VulDigger: A Just-in-Time and Cost-Aware Tool for Digging Vulnerability-Contributing Changes

Limin Yang*, Xiangxue Li*† and Yu Yu‡
*Department of Computer Science and Technology, East China Normal University, Shanghai, China
†Westone Cryptologic Research Center, Beijing, China; National Engineering Laboratory for Wireless Security, XUPT
‡Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China
Email: lmyang@stu.ecnu.edu.cn, xxli@cs.ecnu.edu.cn, yuyu@yuyu.hk

***Abstract*—It has been widely adopted to minimize the maintenance cost by predicting potential vulnerabilities before code audits in academia and industry. Most previous research dedicated to file/component-level vulnerability prediction models is coarse-grained and may suffer from cost-prohibitive and impractical security testing activities. In this paper, we focus on a cost-aware vulnerability prediction model and present a just-in-time change-level code review tool called VulDigger to dig suspicious ones from a sea of code changes. Our contributions benefit from the case study of Mozilla Firefox by constructing a large-scale vulnerability-contributing changes (VCCs) dataset in a semi-automatic fashion. We then further manifest a classification tool with a mixture of established and new metrics derived from both software defect prediction and vulnerability prediction. Consequently, the precision of such tool is extremely promising (i.e., 92%) for an effort-aware software team. We also examine the return on investment by training a regression model to locate most skeptical changes with fewer lines to inspect. Our findings suggest that such model is capable of pinpointing 31% of all VCCs with only 20% of the effort it would take to audit all changes (i.e., 55% better than random predictor). Our outputs can assist as an early step of continuous security inspections as it provides immediate feedback once developers submit changes to their code base.**

I. Introduction

Code audits and security testing have been cost-prohibitive processes since most people today don't test software until it gets into the deployment phase of its life cycle and such practice has been proved ineffective to locate vulnerabilities or security bugs. Considering the disastrous consequence an exploited vulnerability could cause, e.g., Heartbleed [1], and to reduce the inspection effort, researchers have proposed a multitude of vulnerability prediction models for assisting and prioritizing code audits [2], [3], [4], [5].

Most of these studies focus on predicting vulnerable-prone modules (i.e., files or components) and can be beneficial in some contexts. However, these predictions are generally made too late. One of the suggested solutions is to apply security testing on each phase of the development cycle. Early detection is desirable as the later it gets into testing, the higher the cost of finding and fixing a vulnerability would be.

Change-level predictions. Therefore, some researchers introduced change-level prediction methods and concentrated on predictions of vulnerability-contributing commits/changes (VCCs) [6]. Similar to the field of software defect prediction, the advantages of change-level predictions are [7]: (1) Prediction is suitable for code snippets and thus smaller regions of code needs inspection instead of huge files/components. (2) Developers that are responsible for VCCs can easily be traced and they can assist security experts or fix the security bugs by themselves with all design decisions fresh in their minds. (3) Predictions are made early and just-in-time as immediate feedback is given once a change is submitted to the code base.

Challenges for change-level predictions. To our best knowledge, no one has performed change-level vulnerability predictions except [6], we attribute it to the following two challenges:

- *The lack of a ground-truth dataset.* It's arduous to determine which code changes that indeed induced a vulnerability due to the multiplicity of code changes. Therefore, building a VCC ground-truth dataset is challenging and requires considerable human effort.
- *The disorderly structure of code changes.* Code changes could not retain the original structure and integrity like files or components, hence many established measures (e.g., code complexity, coupling, and cohesion) and commercial analysis tools (e.g., Understand C++) are not directly applicable.

Perl et al. [6] analyzed 66 open-source projects in GitHub and presented a database with 640 VCCs. Compared to Flawfinder [8], their results reduced many false positives with same recall. However, they didn't consider the actual effort in code audits as some VCCs are huge (i.e., with thousands of lines of modifications).

