Detection of Protection-Impacting Changes during Software Evolution

Marc-André Laverdière
TCS Innovation Labs, Tata Consultancy Services
&
École Polytechnique de Montréal
Montréal, Canada
marcandre.laverdiere@tcs.com

Ettore Merlo
École Polytechnique de Montréal
Montréal, Canada
ettore.merlo@polymtl.ca

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

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

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

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

***Index Terms*—Security Impact of Changes, Role Based Access Control, Static Analysis, Software Evolution, Software Maintenance**

I. Introduction

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

These security expectations are summarized in industry standards, such as the OWASP Application Security Verification Standard, whose purpose is "to define what a secure application is." (OWASP [1, p.5]). This standard covers *access control* and other security topics. Access control ensures that only the right users execute some specific code.

Web applications may suffer from vulnerabilities, which are defined as "a flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy." (RFC 4949 [2]). An access control vulnerability allows the violation of a resource's security policy (e.g. an unauthorized data access or performing unauthorized operations) [3], [4].

Role-based access control (RBAC) is commonly used for access control. RBAC belongs to a family of access control called authentication-based access control family (NBAC) [5]. In RBAC, developers assign *privileges* to one or more *roles* as necessary and insert privilege checks across the code base according to the application's security needs.

Developers need quality assurance processes to prevent such issues from occurring and ensuring that their code changes only have a planned and desirable impact on security. They need quality assurance processes that encompass verification and validation. These often leverage testing [6], [7], [8], reviews [8], [9] and formal methods [10]. All these verification and validation approaches require developers' effort and are typically time consuming. They also have limitations and thus they should be combined for better results [11], [12].

Like other kinds of software, web applications undergo maintenance and evolution. Software maintenance can be corrective, adaptive, perfective, and preventive [13]. Corrective and adaptive maintenance require an effort of about 30-50% [14], [15]. Maintenance activities oriented towards security [16] are included in the remaining 50-70% of effort. They include vulnerability correction, modifying security checks to accommodate RBAC policy changes and malicious changes from insiders [17]. RBAC policy changes involve adding, removing and renaming roles and privileges. These activities intentionally change the security protection of different parts of a system.

Many maintenance activities are not specifically meant to modify the security of the application. These include bug-fixing, refactoring, design changes, and changing the functionality of the application. These changes may unintentionally introduce

978-1-5386-4969-5/18/$31.00 © 2018 IEEE

security flaws [18]. Thus, code changes may impact the security protection in the application in different and seemingly unrelated parts of the application. This occurs because of added, removed or modified execution paths.

Because RBAC security issues are hard to reason about, developers should ideally verify and validate that each change is free from security regressions throughout the project's evolution. This would enable rapid correction, as well as avoiding complex regressions due to the combined effect of many changes. However, a constant re-verification of the whole application would be very effort-intensive. Even if individual changes alter a small part of the application, "software modified during maintenance should be subjected to the same review as newly developed software" (Landwehr et al. [18]). As developers increasingly adopt frequent and quick release cycles, this overhead represents a major hurdle.

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

We propose an automated static analysis over privilege protection that automatically classifies changed lines of code as protection-impacting or non-protection-impacting. This is intended to safely reduce the amount of changes that developers must assess before a release. Our analysis finds *protection-impacting changes* (PIC), the subset of the code changes which contains the root causes of security changes related to privilege protection. This analysis relies on the automated detection of *definite protection differences (DPD)*, which are changes in statements' definite privilege protection (i.e. privilege protection changes [20], [21]).

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

This article is organized as follows. In Section II, we cover the background necessary to understand our contributions. Then, in Section III, we define protection-impacting changes. Afterwards, we show an example in Section IV and describe our experiments in Section V. Then, we share and discuss our results in Section VI and VII, respectively. We describe related work in Section VIII and consider threats to validity in Section IX. We bring concluding remarks in Section X.

II. Background

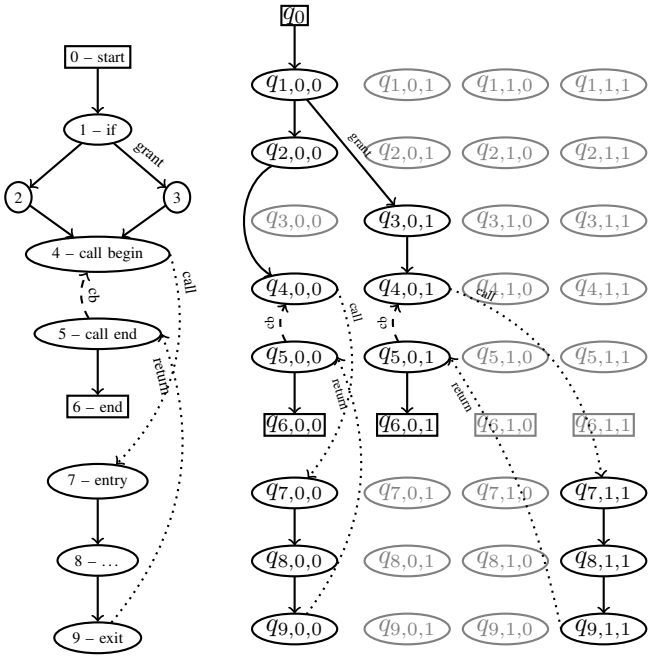
A. Pattern Traversal Flow Analysis

Our research objectives revolve around privilege protection, especially definite privilege protection. Pattern-Traversal Flow Analysis (PTFA) [22], [23], [24], is a domain-specific model-checking approach to determine privilege protection. PTFA verifies the property satisfaction that a specific code pattern was definitely executed on all paths reaching a statement s. Patterns are privilege verification checks for some privilege $priv \in Privileges$.

