Just-in-time Detection of Protection-Impacting Changes on WordPress and MediaWiki

Amine Barrak
SWAT Lab., Polytechnique Montréal
amine.barrak@polymtl.ca

Marc-André Laverdière
Tata Consultancy Services
Polytechnique Montréal
marc-andre.laverdiere-papineau@polymtl.ca

Foutse Khomh
SWAT Lab., Polytechnique Montréal
foutse.khomh@polymtl.ca

Le An
SWAT Lab., Polytechnique Montréal
le.an@polymtl.ca

Ettore Merlo
Polytechnique Montréal
ettore.merlo@polymtl.ca

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 paper, 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, 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.

KEYWORDS

Protection Impacting changes, Privilege protection changes, Security vulnerabilities, Reliability.

CASCON '18, October, 2018, Markham, Ontario, Canada

ACM Reference Format:
Amine Barrak, Marc-André Laverdière, Foutse Khomh, Le An, and Ettore Merlo. 2018. Just-in-time Detection of Protection-Impacting Changes on WordPress and MediaWiki. In *Proceedings of CASCON '18: 28th Annual International Conference on Computer Science and Software Engineering (CASCON '18).* ACM, New York, NY, USA, 11 pages.

1 INTRODUCTION

Access control, also known as authorization, is a key security mechanism used in software applications. Access control mediates access to resources and functions on the basis of identity and according to a predefined policy. In the case of role-based access control (RBAC), specific roles and privileges are assigned to users of the applications and checks are implemented in the application to ensure that all access conforms to the application's policy. Although simple in appearance, this mechanism is difficult to enforce efficiently in practice. Pinto and Stuttard [23] tested hundreds of applications between 2007 and 2011, and report that 71% of web applications suffer from broken access controls. An example of broken access control is a Fintech application checking that a user is allowed to transfer money from a Paypal account, without validating that this PayPal account belongs to the user.

During the evolution of an application, developers modify the code enforcing access control checks as they add more functions and resources. However, unintentional changes to roles and privileges can also occur, inducing security vulnerabilities in the application. The consequences of unintentional protection changes can be devastating. An attacker can exploit vulnerabilities on privilege protections to view, change or delete sensitive content; execute unauthorized functions; or even take control of the administration of the application. It is therefore utterly important to scrutinize code changes that affect privilege protections in an application. In this paper, 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 statistical models to classify the commits as they are 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 lower. 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 tangled protection-impacting changes.

This would make identification of the root cause and corrections much harder. Thirdly, an analysis based on statistical models is likely to be faster. Previous studies on definite protection differences used an automated Pattern-Traversal Flow Analysis (PTFA) static analysis and operated at a release granularity [17–19]. These analyses required about 10 and 17 minutes on average per release pair, respectively for WordPress and MediaWiki.

In this paper, we use an oracle of protection-impacting changes computed using Pattern Traversal Flow Analysis [13] in 211 release pairs of WordPress and 193 release pairs of MediaWiki. We build machine learning models on commits changes that affect the attribution of privileges in the code, then we perform a qualitative analysis to determine the characteristics of misclassified commits. We answer the following research questions:

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

We analyse protection-impacting changes in two repositories WordPress and MediaWiki. We used RBAC approach to determine PIC lines in 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 in general, PIC commits are submitted by developers with higher experience. Developers are likely to change less files, there are less inserted and deleted lines in PIC commits. They tend to implement more complex source code in PIC commits with high number of functions, more number of declarative statements, high nested level of control functions, more cyclomatic complexity of nested functions and more comment ratio. Developers also tend to make more faults 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 or not a commit is a PIC. Our predictive models can reach a precision of 73.8% and a recall up to 98.3% in WordPress. In MediaWiki, we obtain a precision of 77.2% and a recall of 97.8%. 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 can be 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 commits related to documentation or commits 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.

Paper Organization. The rest of the paper is organized as follows. In Section 2, we present background information about Pattern Traversal Flow Analysis and the detection of protection-impacting changes. In Section 3, we present the design of our case study. We answer our research questions in Section 4 and discuss the threats to validity of our study in Section 5. Section 6 discusses the related literature. We conclude the paper in Section 7.