Our contributions. We therefore build a cost-aware change-level vulnerability prediction model based on the code churn of a change. Through the study of Mozilla Firefox project — one of the most vulnerable open-source projects and has been the target in a plethora of vulnerability studies yet most of them focus on file/component-level predictions, our contributions can be outlined as follows:

- We present a change-level code review tool – VulDigger, to flag suspicious code changes immediately on the time of submitting by deriving features from software defect and vulnerability prediction models along with some new metrics (e.g., the maximum changes has been made in the past for files modified in a change). The precision of such tool is extremely promising (i.e., 92%) for a cost-aware software team.

- Besides, we further develop a regression model to locate most skeptical changes with fewer lines to inspect. Such model is capable of pinpointing 31% of all VCCs with 20% of the effort it would take to audit all changes. We notice that six percent of the effort provides the best return on investment as well.
- Finally, we improve the algorithm of mapping vulnerabilities to VCCs by placing more constraints to remove false alarms based on Perl's study [6] and then construct a large-scale dataset for Mozilla Firefox. The dataset contains 178,515 code changes, 1,203 VCCs, and 626 vulnerabilities. To our best knowledge, no large-scale of such dataset has been available for Mozilla Firefox.

Organization. Section II introduces related work. In Section III we elaborate the research experiments and methodology. We present the results and evaluate the performance of our tool in Section IV. Finally Section V reports the limitations and threats to validity.

II. Related Work

Finding potential vulnerabilities of software has gained much attention from both academia and industry as it is a fundamental and crucial problem in the field of computer security. In this section, we review the prior studies focused on vulnerability prediction models and risky code changes.

The concept of vulnerable components was first proposed by [2] and they explored what patterns of imports and function calls would mostly introduce vulnerabilities using frequent pattern mining on early versions of Mozilla project. They successfully leveraged support vector machine techniques to predict vulnerabilities and reported an average precision of 70% and recall of 45%.

Then a wealth of research was conducted on vulnerable function/file/component predictions. Code complexity was first examined whether it can be an indicator of vulnerabilities in Shin and Williams's work [3].

Although they suggested that there is a slightly weak correlation between code complexity and vulnerabilities with regard to the Mozilla Javascript Engine, Shin et al. [4] further evaluated the relationship of complexity, code churn, and developer activity metrics with vulnerabilities. They performed logistic regression to achieve an average recall of 80% as well as a false alarm of 25% on both Mozilla Firefox and Red Hat Enterprise Linux Kernel. Going beyond traditional metrics, other research utilized features like text mining [5], [9].

There are only a few studies focused on finding vulnerabilities on the level of code changes. Meneely et al. [10] first introduced vulnerability-contributing commits and explored their properties like code churn, interactive churn, and dissemination in the Apache HTTP project. They chose "git bisect" instead of "git blame" to perform the mapping from vulnerabilities to VCCs while the former approach requires security test cases thus unavailable in our study. Bosu et al. [11] also examined the various characteristics of VCCs like the type of vulnerabilities, code churn, and developer experience and employment on ten popular open-source projects.

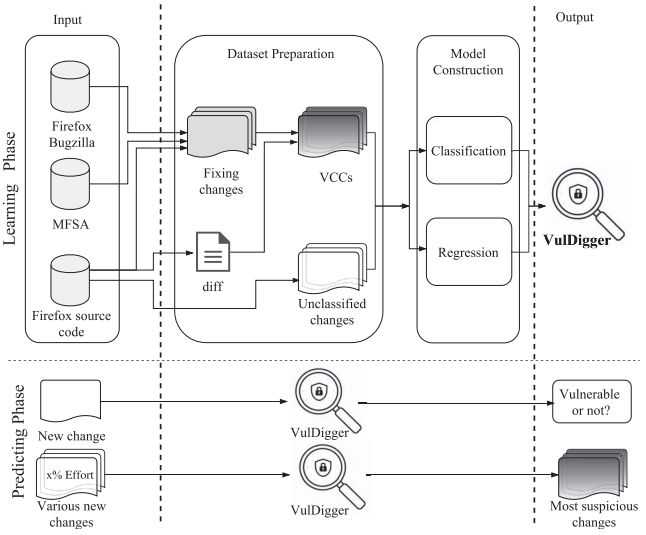


Fig. 1. Overview of VulDigger: The learning phase constructs VCCs and unclassified changes dataset and then feeds them into the classification problem as well as the regression model. The output tool VulDigger then predicts whether a new change is vulnerable and how many VCCs could be found from various new changes with limited resources.