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

PTFA models are generated automatically and statically from the application's source code. This generation is based on graph rewriting rules applied on the application's CFG [24].

We illustrate the CFG into a PTFA model in Figure 1. Please note that, by convention, the i index of PTFA states $q_{i,j,k}$ refers to the CFG vertex number. In this example, we show the effect of grant and interprocedural edges in the CFG. Please note that we represented unreachable states in grey in this figure to facilitate comprehension.



(a) Example CFG (b) Corresponding PTFA Model

Figure 1. Example CFG and Corresponding PTFA Model – Unreachable States Greyed Out

Definite privilege protection $DefProt(w)$ refers to privileges verified on all interprocedural paths prior to the execution of a statement (Equation 1). Without PTFA, it is difficult to determine the definite privilege protection over large systems with a complex interprocedural call graph. The predicate $Prot(v, p, priv)$ represents the protectedness of v for privilege $priv$ on a path p from the initial vertex v_0.

$$DefProt(w, priv) = \forall p\,(v_0, \ldots, w) \mid Prot(w, p, priv) \equiv \nexists p'\,(v_0, \ldots, w) \mid \neg Prot(w, p', priv) \quad (1)$$

In contrast, PTFA indicates definite privilege protection thanks to state reachability. All states present in the model are reachable from q_0. For CFG vertex w, a protected state $q_{w,j,1}$ exists only when there exists a protected path between v_0 and w. And an unprotected $q_{w,j,0}$ exists only when there exists an unprotected path between v_0 and w. Thus, definite privilege protection for w is determined by the existence of protected states corresponding to w and the absence of unprotected states corresponding to w (Equation 2).

$$DefProt(w, priv) = \exists q_{w,j_1,k_1} \in Q_{\mathcal{A}} \mid k_1 = 1 \wedge \nexists q_{w,j_2,k_2} \in Q_{\mathcal{A}} \mid k_2 = 0 \tag{2}$$

B. Definite Protection Differences

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

We consider that statement s is *loss-affected* for privilege $priv$, if it was $priv$ protected in Ver_a and is no longer $priv$ protected in Ver_b (c.f. *lossAffected* in Equation 3). A similar definition applies for *gain-affected* statements, whenever statement s was granted a $priv$ protection in Ver_b that was not there in Ver_a (c.f. *gainAffected* in Equation 4). Please note that statement s may be loss-affected with regards to privilege p_1, but gain-affected with regards to privilege p_2.

$$lossAffected(priv) = \left\{ v_b \;\middle|\; \begin{array}{c} (v_a, v_b) \in vertexMap \wedge \\ DefProt(v_a, priv) \wedge \\ \neg DefProt(v_b, priv) \end{array} \right\} \tag{3}$$

$$gainAffected(priv) = \left\{ v_b \;\middle|\; \begin{array}{c} (v_a, v_b) \in vertexMap \wedge \\ \neg DefProt(v_a, priv) \wedge \\ DefProt(v_b, priv) \end{array} \right\} \tag{4}$$

III. Method

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

We define two criteria for protection-impacting changes in Sections III-A and III-B. We then define protection-impacting changes in Section III-C based on these two criteria.

A. Added and Deleted Edges

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

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

From this heuristic mapping at the CFG level, we compute the changes between PTFA models as follows. Q_a and T_a respectively are the set of states and the set of transitions in the PTFA model for version Ver_a. Q_b and T_b are similarly defined for the PTFA model for Ver_b. The predicate $deletedState(q_{i,j,k})$ (Equation 5) is true whenever $q_{i,j,k} \in Ver_a$ has no corresponding state in Ver_b. Its converse, $addedState$ (Equation 6), is true whenever $q_{i,j,k} \in Ver_b$ has no corresponding state in Ver_a. The symbols dom and $image$ correspond to the function's domain and image, respectively.

$deletedEdges$ is the set of deleted edges in the PTFA model for Ver_a (Equation 7). It contains all edges which either connect one or more deleted states or for which no corresponding edge is present in the PTFA model for Ver_b. Conversely, $addedEdges$ is the set of added edges in to PTFA model for Ver_b (Equation 8). The function $bMap$ is the reverse-function of $vertexMap$.

