changes specifically for vulnerabilities and tune a model specifically for them. 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 conducted on two open source content management systems implemented in PHP WordPress and MediaWiki. We may obtain different results when studying other systems. We may also obtain different results for systems in other languages. Our approach itself is reproducible and language-independent, although the PTFA engine we used only handles PHP at the moment. Consequently, our conclusions depend on the change history of this single system. To counter this issue, studies that include other systems, and systems in other languages, should be performed.

CHAPTER 7 CONCLUSION

This chapter concludes our work and outlines some avenues for future works.

7.1 Summary of the Results

In this thesis, in order to enable just-in-time identification of commits that cause privilege protection changes, we conducted an analysis of protection-impacting changes across 211 release pairs of WordPress and 193 release pairs of MediaWiki. We observed that around 60% of commits submitted in the code repositories of these systems affected privileges protections. To help developers identify these changes early on before they are integrated in the code, we extracted a series of metrics from commit logs and source code, and built statistical models. The evaluation of these models showed that they can achieve a precision up to 73.8% and a recall up to 87.6% in WordPress and for MediaWiki, a precision up to 77.2% and recall up to 96%. Among the metrics that we examined; commit churn, bug fixing, author experiences and code complexity between two releases were the most important predictors in the models. A qualitative analysis of the false positives and false negatives of the models revealed that they are mostly due to documentation-only commits. A minority of wrongly classified commits added control flow branches (conditions or loops), which may in turn have affected some of the observed metrics.

Our approach does not replace security reviews nor does it remove the need to use a PTFA-based protection-impacting change detector. However, it may complement these approaches in a synergic manner and greatly reduce the number of code changes that need to be reviewed for protection impacts at a later stage of the software development process.

7.2 Future Work

In future work, we would like to expand our study to more systems, written in both PHP and other languages. We also plan to conduct usability studies with professional developers to further assess the usefulness of our proposed method. We would like to quantify the savings in terms of review effort.

REFERENCES

Alali, Abdulkareem and Kagdi, Huzefa and Maletic, Jonathan I (2008). What's a typical commit? a characterization of open source software repositories. *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. IEEE, 182–191.

An, Le and Khomh, Foutse (2015a). An empirical study of crash-inducing commits in mozilla firefox. *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, New York, NY, USA, PROMISE '15, 5:1–5:10.

An, Le and Khomh, Foutse (2015b). An empirical study of highly impactful bugs in mozilla projects. *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*. IEEE, 262–271.

An, Le and Khomh, Foutse and Guéhéneuc, Yann-Gaël (2017). An empirical study of crash-inducing commits in mozilla firefox. *Software Quality Journal*.

An, Le and Khomh, Foutse and Guéhéneuc, Yann-Gaël (2018). An empirical study of crash-inducing commits in mozilla firefox. *Software Quality Journal*, *26*(2), 553–584.

Anbalagan, Prasanth and Vouk, Mladen (2009). Days of the week effect in predicting the time taken to fix defects. *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*. ACM, 29–30.

Bertrand STIVALET, Aurelien DELAITRE (2015). Php-vulnerability-test-suite: Collection of vulnerable and fixed php synthetic test cases. `https://github.com/stivalet/PHP-Vulnerability-test-suite`.

Breiman, Leo (2001). Random forests. *Machine learning*, *45*(1), 5–32.

Castelluccio, Marco and An, Le and Khomh, Foutse (2017). Is it safe to uplift this patch? an empirical study on mozilla firefox. *arXiv preprint arXiv:1709.08852*.

Cho, Yoon Ho and Kim, Jae Kyeong (2004). Application of web usage mining and product taxonomy to collaborative recommendations in e-commerce. *Expert systems with Applications*, *26*(2), 233–246.

Chowdhury, Istehad and Zulkernine, Mohammad (2011). Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, *57*(3), 294–313.

Cliff, Norman (1993). Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, *114*(3), 494.

Coe, Robert (2002). It's the effect size, stupid: What effect size is and why it is important.

Ding, Tienan and Li, Xiongfei (????). Classification of evaluation measurement taken predicted scores into account. *JOURNAL OF INFORMATION &COMPUTATIONAL SCIENCE*, *12*(5), 1723–1730.

Dmitrienko, A. and Molenberghs, G. and Chuang-Stein, C. and Offen, W.W. (2005). *Analysis of Clinical Trials Using SAS: A Practical Guide*. SAS Institute.

Efron, Bradley (1983). Estimating the error rate of a prediction rule: improvement on cross-validation. *Journal of the American Statistical Association*, *78*(382), 316–331.