The closest work to our own is Perl et al. [6], who combined repository metadata with text features to feed into a generalized bag-of-words model and reported a precision of 60% and recall of 24%. Such a model is capable of reducing much false positives whereas may suffer from inspecting huge commits. We adopt an effort-aware code churn based model to comprise this problem and find surprisingly that a significant review effort could be saved.

III. Experiments and Methodology

In this section, we describe an overview of VulDigger, how a database of VCCs from Mozilla Firefox is created and which code change measures that have been derived. We detail the data extraction and present both statistical analysis and prediction techniques we have applied.

A. Overview

Fig. 1 gives an overview of our tool VulDigger with two phases: a *learning phase* and a *predicting phase*. In the *learning phase*, Firefox Bugzilla, Mozilla Foundations Security Advisories (MFSA) [12], and Firefox source code (Git Based) are analyzed to retrieve a list of fixing changes. With the diff generated from git version control system, we are capable of mapping fixing changes to corresponding VCCs. VCCs along with unclassified changes are then fed into developing a classification and regression model and output our tool VulDigger. In the *predicting phase*, when a new change is submitted, VulDigger would immediately report whether it is suspicious; when various new changes need to be reviewed, VulDigger presents how many VCCs could be found with x percent of total human effort.

B. Vulnerability-Contributing Changes

In order to distinguish code changes that contribute to a vulnerability from ordinary changes, we first need to locate

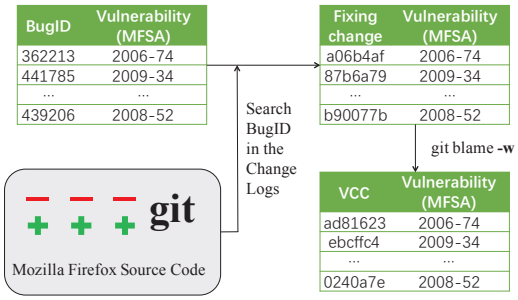


Fig. 2. Mapping vulnerabilities to corresponding VCCs

those vulnerable code changes. The dataset released by Perl [6] is an in-process database instead of the eventual data. Thus we manage to create a VCC dataset of Firefox as it has been studied in many previous file/component-level prediction models, yet no large-scale of such dataset has been available for Firefox to our knowledge.

Prior work of mapping defects/vulnerabilities to defect-prone/vulnerability-contributing changes [10], [7], [6] followed the same principles by SZZ algorithm [13] with different implementations. We comply with this standard and adopt a slightly different method for semiautomatically mapping vulnerabilities to VCCs based on Perl et al.'s methodology [6].

The process of mapping a vulnerability documented in MFSA to corresponding VCC(s) is illustrated in Fig. 2, and the full details can be described as follows:

1) Find the identifier of a bug responsible for that vulnerability based on the dataset shared by [14].
2) Take bug identifier 362213 for example, search for keywords like "fix 362213", "362213", "Bug 362213", and "b=362213" in the change logs to locate fixing change and manually check if it is indeed a fixing change instead of simply adding extra tests for the bug or beautifying the code.
3) For each fixing change, we use **"git blame -w"** for mapping it to one or several VCCs.
   - Ignore changes in tests, documentation, comments, empty lines and not c/c++ files.
   - For each deletion, blame the line that was deleted (the deleted line has to be accessed from the change prior to the fixing change).
   - For every single line inserted, if keywords like "if", "goto", "else", "return", "sizeof", "break", "NULL" or function calls are included, blame one line before and after the line.
   - For every continuous block inserted, we blame one line before and after the block if it is not a function statement as function statement can be inserted anywhere.
4) Eventually, mark the change as VCC that has been blamed most in the steps above. If multiple changes were blamed for the same times, we manually check which one or several changes are VCCs.