$$deletedState(q_{i,j,k}) \doteq \begin{array}{l} i \notin dom(vertexMap) \vee \\ q_{vertexMap(i),j,k} \notin Q_b \end{array} \tag{5}$$

$$addedState(q_{i,j,k}) \doteq \begin{array}{l} i \notin image(vertexMap) \vee \\ q_{vertexMap^{-1}(i),j,k} \notin Q_a \end{array} \tag{6}$$

$$deletedEdges \doteq \left\{ \begin{pmatrix} q_{i_1,j_1,k_1} \\ q_{i_2,j_2,k_2} \end{pmatrix} \in T_a \;\middle|\; \begin{array}{l} deletedState(q_{i_1,j_1,k_1}) \vee \\ deletedState(q_{i_2,j_2,k_2}) \vee \\ \begin{pmatrix} q_{vertexMap(i_1),j_1,k_1} \\ q_{vertexMap(i_2),j_2,k_2} \end{pmatrix} \notin T_b \end{array} \right\} \tag{7}$$

$$addedEdges \doteq \left\{ \begin{pmatrix} q_{i_1,j_1,k_1} \\ q_{i_2,j_2,k_2} \end{pmatrix} \in T_b \;\middle|\; \begin{array}{l} addedState(q_{i_1,j_1,k_1}) \vee \\ addedState(q_{i_2,j_2,k_2}) \vee \\ \begin{pmatrix} q_{bMap(i_1),j_1,k_1} \\ q_{bMap(i_2),j_2,k_2} \end{pmatrix} \notin T_a \end{array} \right\} \tag{8}$$

B. Appropriately Protected Paths

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

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

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

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

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

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

First, v_a is definitely protected for $priv$, which means that all paths to v_a are $priv$-protected. Second, v_b is not definitely protected for $priv$, which means that either it is unreachable (dead code) or there is at least one unprotected path for $priv$ leading to it (Equation 9).

$$\begin{gathered}\forall p\left(v_0,\ldots,v_a\right) \mid Prot(v_a,p,priv) \wedge \\ \left(\exists p'\left(v_0,\ldots,v_b\right), \neg Prot(v_b,p',priv) \vee \nexists p''\left(v_0,\ldots,v_b\right)\right)\end{gathered} \tag{9}$$

As we saw in Section II-A, these universally and existentially quantified predicates correspond to safety and liveness property verification in PTFA models. Equation 9 thus translates to Equation 10. Please note that $Reach$ is a predicate refers to forward-reachability in the PTFA model. We define the $posProtStateExists$ and $negProtStateExists$ predicates in Equation 11. For Equation 10 to hold true, we only need to consider the protected paths to $q_{a,j,1}$ and (if any) the unprotected paths to $q_{b,j',0}$. Thus, protection-impacting changes must belong to these paths.

$$\begin{gathered}posProtStateExists(a) \wedge \neg negProtStateExists(a) \wedge \\ \left(negProtStateExists(b) \vee \nexists j,k \mid \neg Reach(q_{b,j,k})\right)\end{gathered} \tag{10}$$

$$\begin{aligned}posProtStateExists(i) &\doteq Reach(q_{i,0,1}) \vee Reach(q_{i,1,1}) \\ negProtStateExists(i) &\doteq Reach(q_{i,0,0}) \vee Reach(q_{i,1,0})\end{aligned} \tag{11}$$

The situation is similar when dealing with gain-affected CFG vertices. First, v_a is not definitely protected for $priv$, which means that either it is unreachable (dead code) or there is at least one unprotected path for $priv$ leading to it. Second, v_b is definitely protected for $priv$, which means that all paths to v_b are $priv$-protected (Equation 12).

$$\begin{gathered}\forall p\left(v_0,\ldots,v_b\right) \mid Prot(v_b,p,priv) \wedge \\ \left(\exists p'\left(v_0,\ldots,v_a\right) \mid \neg Prot(v_a,priv) \vee \nexists p''\left(v_0,\ldots,v_a\right)\right)\end{gathered} \tag{12}$$

Equation 12 thus translates to Equation 13. For Equation 13 to hold true, we only need to consider the protected paths to $q_{b,j,1}$ and (if any) the unprotected paths to $q_{a,j',0}$. Thus, protection-impacting changes must belong to these paths.

$$\begin{gathered}posProtStateExists(b) \wedge \neg negProtStateExists(b) \wedge \\ \left(negProtStateExists(a) \vee \nexists j,k \mid \neg Reach(q_{a,j,k})\right)\end{gathered} \tag{13}$$

C. Definition of Protection-Impacting Changes

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

For loss-affected code, the protection-impacting changes PIC_{loss} are the deleted edges belonging to positively-protected paths to v_a and the added edges belonging to negatively-protected paths to v_b (Equation 16). This definition depends on $posProtEdges$ (Equation 14) and $negProtEdges$ (Equation 15), which return the set of edges belonging to paths between q_0 and, respectively, protected and unprotected states. Please note that $ReachEdges(q_{i,j,k})$ is a function returning all edges between the initial state q_0 and $q_{i,j,k}$.

$$posProtEdges(i) \doteq \bigcup_{j\in\{0,1\}} ReachEdges(q_{i,j,1}) \tag{14}$$

$$negProtEdges(i) \doteq \bigcup_{j\in\{0,1\}} ReachEdges(q_{i,j,0}) \tag{15}$$

$$\begin{aligned}&posProtDel \doteq deletedEdges \cap posProtEdges(v_a) \\ &negProtAdd \doteq addedEdges \cap negProtEdges(v_b) \\ &PIC_{loss} \doteq (posProtDel\ ,\ negProtAdd)\end{aligned} \tag{16}$$

Similarly, for gain-affected code, protection-impacting changes PIC_{gain} are the deleted edges belonging to negatively-protected paths to v_a ($negProtDel$) and the added edges belonging to positively-protected paths to v_b ($posProtAdd$) (Equation 17).

$$\begin{aligned}&negProtDel \doteq deletedEdges \cap posProtEdges(v_a) \\ &posProtAdd \doteq addedEdges \cap negProtEdges(v_b) \\ &PIC_{gain} \doteq (negProtDel\ ,\ posProtAdd)\end{aligned} \tag{17}$$

IV. Example

We now provide a ficticious example of definite protection differences caused by a code change in Figure 2. Please note that we combined the CFGs of Ver_a and Ver_b, as well as their PTFA models, due to space constraints.