2 PATTERN TRAVERSAL FLOW ANALYSIS AND PRIVILEGE PROTECTION CHANGE DETECTION

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

2.1 Pattern Traversal Flow Analysis

Pattern Traversal Flow Analysis (PTFA) [11, 13, 21] is an automated whole-program static analysis for 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 the verification of 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. We simplify these models by retaining only reachable states and transitions, which we call the *reachable PTFA model*.

2.2 Definite Protection Differences

When a privilege $priv$ is verified on all paths leading to CFG vertex v, 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 and no unprotected state for v_i exists in the model for $priv$. If an unprotected state for v is reachable in the automaton for $priv$, then $priv$ is not verified in at least one path to v.

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 [17]. 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) [19] depend on code changes and graph reachability in reachable PTFA models (Equation 1).

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 either 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 1) is true whenever the state $q_{i,j,k} \in Ver_a$ has no corresponding state in Ver_b. Likewise, the predicate *addedState* (Figure 1) 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 deleted edges belonging to positively-protected paths to v_a and the added edges 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 1. In this figure, $ReachachableEdges(q_{i,j,k})$ 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 combines changed edges (i.e. *addedEdges* or *deletedEdges*) with reachable edges for either Ver_a or Ver_b. Finally, 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 acceptability. 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. 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. for added code). 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.

2.5 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 [33]. 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 forked branch was created. We chose also a series of releases in the same branch where developers test quality and fix bugs to prepare for a production release [33]. We prepare all chosen versions, then determine protection-impacting lines of code as illustrated in Figure 1. First, 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.

3 CASE STUDY DESIGN

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

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

3.1 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),

[1] https://github.com/amino33/PIC_WordPress

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

$$addedState(q_{i,j,k}) \doteq i \notin image(vertexMap) \ \vee \ q_{bMap(i),j,k} \notin Q_a$$

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

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

$$partialPIC(changes, i, k) \doteq changes \ \cap \left(ReachableEdges(q_{i,0,k}) \cup ReachableEdges(q_{i,1,k})\right)$$

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

Equation 1: Equations for Protection-Impacting Changes – Adapted from [19]

$$PIC_{l,a}(v_a, v_b) \doteq \left(\bigcup_{(q_1, \ q_2) \in \pi_1(PIC(v_a, v_b))} \{srcLoc(q_1), srcLoc(q_2)\} \right) \cap deleted$$

$$PIC_{l,b}(v_a, v_b) \doteq \left(\bigcup_{(q_1, \ q_2) \in \pi_2(PIC(v_a, v_b))} \{srcLoc(q_1), srcLoc(q_2)\} \right) \cap \ added$$

Equation 2: Per-Privilege Projection of Protection-Impacting Changes to Lines of Code

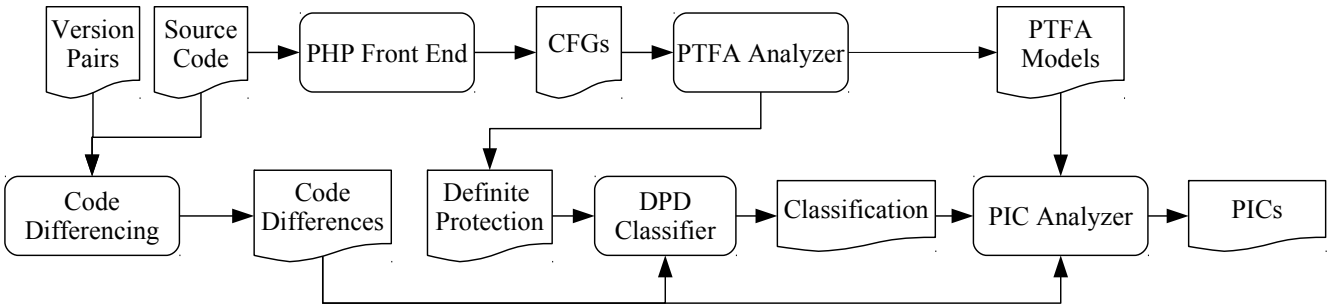

Figure 1: Processing Steps for Detecting Protection-Impacting Lines of Code Using PTFA

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 [24] and release date information. Each edge of that tree is a *release pair*. This approach was used previous RBAC evolution surveys [17–19]. 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

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

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].