We choose MFSA [12] as it's the official database and vulnerabilities have been acknowledged by Mozilla Foundation. Compared to [6], we place more constraints when blaming and in the last step, we manually examine the results rather than blaming them both. What's more, [6] didn't consider the **"-w"** option in the command **git blame**. The **"-w"** option means to ignore whitespace changes when deciding where the lines came from. Without the option, the marked VCC may simply beautify the vulnerable code instead of actually introducing it[1].

To measure the accuracy of such mapping, we randomly select 112 pairs of {fixing change, VCC} and find only 5 (i.e., 4.5%) changes labeled wrongly, possible reasons are that some fixing changes only added functions without explicitly invoking them and hence matched VCCs are not traceable[2]. Since our prediction model can deal with noise, this error rate is tolerable. Nevertheless, improving the automatic mapping still remains as an interesting work.

At this point, we finally build a large dataset consisting of 626 vulnerabilities and 1,203 VCCs. Note that the number of vulnerabilities is less than that of VCCs as some vulnerabilities have various fixes thus mapped to multiple VCCs.

C. Change Measures

To predict whether a change would contribute to a future vulnerability, we adopt various established metrics along with some new metrics shown in Table I as many of them perform well in either defect prediction research or vulnerability prediction models.

Table I lists each measure with its name, full description, the rationale it based on, and the approach how we extract it from the code base. These features are grouped into two dimensions: code and metadata. For brevity reasons, we only explain a few factors here. Most of previous research didn't differentiate TA/TD from CA/CD, while here we consider other modifications of files not written in C/C++ also supplement the complexity of a change. "Past changes" and "Past developers" metrics have been utilized in [6] without considering the distribution of them across different files thus we supplant them with APC, MPC, APD, and MPD. Since extracting functions or variables from unprocessed C/C++ code is difficult without dynamic parsing, similar to [2], [19], we use several regular expressions to identify NRC, NAC, NRFC, NAFC, NRF, NAF, NRVA, and NAVA in an automated fashion.

D. Statistical Analysis of Measures

Before fitting the aforementioned code change measures to the learning-based model, we conduct several statistical analysis to determine whether there is a statistically significant difference between these measures related to VCC and unclassified changes.

Fig. 3 illustrates the box plots of log-scaled values between VCCs and unclassified changes for four representative measures: CA, SLOC, APC, and EXP. Here, value of 0 has been substituted with 0.1 for the log transformation. As illustrated

[1]One of the fixing changes in VCCFinder's database is *https://github.com/lxc/lxc/commit/67e5a20a* and the inappropriately blamed VCC is *https://github.com/lxc/lxc/commit/c414be25*.

[2]https://github.com/mozilla/gecko-dev/commit/a332f016

TABLE I
OVERVIEW OF CODE CHANGE MEASURES (PARTIALLY ADAPTED FROM [7])

Dim.	Name	Description	Rationale and Related Work	Extraction
Metadata	**CA, CD**	Lines of C/C++ code Added/Deleted	Code churn measures have been broadly applied in defect/vulnerability prediction [4], [15], [6].	By parsing the patch of the change
	TA, TD	Total lines of code Added/Deleted	Other languages of lines added/deleted also supplement the complexity of a change.	GitHub API [16]
	SLOC	Average source lines of code before the change.	Larger files are more complex and maybe more defect-prone [17].	https://github.com/flosse/sloc
	NF, ND, NS	Number of Files/Directories/Subsystems (in C/C++) modified	Changes touching many files/directories/subsystems are more likely to be vulnerability-prone [18].	Subsystem is defined as the root directory.
	Entropy	Distribution of modified code across each file	Higher entropy means the change is more fragmented and thus more likely to be defect/vulnerability prone [7], [6].	$E(P) = -\sum_{k=1}^{n}(p_k * log_2 p_k)$ [7]
	AGE	Average time elapsed since last change	More recent changes contribute to more defects.	Git rev-list subcommand
	EXP	Developer experience	Developers contribute little to the project may significantly increase the defect/vulnerability probability [18], [10].	Number of changes the developer has made before the change
	FIX	Whether or not a change is a vulnerability fix	Changes that fix a defect are more likely to introduce new defects [7].	Based on the dataset shared by [14]
	APC, MPC	Average/Maximum Past Changes made to the current modified files	VCCs are statistically significantly touched more times than ordinary changes [6].	Git rev-list subcommand
	APD, MPD	Average/Maximum Past different Developers for modified files	Files touched by more developers are more suspicious [4].	Git rev-list subcommand
Code	**NRC, NAC**	Number of Removed/Added Conditions	Missing or extraneous constraints may lead to vulnerabilities [19].	Regular expression
	NRFC, NAFC	Number of Removed/Added Function Calls	Eliminating or invoking some function calls may expose to vulnerabilities [19].	Regular expression
	NRF, NAF, NMF	Number of Removed/Added/Modified Functions	Missing, extraneous, or modified functions may make the code more secure or not [19].	Regular expression or by parsing the patch of the change
	NRVA, NAVA	Number of Removed/Added Variable Assignments	Variable assignments may attach or mistakenly drop some security constraints.	Regular expression
	KEYWORDS	68 C/C++ keywords like "if", "NULL"	Keywords like "if", "NULL" place more constraints that may obey or violate security policies.	Regular expression