Gauthier Francois and Merlo Ettore (2012). Fast detection of access control vulnerabilities in php applications. *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 247–256.

Fukushima, Takafumi and Kamei, Yasutaka and McIntosh, Shane and Yamashita, Kazuhiro and Ubayashi, Naoyasu (2014). An empirical study of just-in-time defect prediction using cross-project models. *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*. ACM, 172–181.

François Gauthier and Ettore Merlo (2012). Alias-aware propagation of simple pattern-based properties in PHP applications. *Proc. 12th IEEE Int'l Working Conf. Source Code Analysis and Manipulation (SCAM '12)*. 44–53.

Gupta, Mukesh Kumar and Govil, Mahesh Chandra and Singh, Girdhari (2015). Predicting cross-site scripting (xss) security vulnerabilities in web applications. *Computer Science and Software Engineering (JCSSE), 2015 12th International Joint Conference on*. IEEE, 162–167.

Oren Harel (2017). Plainid » blog archive role explosion: The unintended consequence of rbac. `https://www.plainid.com/2017/08/role-explosion-unintended-consequence-rbac/`. (Accessed on 09/13/2018).

Hollander, Myles and Wolfe, Douglas A and Chicken, Eric (2013). *Nonparametric statistical methods*. John Wiley & Sons.

Huang, Feixiang and Wang, Shengyong and Chan, Chien-Chung (2012). Predicting disease by using data mining based on healthcare information system. *Granular Computing (GrC), 2012 IEEE International Conference on*. IEEE, 191–194.

Idenhaus (2017). How to successfully introduce role based access control into a group environment | idenhaus consulting. `https://www.idenhaus.com/how-integrate-role-based-access-control-group-environment/`. (Accessed on 09/12/2018).

Jovanovic, Nenad and Kruegel, Christopher and Kirda, Engin (2006). *Pixy: A static analysis tool for detecting web application vulnerabilities (short paper)*. IEEE.

Kamei, Yasutaka and Monden, Akito and Matsumoto, Shinsuke and Kakimoto, Takeshi and Matsumoto, Ken-ichi (2007). The effects of over and under sampling on fault-prone module detection. *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*. IEEE, 196–204.

Kamei, Yasutaka and Shihab, Emad and Adams, Bram and Hassan, Ahmed E and Mockus, Audris and Sinha, Aloka and Ubayashi, Naoyasu (2013). A large-scale empirical study of just-in-time quality assurance. *Software Engineering, IEEE Transactions on*, *39*(6), 757–773.

Laverdière, Marc-André and Merlo, Ettore (2017a). Classification and distribution of rbac privilege protection changes in wordpress evolution (short paper). *2017 15th Annual Conference on Privacy, Security and Trust (PST)*. IEEE, 349–3495.

Laverdière, Marc-André and Merlo, Ettore (2017b). Classification and distribution of rbac privilege protection changes in wordpress evolution,”. *Proc. 15th Int’l Conf. Privacy, Security and Trust (PST’17)*.

Laverdière, Marc-André and Merlo, Ettore (2018). Detection of protection-impacting changes during software evolution. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 434–444.

Marc-André Laverdière and Ettore Merlo (2017). Computing counter-examples for privilege protection losses using security models. *Proc. 24th IEEE Int’l Conf. Software Analysis, Evolution, and Reengineering (SANER ’17)*. 240–249.

Marc-André Laverdière and Ettore Merlo (2018). Detection of protection-impacting changes during software evolution. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 434–444.

Letarte, Dominic and Gauthier, François and Merlo, Ettore (2011). Security model evolution of php web applications. *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 289–298.

D. Letarte and E. Merlo (2009). Extraction of inter-procedural simple role privilege models from PHP code. *Proc. 16th IEEE Working Conf. Reverse Engineering (WCRE '09)*. 187–191.

X. Li and J. Chen and Z. Lin and L. Zhang and Z. Wang and M. Zhou and W. Xie (2017a). A mining approach to obtain the software vulnerability characteristics. *2017 Fifth International Conference on Advanced Cloud and Big Data (CBD)*. 296–301.

Li, Xiang and Chen, Jinfu and Lin, Zhechao and Zhang, Lin and Wang, Zibin and Zhou, Minmin and Xie, Wanggen (2017b). A mining approach to obtain the software vulnerability characteristics. *2017 Fifth International Conference on Advanced Cloud and Big Data (CBD)*. IEEE, 296–301.

Misirli, Ayse Tosun and Shihab, Emad and Kamei, Yasukata (2015). Studying high impact fix-inducing changes. *Empirical Software Engineering*, 1–37.