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.

3.2 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.

3.3 Computing Metrics

To capture the characteristics of protection-impacting changes, we compute the 16 metrics described in Table 1. We group the metrics in the two following categories:

3.3.1 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 [6]. 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. [32] found that there is usually a correlation between code change size and security issues. Fourth, we computed message size metrics, as Alali et al. [1] 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. [30]. Using this information we compute the Boolean metric *Is bug fix* for each commit.

3.3.2 Code complexity metrics. For each studied commit, we use the Mercurial git log command to extract all of its changed PHP files. Then, we apply the source code analysis tool *Understand* from [28] 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 [2]. 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.

4 STUDY RESULTS

In this section, we report and discuss answers to our research questions. For each research question, we present the motivation, our approach to answer the question, and our 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 software web 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 3.2. 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.

Finding. 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 3.2) we identified 62% (15700 / 25069) of PICs commits in WordPress and 59% (21335 / 35864) of PICs commits

[3] https://github.com/WordPress/WordPress
[4] https://gerrit.wikimedia.org/r/p/mediawiki/core.git

Table 1: Metrics used to compare characteristics

Attribute	Explanation and Rationale
Commit Log Metrics	
Week day	Day of week (from Mon to Sun). Code committed on certain week days may be less carefully written (*e.g.*, Friday) [4, 30].
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.
Code Complexity Metrics	
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

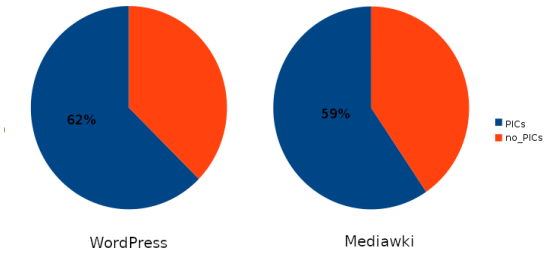

Figure 2: Proportion of Protection-Impacting changes commits in WordPress and MediaWiki

in MediaWiki. Figure 2 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 of the studied releases.*

Nearly half of commits are likely to contain PICs, which are at risk of introducing vulnerabilities [34]. Therefore, software developers should strive to catch PIC commits as soon as possible, e.g., when they are submitted into the version control system.

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 1 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 [14] 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 [9]. 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 [7], which measures the magnitude of the difference while controlling for the confounding factor of sample size [8]. We assess the magnitude using the threshold provided in [27], *i.e.,*, $|d| < 0.147$ "negligible"(N), $|d| < 0.33$ "small"(S), $|d| < 0.474$ "medium"(M), otherwise "large"(L).

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

Finding. Table 2 and 3 summarise differences between the characteristics of commits corresponding to the projects WordPress and MediaWiki, that introduced protection-impacting changes and others *i.e.,* commits that did not alter protection privileges. 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

Table 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 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%	–	–

Table 3: 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%	–	–

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; which means that privileges are often inadvertently modified during bug fixing, which may result in vulnerabilities. Woodraska et al.[34] found that bugs can turn into severe security vulnerabilities which is consistent with our results.

In light of results from Table 2 we reject all the null hypotheses H_i^0. In other words, for all metrics listed in Table 2, there exist statistically significant differences between PICs and non-PICs commits in varying proportions.

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 [12, 16], 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 from delivering defects to end users. In this research question, we examine the possibility of using just-in-time prediction models to identify unintentional privilege protection changes in real-time.

Approach. We use the metrics from Table 1 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: logistic regression, Naive Bayes, decision tree, and Random Forest. Logistic regression extends 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 [31] with strong independence assumptions between the features. This algorithm often obtains, in practice, a good classification result [25]. Compared to logistic regression, 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. To further mitigate the biases and variance from the decision tree model, Leo Breiman and Adele Cutler introduced Random Forest [5], 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.,* [3], Random Forest achieved the best prediction accuracy. In this study, we build 1,000 trees, each of which is 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 [26], we remove the metrics whose VIF values are greater than or equal to 5. In Tables 1, the removed metrics are marked with *.