in Fig. 3, VCCs tend to have larger C/C++ additions, slightly larger lines of source code, modestly more past changes and similar developer experience compared to unclassified changes. Interestingly, developers with less experience seems not contributing to more vulnerabilities and we ascribe it to that maybe experienced developers lack security expertise as well as new committers. All other change measures manifest a similar tendency like CA or APC except for NRF and ND.

Considering that the distribution of VCCs and unclassified changes are unknown, we apply the non-parametric hypothesis test – Mann-Witney-Wilcoxon (MWW) test [20] to examine whether a code change measure distributes differently in the aspect of statistics. It is reliable for validating with the null hypothesis that two samples come from the same population, while the alternative hypothesis is that one of the population tends to have larger value than the other.

MWW test is widely used in precedent vulnerability prediction models [5], [6]. We adopt Cohen's D [21] statistic to ascertain the strength of MWW test. For brevity's sake, we omit to show the test results as most of them are statistically significant except for NRF, ND, and EXP.

Although NRF, ND, and EXP are distributed similarly between VCCs and unclassified changes, it doesn't mean that they might not be good indicators in a machine learning based prediction model. We will decide whether to choose them in the step of feature selection.

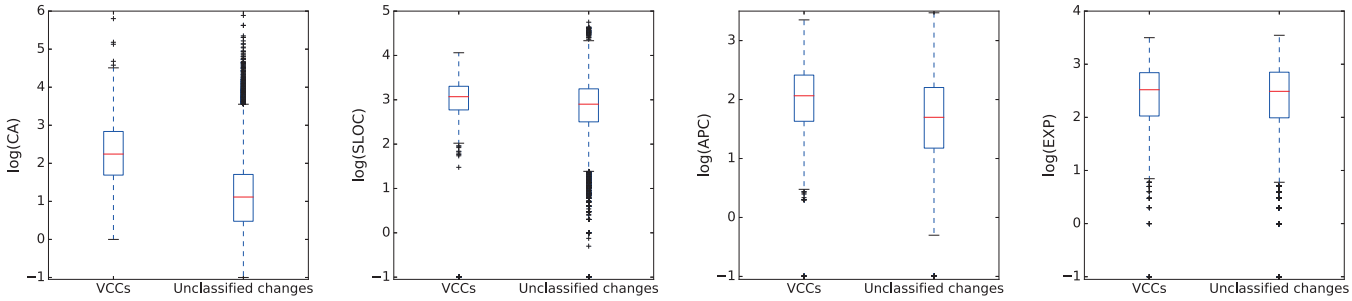
E. Prediction Techniques

The aforesaid code change measures provide some hints for the characteristics of suspicious code changes. We thereby feed them as features to two machine learning based models. One of them is a classification model to distinguish VCC from unflagged changes and the other is a regression model served as a measurement of how much effort could be saved in the process of security inspection.

