2017 IEEE 3rd International Conference on Collaboration and Internet Computing

When Do Changes Induce Software Vulnerabilities?

Manar Alohaly
Dept. of Computer Science and Engineering
University of North Texas
Princess Nourah bint Abdulrahman University
ManarAlohaly@my.unt.edu

Hassan Takabi
Dept. of Computer Science and Engineering
University of North Texas
Takabi@unt.edu

***Abstract*—Version control systems (VCSs) have almost become the de facto standard for the management of open-source projects and the development of their source code. In VCSs, source code which can potentially be vulnerable is introduced to a system through what are so called commits. Vulnerable commits force the system into an insecure state. The far-reaching impact of vulnerabilities attests to the importance of identifying and understanding the characteristics of prior vulnerable changes (or commits), in order to detect future similar ones. The concept of change classification was previously studied in the literature of bug detection to identify commits with defects. In this paper, we borrow the notion of change classification from the literature of defect detection to further investigate its applicability to vulnerability detection problem using semi-supervised learning. In addition, we also experiment with new vulnerability predictors, and compare the predictive power of our proposed features with vulnerability prediction techniques based on text mining. The experimental results show that our semi-supervised approach holds promise in improving change classification effectiveness by leveraging unlabeled data.**

Keywords

Software Security, Software Vulnerabilities, Vulnerability Prediction, Source Code, Semi-supervised Learning

1. Introduction

Version control systems have taken the lead in managing open source software projects [17]. Git, for instance, exceeded 10 million repositories in 2013 [12]. However, this collaborative development environment is not without significant security challenges. The National Vulnerability Database (NVD) reported over 6,000 new vulnerabilities in open source software projects between 2014-2016 [24]. In view of this fact, it is not surprising that security vulnerabilities pose serious impediments to the adoption of open source software components, especially in areas with potential for economic growth [27].

Our interest in tackling this security issue in an open source development context stems from the far-reaching impact of reusing pre-built vulnerable software components. In an ideal world, manually auditing code is deemed to be a best practice to assure a highly secure production system, particularly in an open source environment. That is, in theory, every developer can participate in this manual inspection of code. However, the shortage in expertise, manpower, and time resources remain primary obstacles.

Several techniques and tools have been built to lessen the burden of manually auditing code. Variations of static analysis tools have been designed to search and highlight potential vulnerabilities in source code [2], [3], [4], [5], [8], [9], [10], [15]. However, the need for annotating code manually and configuring rule-sets, paired with long runtime and the high rate of false alarms, undermine trust in the effectiveness of these tools. Other research in security pitfall detection has focused on dynamic analysis and symbolic execution tools [16], [22], [36].

Most of the aforementioned approaches focus exclusively on testing either the entirety of a software project, or its constituent modules as is at certain point in time. However, there is a wealth of historical knowledge made available because of the incremental nature of software development when using VCSs. VCSs are designed to track incremental versions of files in the form of commits. This allows for the exploration of different change patterns, aimed at understanding and learning from prior changes that have previously introduced vulnerabilities to a system, and to detect future similar ones. In this work, we view a vulnerability detection task as a *change* classification problem, in which a machine learning classifier is trained on a history of changes (vulnerable and clean). The rationale behind such vulnerability detection model is to proactively detect changes that might inject vulnerabilities to a system. This will provide the opportunity for a developer to take a corrective step immediately, while (s)he still retains a full mental model of what (s)he has changed. In fact, not only does our proposed model advantage developers, but it also disadvantages attackers, since it shrinks the lifetime of a vulnerability [35].

Several research studies have considered the feasibility of building dependable vulnerability prediction models. One line of research used a variety of software metrics and organizational measures as indicators to discern vulnerable

0-7695-6303-1/17/31.00 ©2017 IEEE
DOI 10.1109/CIC.2017.00020

software components from clean ones [14], [25], [28], [31], [32], [33], [34], [38]. This approach is built upon an implicit assumption that vulnerable software entities are sufficiently distinguishable under a certain set of features. For instance, code complexity is presumed to point to complex software components as more likely to be vulnerable. Another line of vulnerability prediction studies focus on characterizing or modeling software entities with a set of features that capture the pattern of the software component, e.g. bag-of-words [18], [29], [37], without prior expectation of how these features correlate with the presence of vulnerability. While bag-of-words model has shown to be useful in vulnerability prediction studies, the fact that it overlooks the structure of source code makes it error-prone. In this work, we advocate for the second approach. However, as an enhancement, we experimented with another set of features that capture the structure of source code.