The code change (Figure 2a) adds a call to function `foo()` from an unsecured context. Previously, `foo()` was exclusively called from protected contexts. Following that change, the code in `foo()` (vertices 5 to 7) becomes loss-affected.

The CFGs (Figure 2b) are combined and we show differences using colour and dashed edges. In order to avoid confusion, and contrary to the representation in Figure 1, we represented all other edges solid. The blue dashed edge is a deleted edge from Ver_a. We use green for added vertices and edges Ver_b. Counterintuively, adding code caused a deleted edge, since the control flow no longer flows directly from vertex 0 to 1.

The PTFA models (Figure 2c) are similarly combined and coloured. We greyed out unreachable states. We also marked all protection-impacting edges with "**PIC**".

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

V. Experiment Design

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

Our research question is the following:

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

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

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

A. Approach

To determine if protection-impacting changes are smaller than code changes in real software projects, we conduct a survey [25] of one open source project, WordPress. In this survey, we determine code changes and protection-impacting changes between major releases, as detailed in Subsection V-B.

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

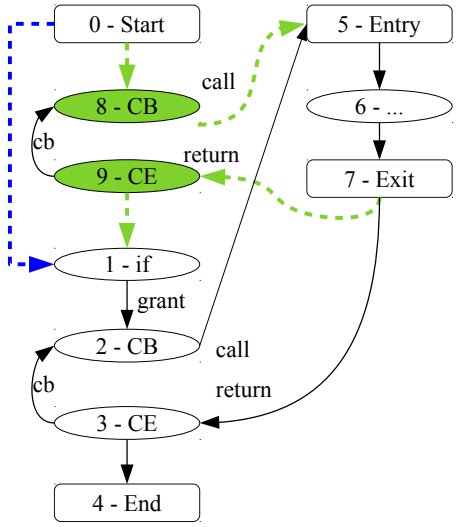
Our process consists of a PHP front-end, a PTFA engine, a DPD classifier and a protection-impacting change analyzer. The PHP front-end feeds CFGs to the PTFA engine, which then generates PTFA models. The DPD classifier determines definite protection differences using code differences and definite privilege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

B. Data Collection and Processing

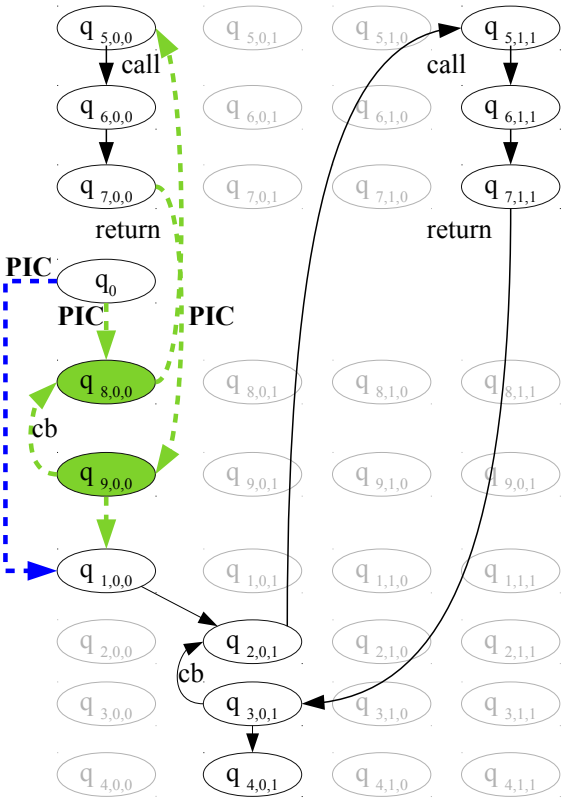
Our approach relies on comparing pairs of versions, and we must choose *which* versions. As an initial experiment, we choose to analyze official final releases (i.e. non-alpha, non-beta, non-release candidate versions). A finer-grained (e.g. commit-level) analysis is possible, which we discuss in Section VII. Since these are the versions that end-users normally deploy, we analyze the official release archives [26]. These archives are bundled with files that are not present in the repository. Additionally, we expect that these releases will be better tested and reviewed than intermediary versions.

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

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



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



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

Figure 2. Example Code, CFG and PTFA Model

Because WordPress developers maintain and apply security fixes to multiple releases in parallel, we cannot consider releases sequentially. We compared release pairs according to their semantic versioning [27] and release date information. We organize release pairs in a tree. The edges of that tree are *release pairs*, which are the releases we compare. This is the same approach we used in our earlier studies [20], [21].