1) Classification: Before using these extracted features in the model, we need to preprocess them to remove highly correlated factors and make a transformation to those highly skewed data. With the aid of Spearman's rank correlation coefficient [22], we found that CA and TA, CD and TD, NF and Entropy, and NAC and Keyword "If" are highly correlated. Thereby, we exclude one of them and then apply greedy forward variable selection. To deal with highly skewed data, we employ the MaxAbsScaler preprocessing from the tool scikit-learn [23].

Approach. Based on previous work, we choose a Random Forests technique to predict vulnerability-contributing changes as it outperforms other techniques like Naive Bayes, Logistic Regression on our dataset when we set the number of trees as 100. Random Forests is an ensemble learning method for classification and regression that fits a multitude of decision trees classifiers on diversified sub-samples from the dataset and utilizes averaging to avoid over-fitting and improves the accuracy. Besides, ensemble learning methods are known to cope well with highly imbalanced data like ours, where VCCs only represent a tiny percentage of all code changes (i.e., $1203/178515 = 0.67\%$).

2) Regression: Considering that VCCs tend to be larger changes (i.e., multiple additions and deletions of code) [6], it still requires considerable human effort to locate the vulnerability once a huge code change labeled as suspicious.



(a) CA (b) SLOC (c) APC (d) EXP

Fig. 3. Comparison of metric values for VCCs and unclassified changes.

Therefore, similar to prior work [24], we introduce an effort-aware model to assess how many VCCs could be caught with limited resources for security testing compared to a random predictor, the latter is an average model where the amount of VCCs could be found with the same proportion of effort.

Approach. Assume that only 20% of effort is available for reviewing the suspect changes flagged by our prediction model. Then we determine how many changes are VCCs from the most risky changes with fewer lines to inspect. In this model, the dependent variable $D(x)$ is defined as follows:

$$D(x) = \frac{Y(x)}{Effort(x)} \tag{1}$$

Here if the change x is predicted as a VCC, then $Y(x)$ is 1 and 0 otherwise. $Effort(x)$ is measured by the total lines modified (i.e., CA + CD) in C/C++ files from a change x. As for independent variables, similar code change features in the classification model are adopted except for TA, TD, CA, and CD since CA and CD together make up the effort in the dependent variable and TA and TD is highly correlated with CA and CD respectively.

Then we use a Random Forests regression model to predict $D(x)$ and prioritize the list of code changes to review by ranking the value of $D(x)$ in descending order. With 20% effort off the whole list, we calculate the number of VCCs contained in these most suspicious changes and compare the result with the baseline – the random predictor. In next part we discuss the results as well as the evaluation of our models.

IV. Results and Evaluation

We present our results of experiments and evaluate the performance of our classification and regression models in various ways. We elaborate the performance indicators used for our approach initially, then the validation techniques would be detailed along with illustrated results. Finally we make comparisons to existing tools.

A. Performance Indicators

For the binary classification problem, we measure the performance using precision and recall as they have been broadly used in the literature of vulnerability prediction [2], [25], [6]. Additionally, to assess the effort of reviewing suspect code changes, we introduce file inspection ratio and line inspection ratio criteria like [25]. For the regression problem, we adopt **cumulative lift chart** to examine the percentage of total VCCs that could be found with limited effort.

Before explaining these indicators in detail, we need to elucidate that there are four types of outcomes in a binary classification problem. Correct results are called *True Positives* (TP, vulnerable changes correctly identified as vulnerable) and *True Negatives* (TN, unclassified changes correctly predicted as unclassified). There are also two kinds of errors: *False Positives* (FP) and *False Negatives* (FN). A FP happens when an unclassified change is incorrectly identified as vulnerable and FN denotes the number of vulnerable changes incorrectly identified as unclassified.