Table 4: Accuracy, precision, recall and F-measure (in %) 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%

We apply ten-fold cross validation [10] to measure the fitness of the models. We will report respectively the general accuracy, as well as the precision, recall, and F-measure on the commits or change files that contain PIC(s) for each model. 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 [15]. 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.

Finding. Table 4 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 achieves the best F-measure when predicting PICs commits.

Our predictive models can achieve a precision of 73.8%, and a recall of 97.3% in Wordpress. In MediaWiki, models achieve a precision of 77.2%, and a recall of 97.8%. 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.

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.

Finding. We summarize our observations in Table 5. 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 withing 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.

These observations lead us to methodological improvements to consider in future research. For example we could use island parsing to ignore/minimise non-PHP code changes, which would likely reduce these misclassifications.

Table 5: 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

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, for example using island parsing. 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.*

5 THREATS TO VALIDITY

We now discuss the threats to the validity of our study following the guidelines for case study research [35].

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) [18].

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.

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.

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.

6 RELATED WORK

In this section, we discuss related studies on protection-impacting analysis and just-in-time prediction.

6.1 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. [20] conducted a longitudinal study of privilege protection over 31 phpBB releases. This application only used a binary distinction between administrator and unprivileged users. Laverdière and Merlo [17, 18] defined definite protection differences (previously named privilege protection changes) for richer protection schemes than Letarte et al. They conducted longitudinal studies over 147 release pairs of WordPress on the presence of privilege protection changes, and their classification. They also showed how to compute counter-examples for privilege protection losses. These counter-examples are paths that do not integrate code change information. Laverdière and Merlo [19] defined protection-impacting changes using PTFA. They conducted a survey of 210 release pairs of WordPress. Their static analysis operated at the granularity of release pairs and takes many minutes to complete.

In our paper, we built predictive models for security-impacting changes that operate to commit granularities.

Protection-impacting change analysis is conceptually similar to approaches that identify the cause of bugs during evolution, such as Bugginings [29]. This tool identifies bug-introducing code changes. It computes differences dependence graphs and may investigate multiple versions. However, a major difference is that these tools rely on an external oracle (*e.g.,* bug reports), whereas our approach is predictive.

6.2 Just-in-time Prediction

Traditional defect prediction techniques often use metrics from bug reports to identify fault-prone modules or severity of bugs. Though such defect prediction techniques can help software organizations prevent defects to some extent, they don't help developers handle defects as soon as they are introduced in the system. In other words, before a defect is definitely resolved, users have to suffer from frustrations, such as bad user experience, data loss, and–or privacy threats.

Just-in-Time defect prediction techniques are designed to predict defects at commit level; helping developers to locate and address defects right after a commit is submitted for integration in a version control system. In a previous work, Kamei et al. [16] extracted a variety of source code metrics at commit level to predict defect-prone commits from six open-source systems and five commercial systems. Using just-in-Time defect prediction techniques, Fukushima et al. [12] performed cross-project defect predictions. They observed that the cross-project approach can be used for projects that possess little historical data. Misirli et al. [22] extracted a series of code and process factors at commit level and built statistical models to predict high impact fix-inducing changes.

As far as we know, this is the first study of just-in-time prediction techniques to identify protection-impacting changes.

7 CONCLUSIONS AND FUTURE WORK

In this paper, 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 observe that around 60% of commits submitted into 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 build statistical models. The evaluation of these models showed 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 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.

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.

ACKNOWLEDGEMENT

This work is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

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

[2] Le An and Foutse Khomh. 2015. An Empirical Study of Highly Impactful Bugs in Mozilla Projects. In *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*. IEEE, 262–271.

[3] Le An, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2017. An empirical study of crash-inducing commits in Mozilla Firefox. *Software Quality Journal* (2017).