Peter Reutemann Eibe Frank, Mark Hall and Len Trigg (2018). Weka 3 - data mining with open source machine learning software in java. `https://www.cs.waikato.ac.nz/ml/weka/`.

Piancó, Marcus and Fonseca, Baldoino and Antunes, Nuno (2016). Code change history and software vulnerabilities. *Dependable Systems and Networks Workshop, 2016 46th Annual IEEE/IFIP International Conference on*. IEEE, 6–9.

Marcus Pinto and Dafydd Stuttard (2011). *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. John Wiley & Sons.

Tom Preston-Werner (2013). Semantic Versioning 2.0.0.

RulesQuest Research (2017). Is c5.0 better than c4.5? `http://www.rulequest.com/see5-comparison.html`. (Accessed on 11/14/2018).

Rish, Irina (2001). An empirical study of the naive bayes classifier. *IJCAI 2001 workshop on empirical methods in artificial intelligence*. IBM New York, vol. 3, 41–46.

Rogerson, Peter A (2010). *Statistical methods for geography: a student's guide.* Sage Publications.

Romano, Jeanine and D. Kromrey, Jeffrey (2006). Appropriate statistics for ordinal level data: Should we really be using t-test and cohen's d for evaluating group differences on the nsse and other surveys?

Margaret Rouse (2018). What is role-based access control (rbac)? - definition from whatis.com. `https://searchsecurity.techtarget.com/definition/role-based-access-control-RBAC`. (Accessed on 09/29/2018).

Jonathan Sander (2009). Rbac and abac and roles, oh my. | identity sander. `https://identitysander.wordpress.com/2009/11/03/rbac-and-abac-and-roles-oh-my/`. (Accessed on 09/12/2018).

Sandhu, Ravi S and Coyne, Edward J and Feinstein, Hal L and Youman, Charles E (1996). Role-based access control models. *Computer*, *29*(2), 38–47.

Scholte, Theodoor and Balzarotti, Davide and Kirda, Engin (2012). Have things changed now? an empirical study on input validation vulnerabilities in web applications. *Computers & Security*, *31*(3), 344–356.

scitool (2017). Understand tool. `https://scitools.com`. Online; accessed October 30th, 2017.

Shar, Lwin Khin and Tan, Hee Beng Kuan (2012). Mining input sanitization patterns for predicting sql injection and cross site scripting vulnerabilities. *Proceedings of the 34th International Conference on Software Engineering.* IEEE Press, 1293–1296.

Shihab, Emad and Ihara, Akinori and Kamei, Yasutaka and Ibrahim, Walid M and Ohira, Masao and Adams, Bram and Hassan, Ahmed E and Matsumoto, Ken-ichi (2013). Studying re-opened bugs in open source software. *Empirical Software Engineering*, *18*(5), 1005–1042.

Shin, Yonghee and Meneely, Andrew and Williams, Laurie and Osborne, Jason A (2011). Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, *37*(6), 772–787.

Sinha, Vibha Singhal and Sinha, Saurabh and Rao, Swathi (2010). BUGINNINGS: Identifying the origins of a bug. *Proc. 3rd India Software Engineering Conf. (ISEC '10).* 3–12.

Śliwerski, Jacek and Zimmermann, Thomas and Zeller, Andreas (2005). When do changes induce fixes? *ACM sigsoft software engineering notes.* ACM, vol. 30, 1–5.

Spanos, Georgios and Angelis, Lefteris and Toloudis, Dimitrios (2017). Assessment of vulnerability severity using text mining. *Proceedings of the 21st Pan-Hellenic Conference on Informatics*. ACM, 49.

Telang, Rahul and Wattal, Sunil (2005). Impact of software vulnerability announcements on the market value of software vendors-an empirical investigation.

Vapnik, Vladimir Naumovich and Vapnik, Vlamimir (1998). *Statistical learning theory*, vol. 1. Wiley New York.

Jaikumar Vijayan (2009). 25 most dangerous software coding errors hackers exploit | cio. `https://www.cio.com/article/2431326/infrastructure/25-most-dangerous-software-coding-errors-hackers-exploit.html`. (Accessed on 10/05/2018).

Walden, James and Doyle, Maureen and Welch, Grant A and Whelan, Michael (2009). Security of open source web applications. *Proceedings of the 2009 3rd international Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 545–553.