The main contributions of this paper are as follows:

1) We view a vulnerability detection task as a *change classification problem*. The key motivation behind this work is to detect vulnerability at a much finer level of granularity, before it slips into the system.
2) We adopt a semi-supervised learning framework to predict vulnerable software changes. To the best of our knowledge, this is the first effort to predict vulnerable software changes using a semi-supervised learning approach.
3) We explore a new feature set to convert software entities, changes in our case, to numerical vectors. This makes them suitable input for machine learning algorithms
4) We compare the predictive power of our feature set against a bag-of-words based model, considering a semi-supervised learning framework.

The remainder of this paper is organized as follows: Section 2 provides a background . In Section 3, we review related work. In Section 4, we introduce the dataset required to build software change classification model. In Section 5,we present the methodology for modelling software changes along with our experiments. We then report the results and discuss limitations in section 6. In Section 7, we conclude our study with the future work.

2. Background

Source code changes: source code changes, also known as hunks or deltas, should be interpreted as source code transformations/modifications applied to files throughout two successive commits in VCS environments. A change could be applied on multiple files; and, a file can be modified at more than one place. Generally, changes are aimed to introduce a new feature to a system, extend an existing one or delete it all together. Changes could also be made for code refactoring or convention adherence. However, a subset of these changes tends to induce bugs and/or vulnerabilities to the production system.

While the concept of vulnerabilities is often considered as a specific class of bugs, it differs in certain aspects. Unlike software bug, vulnerability does not necessarily annoy users while it benefits attackers. It is therefore more likely for a vulnerability to remain undiscovered for a long time before it manifests itself as an undesirable behavior.

Vulnerability inducing changes: it refers to a change that contains suspicious source code with which a vulnerability might slip into to a system.

Vulnerability fixing change: this class of changes refers to commits that clean prior vulnerable code. Vulnerability fixes are commonly provided as extra check conditions for a missing input validation or sanitization.

These two categories of changes are of primary importance in vulnerability prediction studies. Figure 1 shows an example of two changes made on the source file error.class.php. One introduces a vulnerability while the other fixes this vulnerability six years later. During this time period the vulnerability could migrate between several files due to massive renames of files [37]. This is a significant factor that hinders the adoption of SZZ-type [21] of algorithm to automatically identify vulnerability inducing changes. The thing that motivates the decision of using Walden's dataset [37] as discussed in the following sections.

3. Related Work

In this work, we borrow the notion of change classification from the literature of bug detection to further explore its applicability to predict software vulnerabilities. We also adopt a new feature set to represent changes in source code and build our model. We therefore categorize related work into two areas: change classification and its application to bug detection, and the application of machine learning techniques to the domain of software security vulnerability prediction.

3.1. Mining Software Changes

An accurate software *change classification model* should answer this question: will this commit/change force the system into an undesirable state? Kim et al. first proposed the concept of change classification to detect the presence of bugs in software changes. They trained a classifier (SVM) on features extracted from the revision history of 12 open source projects. The resulting model can classify changes as buggy or clean, with a 78% accuracy and 60% recall [20]. Kims approach of change classification follows the classical steps of any machine learning task. It involves creating a corpus, training, and testing.