C. Measures

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

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

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

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

Respectively, $PIC_{loss,lines}$ and $PIC_{gain,lines}$ are the projection of PIC_{loss} and PIC_{gain} to a set of lines. We define PIC (Equation 18) as the union of both these projections. Additionally, we combined the PIC's elements into the metric $allPIC$ (Equation 19). To simplify the equations below, we use the projection function π_i to extract the ith element of the pair $x = (Ver_a, Ver_b)$.

$$PIC = \begin{pmatrix} \pi_1(PIC_{loss,lines}) \cup \pi_1(PIC_{gain,lines}) \\ \pi_2(PIC_{loss,lines}) \cup \pi_2(PIC_{gain,lines}) \end{pmatrix}, \quad (18)$$

$$allPIC = |\pi_1(PIC)| \ + \ |\pi_2(PIC)| \quad (19)$$

The relative size of protection-impacting changes per release pair is the ratio between the total protection-impacting lines and the total modified lines of code (Equation 20). In the equations below, $delLines$ and $addLines$, respectively are the number of lines deleted from Ver_a and added to Ver_b.

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

$$\begin{aligned} relPIC_{loss} &= \\ &\left(\frac{|\pi_1(PIC_{loss,lines})|}{delLines} \ , \frac{|\pi_2(PIC_{loss,lines})|}{addLines} \right) \\ relPIC_{gain} &= \\ &\left(\frac{|\pi_1(PIC_{gain,lines})|}{delLines} \ , \frac{|\pi_2(PIC_{gain,lines})|}{addLines} \right) \\ relPIC &= \left(\frac{|\pi_1(PIC)|}{delLines} \ , \frac{|\pi_2(PIC)|}{addLines} \right) \\ relAllPIC &= \frac{|\pi_1(PIC)| \ + \ |\pi_2(PIC)|}{delLines \ + \ addLines} \end{aligned} \quad (20)$$

VI. Experimental Results

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

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

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

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

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

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

Table I
PROTECTION-IMPACTING CHANGES PER RELEASE PAIR

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

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

We show the histogram of protection-impacting changes per release pair for security-affected release pairs in Figure 3. In this histogram, we report protection-impacting changes for gains (PIC_{gain}), losses (PIC_{loss}), their combined measure (PIC) and the combined metric $allPIC$.Please note that protection-impacting changes may be both gain-impacting and loss-impacting. We used a non-linear X axis, with bins of different widths, in order to better represent the distribution.

In Figure 3, we observe that, among security-affected release pairs, 50/87 (57%) release pairs have less than 100 lines of code classified as protection-impacting changes. We also observe that 61/87 (70%) security-affected release pairs have less than 500 protection-impacting lines of code. Only 26/87 (30%) release pairs have 500 or more lines of protection-impacting changes. The distribution is long-tailed in reality. We observe outliers on the right-hand side of the distribution. We found 8/87 (9%) release pairs for which $allPIC > 10,000$. All these release pairs have a minor or major release number (i.e. a .0 release) for Ver_b. The volume of code changes for these releases is very high, with a mean of over 31 KLOC and 16 KLOC added and deleted lines, respectively. Thus, the relative proportion of protection-impacting change in these release pairs is not very high, with a mean $relAllPIC = 38.75\%$. This situation explains the gap between the medians and means in Table I.

In Figure 4, we present an histogram of the relative distributions $relPIC_{loss}$, $relPIC_{gain}$, $relPIC$ and $relAllPIC$. We first observe the large spike at zero in the distribution of $relPIC_{loss}$. In part, this is because 46 release pairs have no loss-affected code in the first place The relative combined measure ($relAllPIC$) is somewhat skewed towards the left. Among security-affected release pairs, 16/87 (18%) release pairs, have less than 10% of changed lines of code classified as protection-impacting changes. Additionally, 42/87 (48%) security-affected release pairs have less than 25% of changed lines of code considered as protection-impacting. We also see that the distribution has 8/87 (9%) security-affected release pairs above 50%. Thus, protection-impacting changes are generally very small compared to the total number of code changes between versions. Two distributions appear long tailed, $relPIC_{gain}$ and $relPIC$ for Ver_a. These occur for release pairs with a small number of changes (<100 LOC) and appear to be very targeted.

As we mentioned earlier, the difference in shape between Figures 3 and 4 is explained by the large volume of changes. For instance, the releases which have over 5000 protection-impacting changes ($allPIC$ > 5000) show a large amount of code changes. In these release pairs, the largest measure of $relAllPIC$ is 54%.

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

VII. DISCUSSION

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

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

Static analysis tools should reduce the amount of information to review [28]. For security-affected releases, although the mean is large (2002.56 LOC), the median of protection-impacting changes appear small (34.00 LOC).