[4] Prasanth Anbalagan and Mladen Vouk. 2009. Days of the week effect in predicting the time taken to fix defects. In *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.

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

[6] Marco Castelluccio, Le An, and Foutse Khomh. 2017. Is It Safe to Uplift This Patch? An Empirical Study on Mozilla Firefox. *arXiv preprint arXiv:1709.08852* (2017).

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

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

[9] A. Dmitrienko, G. Molenberghs, C. Chuang-Stein, and W.W. Offen. 2005. *Analysis of Clinical Trials Using SAS: A Practical Guide*. SAS Institute. http://www.google.ca/books?id=G5ElnZDDm8gC

[10] Bradley Efron. 1983. Estimating the error rate of a prediction rule: improvement on cross-validation. *J. Amer. Statist. Assoc.* 78, 382 (1983), 316–331.

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

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

[13] François Gauthier and Ettore Merlo. 2012. Alias-Aware Propagation of Simple Pattern-Based Properties in PHP Applications. In *Proc. 12th IEEE Int'l Working Conf. Source Code Analysis and Manipulation (SCAM '12)*. 44–53. https://doi.org/10.1109/SCAM.2012.19

[14] Myles Hollander, Douglas A Wolfe, and Eric Chicken. 2013. *Nonparametric statistical methods*. John Wiley & Sons.

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

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

[17] Marc-André Laverdière and Ettore Merlo. 2017. Classification and distribution of RBAC privilege protection changes in wordpress evolution,âĂİ. In *Proc. 15th Int'l Conf. Privacy, Security and Trust (PST'17)*.

[18] Marc-André LaverdiÃĺre and Ettore Merlo. 2017. Computing Counter-Examples for Privilege Protection Losses Using Security Models. In *Proc. 24th IEEE Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER '17)*. 240–249.

[19] Marc-André LaverdiÃĺre and Ettore Merlo. 2018. Detection of protection-impacting changes during software evolution. In *2018 IEEE 25th International*

Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 434–444.

[20] Dominic Letarte, François Gauthier, and Ettore Merlo. 2011. Security Model Evolution of PHP Web Applications. In *Proc. Fourth IEEE Int'l Conf. Software Testing, Verification and Validation (ICST '11)*. 289–298. https://doi.org/10.1109/icst.2011.36

[21] D. Letarte and E. Merlo. 2009. Extraction of Inter-procedural Simple Role Privilege Models from PHP Code. In *Proc. 16th IEEE Working Conf. Reverse Engineering (WCRE '09)*. 187–191. https://doi.org/10.1109/WCRE.2009.32

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

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

[24] Tom Preston-Werner. 2013. Semantic Versioning 2.0.0. (06 2013). http://www.semver.org

[25] Irina Rish. 2001. An empirical study of the naive Bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, Vol. 3. IBM New York, 41–46.

[26] Peter A Rogerson. 2010. *Statistical methods for geography: a student's guide*. Sage Publications.

[27] Jeanine Romano and Jeffrey D. Kromrey. 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? (01 2006).

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

[29] Vibha Singhal Sinha, Saurabh Sinha, and Swathi Rao. 2010. BUGINNINGS: Identifying the Origins of a Bug. In *Proc. 3rd India Software Engineering Conf. (ISEC '10)*. 3–12. https://doi.org/10.1145/1730874.1730879

[30] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes?. In *ACM sigsoft software engineering notes*, Vol. 30. ACM, 1–5.

[31] Vladimir Naumovich Vapnik and Vlamimir Vapnik. 1998. *Statistical learning theory*. Vol. 1. Wiley New York.

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

[33] C. Walrad and D. Strom. 2002. The importance of branching models in SCM. *Computer* 35, 9 (Sep 2002), 31–38. https://doi.org/10.1109/MC.2002.1033025

[34] Daniel Woodraska, Michael Sanford, and Dianxiang Xu. 2011. Security mutation testing of the FileZilla FTP server. In *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, 1425–1430.

[35] Robert K.. Yin. 2002. *Case Study Research: Design and Methods - Third Edition* (3rd ed.). SAGE Publications.