Creating a corpus: The revision history of a project is mined to identify bug-fix changes by searching through change log messages for fix-related keywords. Once the fix-change is found and localized at the file level, the corresponding bug-introducing commit is identified by

traversing the revision history backward using a SZZ-type of algorithm, described in [21]. Each change is labeled as either buggy or clean. Then, features are extracted to convert changes to numerical vectors suitable for machine learning algorithms. This approach works well for constructing a corpus required to predict buggy changes [20], [21]. However, for identifying vulnerability introducing changes, Kims-type of approach wouldn't be as effective. In a great majority of cases in the corpus we used, vulnerable code was modified multiple times before being fixed. And, the fixes themselves do not necessarily modify code, but rather add some check conditions. Thus, traversing the revision history back starting from fix commits is not guaranteed to be feasible. Which is to say that the introduction of the vulnerability would wrongly be traced to its most recent modification. Hence, the availability of dataset-similar in nature to the one used in [21]- challenges the reliable transformation of change classifications concept to vulnerability prediction domain. In our work, we derived our dataset from a hand-crafted dataset of vulnerabilities in PHP web applications [37]. For more details, please refer to dataset section.

Training and Testing: The crafted carpus is used to train the change classification model which will later be tested to determine whether a new change is more likely to be a buggy or clean change. Similar in spirit to change classification is VCCFinder, a work that trains a model to flag vulnerable commits [28]. This work overlaps ours in that both learn from vulnerability introducing commits. In contrary to this prior effort, our work is built upon gold standard dataset constructed by experts. Another distinction is that our approach learns and classifies the transition pattern between successive commits, while VCCFinder classifies static code on a particular commit.

CVE-2013-4998
Summery: phpMyAdmin 3.5.x before 3.5.8.2 and 4.0.x before 4.0.4.2 allows remote attackers to obtain sensitive information via an invalid request, which reveals the installation path in an error message, related to pmd_common.php and other files.

Commit: #0fea53b **(Vulnerability fixing change)**
Title: Move the PHPMYADMIN constant definition earlier
Date: Jul 11, 2013

```
* @package PhpMyAdmin */
+  if (! defined('PHPMYADMIN')) {
+          exit;
+      }
+            /**
               * base class
              */
              ...
```

Commit: #e623dc4 **(Vulnerability inducing change)**
Title: new error handler - needs some more work, but works already
Date: Oct 8, 2007

```
+   public function displayBacktrace()
+   {
+      foreach ($this->getBacktrace() as
    $step) {
+         echo
    PMA_Error::relPath($step['file']) . '#' .
+    $step['line'] . ': ';
...
```

Figure 1. phpMyAdmin CVE-2013-4998 vulnerability disclosure followed by its fixing and inducing changes

3.2. Vulnerability Prediction Models

The main goal of most software vulnerability prediction studies is to classify software components as either clean or vulnerable. To perform the prediction, original software data (file, module, function, operation etc.) must be mapped to metrics by means of some feature extraction techniques. Prior vulnerability prediction research includes three different approaches for feature extraction.

One approach depends on encoding a prior knowledge of the root causes of vulnerabilities and their potential counter measures. Shar et al, for instance, have used program slices to build static code attributes that characterize code patterns which have been known as common vulnerability counter measures [30]. They have mined input validation and sanitization code patterns to build a classification model that flags operations that might lead to potential SQL injection, or cross site scripting in source code. Medeiros et al, have followed the same approach for feature selection: they used the frequency of occurrence of operations that have been particularly designed as mechanisms to mitigate web application vulnerabilities [23]. The context in which they applied their model is different, though. They used it as a complementary step after running a static analyzer a on web application to filter false positive alarms.

A second approach for feature selection is based on correlating the presence of vulnerabilities with properties of source code, e.g. cyclometric complexity, or process metrics, that capture several aspects of the development process [14], [25], [31], [32], [33], [34], [38]. The rationale behind this approach is that, for instance, the more complex the code, the more vulnerability-prone it can be.

A third feature extraction technique relies less on prior knowledge or assumptions about how vulnerable code might look like, but rather on what are so-called brute force features to represent source code. An initial exploration of the latter approach is credited to Hovsepyan et al. [18]. Hovsepyan et al. viewed programming language as a human language, hence the analysis of vulnerable source code by means of text mining techniques. Building upon this observation, bag-of-words has been used as a representation

of source code in several vulnerability prediction studies [18], [29], [37]. While the predictive performance of the bag-of-words model has exceeded the prediction power of software metrics [37], the fact that it overlooks the structural features of source code reduces its robustness. In this work, we propose a new feature set similar to bag-of-words in the sense that it is less constrained by prior expectation. Unlike bag-of-words, our feature set captures structural and some other lexical features- beyond term frequencies- of source code.