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

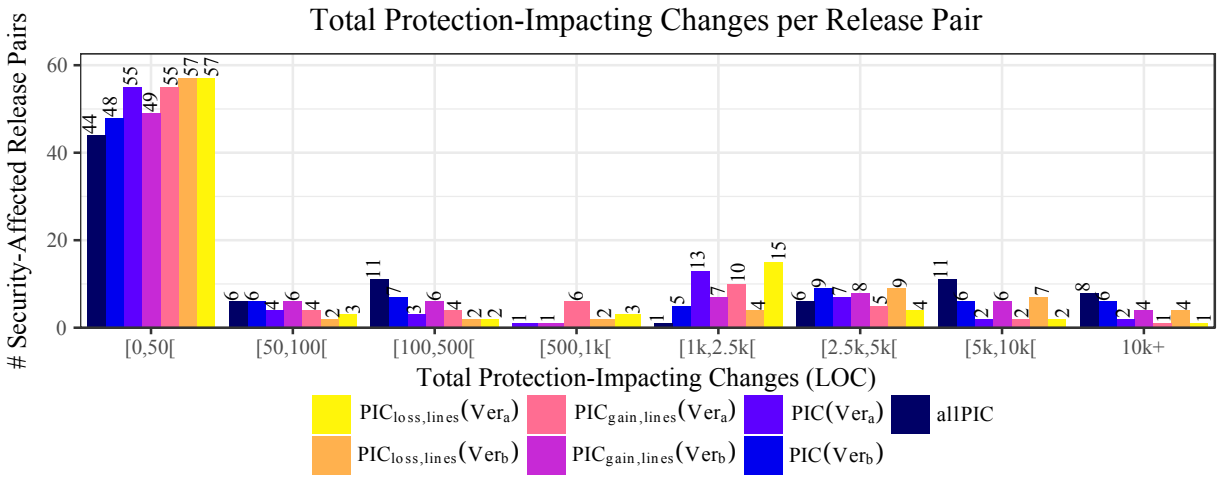


Figure 3. Distribution of Protection-Impacting Lines of Code per Release Pair for security-affected Release Pairs

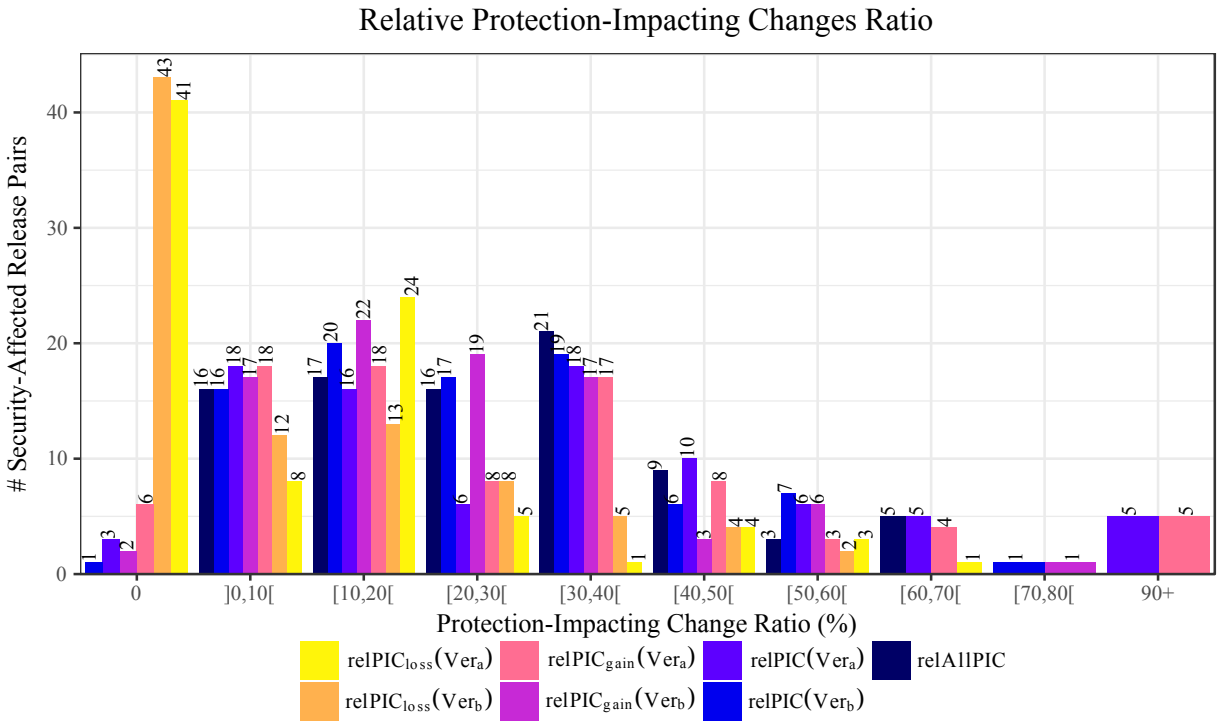


Figure 4. Distribution of Protection-Impacting Lines of Code per Release Pair for security-affected Release Pairs

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

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

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

Furthermore, we envision adding our approach to integrated development environments (IDE), which would allow developers to determine definite protection differences and protection-impacting changes interactively. On-demand analysis may also accelerate the assessment of vulnerability reports and

the creation of security patches. For instance, a developer receiving a vulnerability report may quickly obtain protection-impacting changes for the vulnerable code segment. Then, she may determine which code changes are responsible for the vulnerability and which correction is needed.

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

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

VIII. Related Work

Other researchers used PTFA for evolution studies of RBAC systems. Letarte et al. [29] surveyed privilege protection over 31 phpBB releases, which only handled a binary distinction between administrator and unprivileged users. In our study, we detect protection-impacting changes, handle a richer protection scheme and performed a larger survey.

We previously performed surveys of definite protection difference over WordPress. We defined definite protection differences in a first study [20], and produced of counter-examples for loss-affected code. In this article, we defined and detected protection-impacting changes. We also proposed a classification of definite protection differences [21]. In contrast, this paper focuses on the identification of a superset of the root causes of definite protection differences and is not aimed a the classification and distribution of these differences.