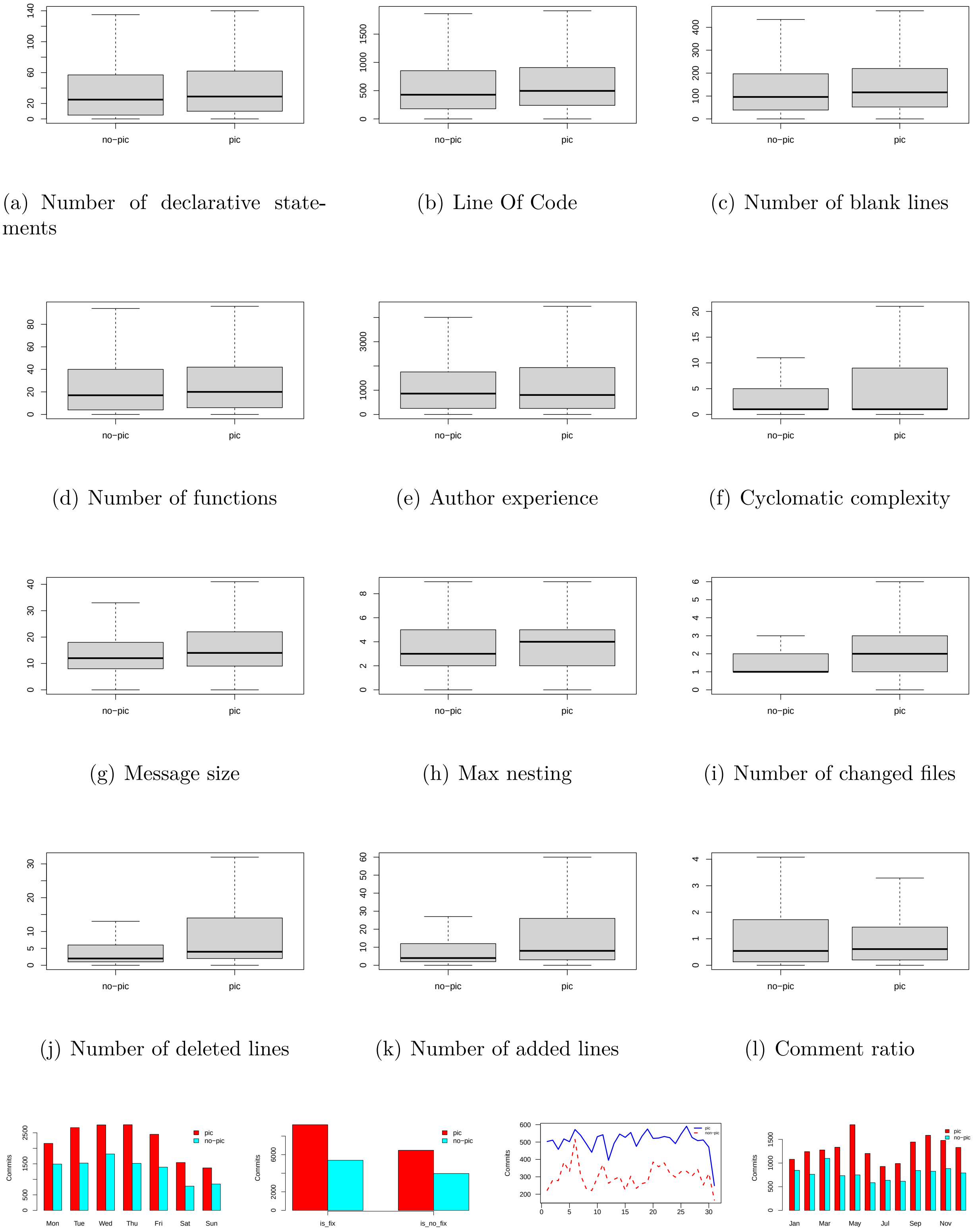
C. Walrad and D. Strom (2002). The importance of branching models in scm. *Computer*, *35*(9), 31–38.

Wassermann, Gary and Su, Zhendong (2007). Sound and precise analysis of web applications for injection vulnerabilities. *ACM Sigplan Notices*. ACM, vol. 42, 32–41.

Woodraska, Daniel and Sanford, Michael and Xu, Dianxiang (2011). Security mutation testing of the filezilla ftp server. *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, 1425–1430.

Robert K.. Yin (2002). *Case Study Research: Design and Methods - Third Edition*. SAGE Publications, troisième édition.