4. Dataset

To establish a ground truth, we derive our dataset from the corpus built and released by Walden et al [37]. It contains data on security vulnerabilities found in three open-source PHP web applications: phpMyadmin, Drupal, and Moodle. Generally, it includes information on the commits at which a vulnerability was introduced and fixed. Authors mined a random sample of security advisories presented in public vulnerabilities database e.g. NVD. From there they manually mapped each advisory to the Git commit that fixed the vulnerability. They, then, traversed the revision history back until the fixed exploit was impossible before the commit. This point is deemed to be the introduction of the vulnerability. Figure 1 shows an example of an entry in walden's dataset.

The introduction of vulnerabilities and their fixes were verified and manually traced back to the file that contained the vulnerability along with the file that provided the corresponding fix at each revision/commit of the software. The resulting vulnerabilities dataset contained in total 75 vulnerabilities for phpMyadmin, 51 for Moodle, and 97 for Drupal. In the case of Drupal, vulnerabilities were not mapped to specific commits/revisions where they were originally introduced. Since identifying vulnerability inducing commits is a corner stone for this study, Drupal was excluded from our experiment. From the list of vulnerability introducing commits along with the fix (clean) commits provided by [37], we extracted - using JGit [6] a source code change history of both vulnerable and clean changes in phpMyadmin and Moodle. Since the number of these manually annotated changes is relatively small, this dataset as is is less likely to be truly representative. Hence, it is insufficient to build a dependable model. To remedy this shortage, we retrieve a random sample of change history of commits that span the period between oldest vulnerable commit and most recent fix for each application. The latter unlabeled set of changes will be used in a conjunction with the labeled dataset within a semi-supervised learning framework.

5. Methodology

Building a vulnerable change detector requires histories of vulnerability inducing changes along with the corresponding fixes. Because the dataset contains fairly small number of manually annotated software changes/commits (either clean or vulnerable), we adopt a semi-supervised learning approach that requires minimal supervision in order to initiate the learning algorithm. Particularly, we use self-training expectation-maximization learning process similar to that described in [26].

The algorithm first trains a classifier with only the available labeled vulnerable and clean changes. Then, it uses the learned classifier to assign the most likely class labels to each unlabeled changes by calculating the expectation of the missing class labels. It then trains a new classifier using all the changes, including both the originally and the recently labeled ones and then iterates. Features used to train this model are discussed in the following subsection.

5.1. Feature set

Source code changes must be transformed to numerical vectors that can be fed into machine learning algorithms. As described earlier in section 3.2, vulnerability prediction studies can be categorized into three classes by the means of feature extraction techniques. Our proposed feature set belongs to the third type, in the sense that the features are deemed to represent source code without making any expectations or assumptions of how these predictors correlate with the presence of vulnerabilities. Unlike the prior work [18], [29], [37] that relies solely on text mining techniques which ignore source code structural information, our work focus on capturing different structural aspects of source code, as well as capturing some lexical aspects. These indicators, among others, have shown effectiveness in capturing coding style to prove author attribution of a source code [13]. We conjecture that representing software changes/commits with such indicators will help in capturing the coding style of clean and vulnerable changes. Hence, a model trained on a history of vulnerable and clean changes will be able to flag future vulnerable ones.

Since each software change includes two versions of a file, before and after committing a change, we compute the features for both versions of files. Then, a change is represented using the difference in values between the two versions. This representation is deemed to capture structural changes in source code that might lead to vulnerability. Table 1 summarizes our feature set. Computing these features (except the last three) requires access to an abstract syntax tree (AST). We used PhP_Parser [7], to obtain the AST representation of PhP source code.

To assess the prediction quality of our proposed indicators, we compare it with the bag-of-words model. We chose to compare against this particular model because both are neutral to the presence of vulnerability.