Other researchers used formal methods and static analysis tools to analyze RBAC in applications. MARGRAVE [30] is a tool leveraging formal methods to support access control evolution. It translates XACML to decision diagrams and differences the policies. However, MARGRAVE does not take the implementation of the RBAC policy in consideration.

Other approaches use formal static analysis methods for RBAC systems. One system, ROLECAST [31], automatically detects privilege protection checks in PHP and JSP applications. ROLECAST uses inferences to identify critical variables controlling the reachability of security sensitive statements, infers roles and determines if checks are performed consistently. When reporting a potential vulnerability, ROLECAST mentions many details, including the security sensitive statement, its calling context, which security variables are erroneously verified. Our contribution is very different as we relyy on graph reachability and code differences to connect definite protection differencesto their potential causes.

An extension tool, FIX ME UP [32], statically finds access-control errors of omission and proposes candidate repairs. The approach generates an access control template, detects deviations from this template and changes them to conform. Their algorithm relies on slicing, a more complex algorithm than ours, and does not detect definite protection differences.

RBAC policy evolution is also useful to perform test selection [33]. This approach relies on semantic differencing of XACML policies and requires running the full test suite at least once. Another study [34] used testing to detect hidden and implicit security mechanisms. In comparison, our approach is static and identifies potential root causes for protection changes.

Model checking is also useful to identify candidate causes of errors. Ball et al. [35] proposed an method which identifies transitions in an error trace that are in no correct trace and injects halt statements to produce additional error traces. Groce et al. [36] expressed counter-example generation as a minimization problem whose distance metric considers causal dependence. While we have in common the use of model checking, these approaches do not address privilege protection.

BUGGININGS [37] identifies the cause of a bug in the context of software evolution by differencing dependence graphs. In contrast, our approach targets only definite protection differences. We also determine differences at the PTFA level and rely on reachability in PTFA models.

IX. Threats to Validity

Threats to internal validity refer to confounding variables that may influence our results. Our results depend on the accuracy of the PTFA engine we used. Because PHP applications like WordPress rely on many dynamic features, the engine relies on sound but conservative approximations, especially for dynamic calls in the call graph, that may lead to spurious paths. In turn, spurious paths may lead to the identification of spurious protection-impacting changes. Consequently, the real set of protection-impacting changes may be smaller than reported. We did not calculate the spurious path rate in this study, but we previously reported a spurious path rate for PTFA on WordPress of $10.96 \pm 3.18\%$ (95% confidence level) [20].

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

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

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

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

X. Conclusion and Future Work

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

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

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

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

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

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

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

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

Developers would need to review security-affected releases. While definite privilege protection information may help in this task, a reduction of candidate root causes is likely to reduce the required effort. In these releases, the median *protection-impacting changes* represent about 27% of total changes. Thus, when re-validating the RBAC security of their applications, developers need not investigate 73% of code changes, because these changes have no impact on protection differences.

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

Further research could also be devoted to investigating the interaction with developers and the visualization of *protection-impacting changes*. An approach about determining explanatory counter-examples [20] could also be used, combined with visualization techniques to supply information to developers.

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

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

References

[1] The Open Web Application Security Project, "OWASP application security verification standard 3.0," https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project, 2015, accessed 2018/01/06.

[2] R. Shirey. (2007, 08) Internet security glossary. Internet Engineering Task Force. [Online]. Available: https://www.ietf.org/rfc/rfc4949.txt

[3] MITRE Corporation. (2015, 12) CWE-425: Direct Request ('Forced Browsing'). [Online]. Available: https://cwe.mitre.org/data/definitions/425.html

[4] The Open Web Application Security Project. (2014, 9) OWASP testing guide v4. [Online]. Available: https://www.owasp.org/index.php/OWASP_Testing_Project

[5] A. H. Karp, H. Haury, and M. H. Davis, "From ABAC to ZBAC: The evolution of access control models," HP Laboratories, Tech. Rep. HPL-2009-30, 2009. [Online]. Available: http://www.hpl.hp.com/techreports/2009/HPL-2009-30.pdf

[6] M. Felderer and E. Fourneret, "A systematic classification of security regression testing approaches," *Int'l J. Software Tools for Technology Transfer*, vol. 17, no. 3, p. 305–319, Jun. 2015.

[7] B. Arkin, S. Stender, and G. McGraw, "Software penetration testing," *IEEE Security Privacy*, vol. 3, no. 1, pp. 84–87, Jan 2005.

[8] G. McGraw, *Software Security: Building Security in*. Upper Saddle River, NJ, USA: Addison-Wesley Professional, 2006.

[9] H. Peine, M. Jawurek, and S. Mandel, "Security goal indicator trees: A model of software features that supports efficient security inspection," in *Proc. 11th IEEE High Assurance Systems Engineering Symp. (HASE '08)*, 2008, pp. 9–18.

[10] N. Qamar, Y. Ledru, and A. Idani, "Evaluating RBAC supported techniques and their validation and verification," in *Proc. Sixth Int'l Conf. Availability, Reliability and Security (ARES '11)*, Aug 2011, pp. 734–739.