APPENDIX



(a) Number of declarative statements

(b) Line Of Code

(c) Number of blank lines

(d) Number of functions

(e) Author experience

(f) Cyclomatic complexity

(g) Message size

(h) Max nesting

(i) Number of changed files

(j) Number of deleted lines

(k) Number of added lines

(l) Comment ratio

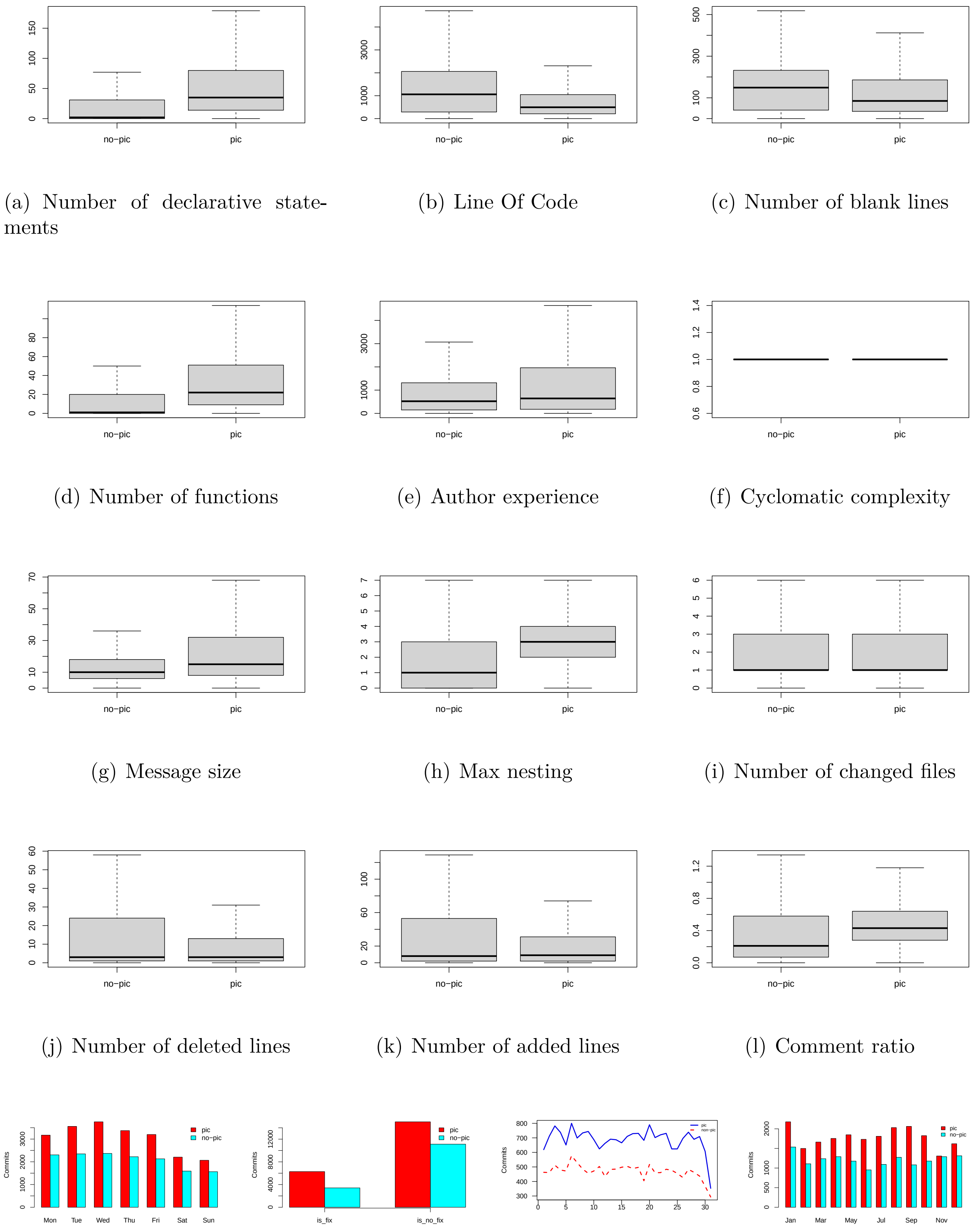
(m) Week days

(n) Is bug fix

(o) Month day

(p) Month

Figure A.1 Distribution of different metrics specifying the characteristics of PICs and non-PICs commits in WordPress.



(a) Number of declarative statements

(b) Line Of Code

(c) Number of blank lines

(d) Number of functions

(e) Author experience

(f) Cyclomatic complexity

(g) Message size

(h) Max nesting

(i) Number of changed files

(j) Number of deleted lines

(k) Number of added lines

(l) Comment ratio

(m) Week days

(n) Is bug fix

(o) Month day

(p) Month

Figure A.2 Distribution of different metrics specifying the characteristics of PICs and non-PICs commits in MediaWiki.