5.2. Experiment

The primary goal of this experiment is to address the following research questions:

1) Does our semi-supervised change classification perform better than the baseline?

TABLE 1. THESE FEATURES ARE COMPUTED FOR EACH FILE BEFORE AND AFTER A COMMIT TO CAPTURE A CHANGE THAT HAS BEEN MADE

Featur	Description
MaxDepth	The length of a path towards the deepest node in AST.
AvgNodeDepth	This feature computes the average depth of each node type in the AST. Analyzing source code files in our dataset produced 89 different node types.
NestingDepth	The length of deepest nested control and loop structures.
BranchingFactor	Average branching factor of AST.
Ln(comments/len)	Natural logarithm of the number of comments over length of file in characters.
Ln(strLiterals/len)	Natural logarithm of the number of strings over length of file in characters.
Ln(functions/len)	Natural logarithm of the number of functions over length of file in characters.
AvgParameters	The average number of parameters per function.
AvgLineLen	The average length of lines in characters.
SDLineLen	The standard deviation of number of characters in lines of code.
Ln(tokens/len)	Natural logarithm of the number of words/tokens over length of file in characters.

2) How does the predictive power of our feature set compare to prediction techniques based on text mining?
3) Given the available, manually annotated data, can unlabeled examples be utilized to improve the learning process?

To answer these questions, we performed a self-training classification experiment twice on both web applications. In the first experiment, we used our proposed feature set to build the model, while the term frequencies and bag-of-words approach was used in the second one. We then repeated each experiment 15 times with varying amounts of unlabeled data, as shown in Figures 2-3, to gain insight into how models react to the unlabeled examples.

For labeled data, we considered the list of vulnerability introducing commits and corresponding fixes provided by Walden [37]. As mentioned previously, each vulnerability along with its fix is mapped to a particular file in a particular commit. To capture a change that either leads to or cleans a vulnerability, we applied feature extraction techniques as described in section 5.1. This was performed on each file that contained either the vulnerable code or the fix, before and after the commit. The difference between the two values represents a change.

For unlabeled data, we mined a random sample of commit histories for source code changes that span the period between the oldest vulnerable commit, and most recent fix, for each application. Then, the same feature extraction techniques were applied to these unlabeled changes.

We performed stratified sampling to split labeled data, roughly 20% for testing and 80% for training. Next, using Weka CollectiveEM Classification (with J48 classifier and C=0.25 , M=2 parameters) [1], we iteratively applied self-training classification where each iteration contains an increased increment of the unlabeled data.This repetitive process can be terminated after reaching a predetermined number of iterations, running out of unlabeled data, or after convergence [19]. Throughout this study, we followed the first stopping criterion. We initiated the self-training process with zero unlabeled changes in the first iteration, and ended it with 5000 unlabeled changes in the last iteration. Figures 2-3 show the number of unlabeled examples in each iteration.

TABLE 2. NUMBER OF VULNERABLE AND CLEAN CHANGES USED IN TRAINING AND TESTING FOR BOTH WEB APPLICATIONS

	phpMyadmin	Moodle
Training vulnerable changes	51	41
Training clean changes	55	41
Testing vulnerable changes	20	10
Testing clean changes	20	10

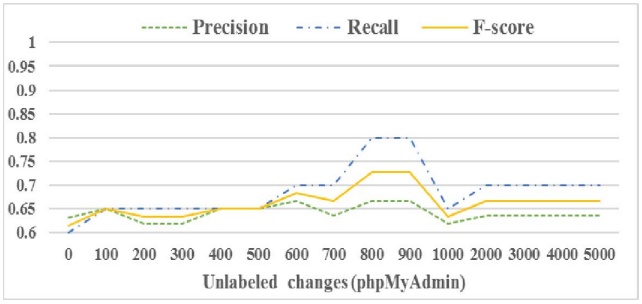
6. Experimental Results

Information required to interpret the results are provided in Table 2. It includes the number of vulnerable changes, and clean changes (fixes) that are used for training and testing in both web applications. It should be mentioned that in the case of phpMyadmin, we could not find four of the vulnerable files in the Git repository. As a result, the total of vulnerable changes is 71 rather than 75, as reported by Walden [37]. As can be seen from Table 2, the class variable has almost equal distribution for both phpMyadmin and Moodle. However, there is no guarantee that a random sample of unlabeled examples will have similar distribution of vulnerable and clean changes. This potential inconsistency between modeling assumptions and the actual model that generates the data will make the task of self-training more challenging.