[11] A. Edmundson, B. Holtkamp, E. Rivera, M. Finifter, A. Mettler, and D. Wagner, "An empirical study on the effectiveness of security code review," in *Proc. 5th Int'l Symp. Engineering Secure Software and Systems (ESSoS '13)*, 2013, pp. 197–212.

[12] A. Austin and L. Williams, "One technique is not enough: A comparison of vulnerability discovery techniques," in *Proc. Int'l Symp. Empirical Software Engineering and Measurement (ESEM '11)*, 2011, pp. 97–106.

[13] *Software Engineering – Software Life Cycle Processes – Maintenance*, ISO/IEC Std. 14 764:2006, 2006.

[14] M.-G. Lee and T. Jefferson, "An empirical study of software maintenance of a web-based Java application," in *21st IEEE Int'l Conf. Software Maintenance (ICSM'05)*, 2005.

[15] M. K. Davidsen and J. Krogstie, "A longitudinal study of development and maintenance," *Information and Software Technology*, vol. 52, no. 7, pp. 707–719, jul 2010.

[16] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *J. Software Maintenance and Evolution: Research and Practice*, vol. 13, no. 1, pp. 3–30, 2001.

[17] D. Cappelli, A. Moore, and R. Trzeciak, *The CERT Guide to Insider Threats: How to Prevent, Detect, and Respond to Information Technology Crimes (Theft, Sabotage, Fraud)*. Upper Saddle River, NJ, USA: Addison-Wesley Professional, 2012.

[18] C. E. Landwehr, A. R. Bull, J. P. Mcdermott, and W. S. Choi, "A taxonomy of computer program security flaws, with examples," Naval Research Laboratory, Technical Report, 1994. [Online]. Available: http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA465587

[19] D. Baca, B. Carlsson, and L. Lundberg, "Evaluating the cost reduction of static code analysis for software security," in *Proc. 3rd ACM SIGPLAN Workshop Programming Languages and Analysis for Security (PLAS '08)*, 2008, pp. 79–88.

[20] M.-A. Laverdière and E. Merlo, "Computing counter-examples for privilege protection losses using security models," in *Proc. IEEE 24th Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER '17)*, 2017, pp. 240–249.

[21] ——, "Classification and distribution of RBAC privilege protection changes in wordpress evolution," in *To appear in Proc. 15th Int'l Conf. Privacy, Security and Trust (PST '17)*, 2017.

[22] F. Gauthier and E. Merlo, "Alias-aware propagation of simple pattern-based properties in PHP applications," in *Proc. 12th IEEE Int'l Working Conf. Source Code Analysis and Manipulation (SCAM '12)*, 2012, pp. 44–53.

[23] ——, "Fast detection of access control vulnerabilities in PHP applications," in *Proc. 19th Working Conf. Reverse Engineering (WCRE '12)*, 2012, pp. 247–256.

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

[25] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, dec 2008.

[26] WordPress. (2018, 01) Release archive. [Online]. Available: https://wordpress.org/download/release-archive/

[27] T. Preston-Werner. (2013, 06) Semantic Versioning 2.0.0. [Online]. Available: http://www.semver.org

[28] T. B. Muske, A. Baid, and T. Sanas, "Review efforts reduction by partitioning of static analysis warnings," in *Proc. 13th IEEE Int'l Working Conf. Source Code Analysis and Manipulation (SCAM 13)*, Sept 2013, pp. 106–115.

[29] D. Letarte, F. Gauthier, and E. Merlo, "Security model evolution of PHP web applications," in *Proc. Fourth IEEE Int'l Conf. Software Testing, Verification and Validation (ICST '11)*, 2011, pp. 289–298.

[30] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, "Verification and change-impact analysis of access-control policies," in *Proc. 27th Int'l Conf. Software Engineering (ICSE '05)*, 2005, pp. 196–205.

[31] S. Son, K. S. McKinley, and V. Shmatikov, "RoleCast: Finding missing security checks when you do not know what checks are," in *Proc. 26th ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 11)*, 2011, pp. 1069–1084.

[32] ——, "Fix Me Up: Repairing access-control bugs in web applications," in *Proc. 20th Network and Distributed System Security Symp. (NDSS 2013)*, 2013.

[33] J. Hwang, T. Xie, D. El Kateb, T. Mouelhi, and Y. Le Traon, "Selection of regression system tests for security policy evolution," in *Proc. 27th IEEE/ACM Int'l Conf. Automated Software Engineering (ASE '12)*, 2012, pp. 266–269.

[34] Y. Le Traon, T. Mouelhi, A. Pretschner, and B. Baudry, "Test-driven assessment of access control in legacy applications," in *Proc. Int'l Conf. Software Testing, Verification, and Validation (ICST '08)*, 2008, pp. 238–247.

[35] T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: localizing errors in counterexample traces," in *Proc. 30th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '03)*, 2003.

[36] A. Groce and D. Kroening, "Making the most of BMC counterexamples," *Electron. Notes Theor. Comput. Sci.*, vol. 119, no. 2, pp. 67–81, Mar. 2005.

[37] V. S. Sinha, S. Sinha, and S. Rao, "BUGINNINGS: Identifying the origins of a bug," in *Proc. 3rd India Software Engineering Conf. (ISEC '10)*, 2010, pp. 3–12.