Precision: The precision measures the percentage of predicted VCCs that are correctly classified as vulnerable. Higher precision is desirable as fewer false alarms would generate, hence, less time is wasted on scrutinizing changes that are actually clean. It is defined in Eq (2):

$$Precision = TP/(TP + FP) \tag{2}$$

Recall: We also concern about the recall that represents the percentage of actual VCCs correctly identified as vulnerable. Higher recall is preferable as it means more VCCs could be found. Recall is defined in Eq (3):

$$Recall = TP/(TP + FN) \tag{3}$$

File Inspection Ratio (FIR): Some code changes contain modifications in various files and inevitably complicated to understand. Besides, it would be ineffective if reviewers have to inspect too many files only to find a few vulnerabilities. We interpret the number of modified C/C++ files of a change that is a TP as TP_{NF}. FP_{NF}, TN_{NF}, and FN_{NF} are similarly defined. Thence we define FIR as the percentage of files to be inspected based on the results of prediction in Eq (4):

$$FIR = \frac{TP_{NF} + FP_{NF}}{TP_{NF} + FP_{NF} + TN_{NF} + FN_{NF}} \tag{4}$$

Lines Inspection Ratio (LIR): Similarly with files, a larger code change may require more effort than changes with fewer lines modified. Accordingly we additionally measure LIR (i.e., the percentage of lines of source code to be inspected in terms of predicted results). We define TP_{LOC} as the total lines of C/C++ source code modified in a change that is a TP, similarly

with FP_{LOC}, TN_{LOC}, and FN_{LOC}. And LIR is calculated in Eq (5):

$$LIR = \frac{TP_{LOC} + FP_{LOC}}{TP_{LOC} + FP_{LOC} + TN_{LOC} + FN_{LOC}} \quad (5)$$

B. Experimental Results

When applying machine learning techniques, we perform a temporal time split between the training and test data instead of classic cross validation. As it is an instance of "future prediction" and generally considered better than cross validation as the former simulates a realistic scenario usage.

Since the BugID-vulnerability data shared by [14] only contain verified information as of the year of 2014, we collect code changes of Firefox as of 2014 and only focus on those with C/C++ files modified. We pick a specific time to contain two thirds of all VCCs as training data. And a short description of training and test data is listed in Table II.

TABLE II
DISTRIBUTION OF CHANGES AND VCCS.

	Dataset		
	Training	Test	Total
VCCs	802	401	1,203
Unclassified changes	117,120	60,192	177,312

For the classification problem, we argue that the precision and LIR are the most important metrics as these two features determine how much effort of reviewing the code could be saved for security researchers. Fig. 4 shows the detection performance with different feature sets. As can be seen, the classifier that combined code and metadata metrics outperforms the classifiers which only operate on code or metadata features, in both measurements. Moreover, metadata features are slight better than code features in terms of precision while more lines of code needs inspection. We will further make comparison to existing tools in the next part thus omitted here.

While for the regression problem, as shown in Fig. 5, there is a nearly 10 percent gap between our effort-aware model and random predictor when no more than 80% effort is available. We observe that the performance of our tool is slightly poorer than that of random predictor when effort > 80%, but this condition is considered to be impractical since 80% effort would require a quite huge amount of lines to inspect thus omitted here from a realistic point of view.

In terms of only 20% effort available for reviewing suspicious code changes flagged by VulDigger, we find that VulDigger could successfully identify 123 VCCs (i.e., 31% of all VCCs in the test data) compared to 20% of all VCCs found by the random predictor in an ideal setting, which is a 55% improvement and we prove that such tool is effective in locating vulnerabilities in the process of security testing. By comparison, [7] detected less than 20% defect-inducing changes when 20% effort was spent on the Mozilla project, i.e., their model is even worse than a random predictor.

To answer another interesting problem that how many percentage of effort provides the best return on investment,

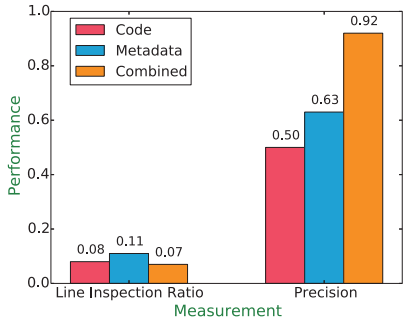


Fig. 4. Detection performance of VulDigger using different feature sets.