Figures 2-3 show performance results obtained from 15 iterations of the self-training change classification applied on both web applications using two different models. The performance is evaluated by the means of precision, recall and F-score. Generally, the prediction techniques based on the text mining perform better when considering the labeled data only. As we add more unlabeled data, the performance peaks when using our feature set as indicators on the case of phpMyadmin, while with Moodle it peaks when using the bag-of-words model. After 15 iterations of adding unlabeled data, both models exhibit almost identical performance for each application.

6.1. Discussion

Here we will revisit our three research questions mentioned earlier in section 5.2.



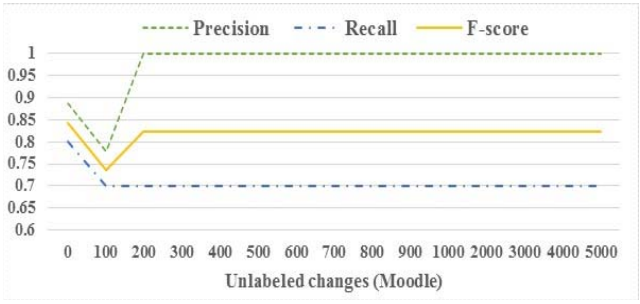
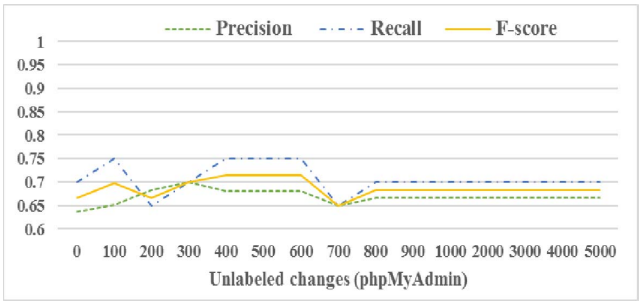


Figure 2. Change classification performance by the means of precision, recall, and F-score when using the proposed feature set



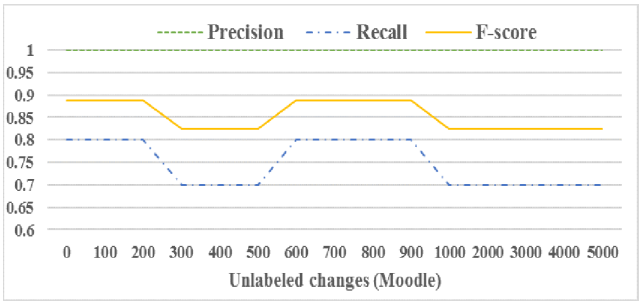


Figure 3. Change classification performance by the means of precision, recall, and F-score when using the bag-of-words model

Does our change classification perform better than the baseline?

Both the precision and recall of a random guess classifier would be 0.5 since it is a binary classification problem. Thus, the performance of both change classification models on both datasets surpass the baseline. A semi-supervised change classification predicts vulnerable changes at recall ranges from 0.6 to 0.8 and precision from 0.63 to 1.

How does the predictive power of our feature set compare to prediction techniques based on text mining?

Generally, the bag-of-words model yields better performance as suggested by the box-plot comparison of the models' F-measures shown in Figure 4. The analysis is statistically significant p = 0.003 (Moodle) and P= 0.01 (phpMyAdmin) as reported by single factor ANOVA test. However, the stability of the learning process across successive increments of unlabeled data is of a significant importance when considering a semi-supervised learning framework. As can be seen from Figure 3, the recall of bag of words model tends to fluctuate across consecutive iterations. This indicates that such a model may not have the ability to generalize knowledge gained from labeled examples alone. On the other hand, the recall of the model built upon our feature set, shown in Figure 2, exhibits steady performance after it peaks at the third iteration (addition of 200 unlabeled examples) in the case of Moodle, while it shows constant improvement when applied to phpMyadmin up to the tenth iteration. This suggests that our proposed indicators are more appropriate for semi-supervised learning given the available seeding.

It should be mentioned that for both applications, and using both models, the addition of unlabeled examples after the tenth iteration does not yield any improvement. In fact, in some cases it inflicts poor prediction performance. This suggests that the ratio of labeled to unlabeled examples affect the classifiers stability. A large amount of unlabeled data paired with a very small amount of labelled data hurts performance, as this introduces too much uncertainty to the learning process.

Given the available manually annotated data can unlabeled examples be utilized to improve the learning process?

The main motivation for following a semi-supervised learning framework originated from the fact that the fully supervised classifier, built upon the available manually annotated data, is relatively weak due to small seed size. As a result, its predictions may not be reliable. A semi-supervised learning approach is deemed to address this limitation. However, a major difficulty that can hinder its performance is the inconsistency of class variable distribution between labeled and unlabeled examples. The learning process in a semi-supervised setting becomes even more challenging when the knowledge of the underlying unlabeled data distribution is only available via the training labeled sample. This suggests the selection of unlabeled examples play a crucial role in semi-supervised learning,

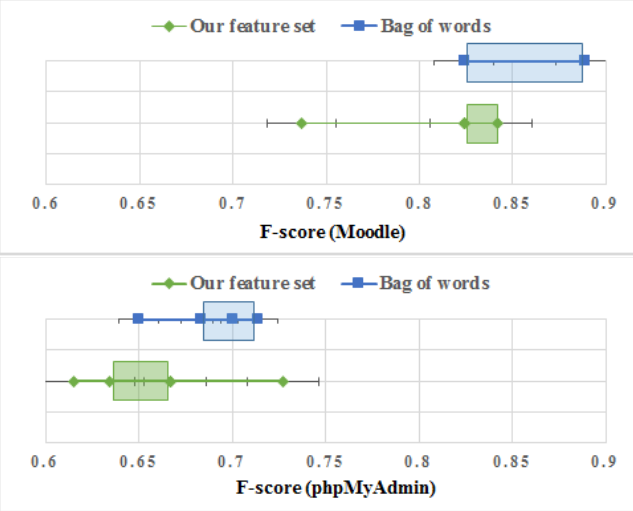


Figure 4. F-score across the 15 iterations of self-training of both models when trained on the change histories of Moodle and phpMyAdmin. The box indicates the upper/lower quartiles ,while the dotted crossbars indicate the maximum/minimum values

especially a self-training approach. The selection can be made by a preselection strategy [11], or simply by following the original order of the unlabeled data. We follow the latter approach to establish a baseline, and to gain insights into the utility of using unlabeled data. Empirically we have shown that including unlabeled data can render better predictions. We conclude that a semi-supervised learning approach has the potential to surpass the performance of a fully supervised learner.

Threats to Validity: The fact that the experiment is done with only two web applications might hurt the validity of the results. However, there is no available vulnerability dataset that we are aware of that contains all information needed for such study, i.e., vulnerability introducing commits and fix commits.

7. Conclusion and Future Work

This paper has studied the transformation of the notion of change classification from the literature of bug detection into the area of vulnerability prediction. The rationale behind such a model is to proactively detect vulnerable code snippets. Hence a developer knows that a change (s)he just made would probably inject a vulnerability into a system. The shortage in data challenges a reliable transformation of change classification into software vulnerability domain. We therefore adopted a semi-supervised framework to leverage the wealth of unlabeled data. Our semi-supervised change classification predicts vulnerable changes at recall ranges from 0.6 to 0.8 and precision from 0.63 to 1.

The currently available- manually crafted- dataset provides a gold standard to begin studying vulnerable software changes. In the future, we plan to study automated or semi-automated approaches for data collection which is critically important for a large-scale coverage of software security vulnerability.

References

[1] collective-classification-weka-package. [Online]. Available: https://github.com/fracpete/collective-classification-weka-package

[2] Coverity scan static analysis. [Online]. Available: https://scan.coverity.com/

[3] Flawfinder. [Online]. Available: https://www.dwheeler.com/flawfinder/

[4] Hp fortify. [Online]. Available: https://www.hpfod.com/

[5] Ibm security appscan source. [Online]. Available: https://www.ibm.com/software/products/en/appscan-source/

[6] Jgit. [Online]. Available: https://eclipse.org/jgit/

[7] Php-parser. [Online]. Available: https://github.com/nikic/PHP-Parser

[8] Prefast analysis tool. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms933794.aspx

[9] Rough auditing tool for security (rats). [Online]. Available: https://code.google.com/archive/p/rough-auditing-tool-for-security/

[10] Splint annotation-assisted lightweight static checking. [Online]. Available: http://splint.org/

[11] S. Abney, "Semisupervised learning for computational linguistics."

[12] D. Brian, "10 million repositories," Dec 2013. [Online]. Available: https://www.gartner.com/doc/2607617

[13] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry," in *24th USENIX Security Symposium (USENIX Security), Washington, DC*, 2015.

[14] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of Systems Architecture*, vol. 57, no. 3, pp. 294–313, 2011.

[15] CodeSonar. Grammatech static analysis. [Online]. Available: https://www.grammatech.com/codesonar/

[16] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code," in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 281–290.

[17] D. M. German, B. Adams, and A. E. Hassan, "Continuously mining distributed version control systems: an empirical study of how linux uses git," *Empirical Software Engineering*, vol. 21, no. 1, pp. 260–299, 2016.

[18] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden, "Software vulnerability prediction using text analysis techniques," in *Proceedings of the 4th international workshop on Security measurements and metrics*. ACM, 2012, pp. 7–10.

[19] Z. Kaljahi and R. Samad, "Adapting self-training for semantic role labeling," in *Proceedings of the ACL 2010 Student Research Workshop*. Association for Computational Linguistics, 2010, pp. 91–96.

[20] S. Kim, E. J. Whitehead Jr, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.

[21] S. Kim, T. Zimmermann, K. Pan, E. James Jr *et al.*, "Automatic identification of bug-introducing changes," in *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE, 2006, pp. 81–90.

[22] G. Lu and S. Debray, "Automatic simplification of obfuscated javascript code: A semantics-based approach," in *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*. IEEE, 2012, pp. 31–40.

[23] I. Medeiros, N. F. Neves, and M. Correia, "Automatic detection and correction of web application vulnerabilities using data mining to predict false positives," in *Proceedings of the 23rd international conference on World wide web*. ACM, 2014, pp. 63–74.

[24] P. Mike. (2016, May) 5 steps to better open source code security. [Online]. Available: http://www.itproportal.com/2016/05/29/5-steps-to-better-open-source-code-security/

[25] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 529–540.

[26] K. Nigam, A. K. McCallum, S. Thrun, and T. Mitchell, "Text classification from labeled and unlabeled documents using em," *Machine learning*, vol. 39, no. 2, pp. 103–134, 2000.

[27] N. Nissanke, "Component security-issues and an approach," in *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, vol. 2. IEEE, 2005, pp. 152–155.

[28] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 426–437.

[29] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2014.

[30] L. K. Shar and H. B. K. Tan, "Predicting common web application vulnerabilities from input validation and sanitization code patterns," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 310–313.

[31] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.

[32] Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2008, pp. 315–317.

[33] ——, "Is complexity really the enemy of software security?" in *Proceedings of the 4th ACM workshop on Quality of protection*. ACM, 2008, pp. 47–50.

[34] ——, "Can traditional fault prediction models be used for vulnerability prediction?" *Empirical Software Engineering*, vol. 18, no. 1, pp. 25–59, 2013.

[35] F. Stefan. (2013) The known unknowns empirical analysis of publicly unknown security vulnerabilities.

[36] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in c++," in *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*. IEEE, 2015, pp. 46–55.

[37] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," in *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*. IEEE, 2014, pp. 23–33.

[38] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. IEEE, 2010, pp. 421–428.