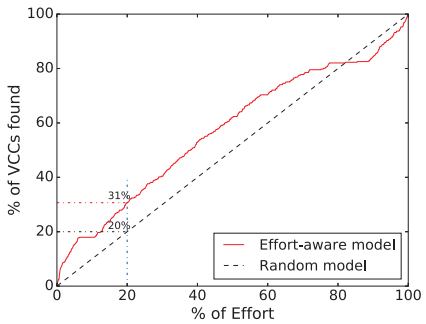


Fig. 5. Effort-aware cumulative lift chart.

we use the number of detected VCCs divided by the required effort as a measurement. This measurement is reflected as the slope of the curve in Fig. 5 and we can observe that 6% effort provides the best return on investment.

C. Comparison to FlawFinder

Similar to [6], we additionally compare our findings against Flawfinder [8] version 1.31. Flawfinder is a mature static C/C++ source code scanner and marks lines in a source code file with potential vulnerabilities. To compare the performance, we run Flawfinder on each modified or added file of a code change and mark a change as vulnerable if Flawfinder reports at least a flaw in one of lines the change inserted.

We then evaluate the classification performance between our tool and Flawfinder against the test dataset. Table III shows the contingency table as well as precision, recall, FIR, and LIR for both tools. As stated before, the precision and LIR are two most important metrics in this table. While a high recall is theoretically meaning that more vulnerabilities would be found, in reality they would be buried in a lot of false positives [6]. So we accept that we will not find all of VCCs but provide a more realistic solution for security testing.

As shown in Table III, our tool VulDigger significantly outperforms Flawfinder in most configuration settings. The most practical result comes with a remarkably high precision (i.e., 92%) and an extremely low LIR (i.e., 7%), which means by reviewing only 60 of 60,593 changes, we can locate 55 VCCs with only 5 false alarms. As for the same precision, VulDigger indeed locates more than twice VCCs while more files and lines have to be reviewed. In terms of the same recall, VulDigger presents a precision of 18% compared to Flawfinder's precision of 3% and $27\% = (0.44 - 0.32)/0.44$ effort can be saved with our tool. We also examine when

TABLE III
COMPARISON OF FLAWFINDER AND VULDIGGER. MEASUREMENTS IN THE FIRST ROW IS DEFINED IN SECTION IV-A.

	True Positive	False Positive	False Negative	True Negative	Precision	Recall	FIR	LIR
Flawfinder	89	2,495	312	57,697	0.03	0.22	0.13	0.44
VulDigger								
- Most practical result	55	5	346	60,187	**0.92**	0.14	0.02	**0.07**
- With same precision	197	6,255	204	53,937	0.03	0.49	0.28	0.62
- With same recall	88	393	313	59,799	0.18	0.22	0.07	0.32
- With same LIR	129	1,714	272	58,478	0.07	0.32	0.13	0.45

the same effort (i.e., LIR) is applied, VulDigger can locate $45\% = (0.32 - 0.22)/0.22$ more VCCs and $133.3\% = (0.07 - 0.03)/0.03$ more accuracy with the comparison of Flawfinder. In brief, VulDigger can be applied in a more realistic situation than Flawfinder.

We intend to compare our tool with the state-of-the-art approach [6] on the same dataset, however, their data is temporarily unavailable and we may remain it as future work.

V. LIMITATIONS AND THREATS TO VALIDITY

Internal validity. One may notice in VCCs that only acknowledged vulnerabilities with linked fixing changes can be mapped to corresponding VCCs. Whereas, 1203 VCCs are large enough (compared to 640 VCCs in [6]) for training the classifier and wrongly labeled VCCs are eliminated by manual inspection. Besides, some code changes are refactoring or repeating previous modifications, thus may compromise the precision of our tool (as shown in Table III). Further studies on identifing these changes might further improve our predictions.

External validity. We focus on C/C++ code changes as we want to ensure comparability between the features like keywords. More studies are necessary to generalize our results to other types of applications and programming languages.

ACKNOWLEDGMENT

The work was supported by the National Natural Science Foundation of China (Grant Nos. 61472249, 61572192, 61571191) and International Science & Technology Cooperation & Exchange Projects of Shaanxi Province (2016KW-038).

REFERENCES

[1] "The heartbleed bug," [Accessed 20-March-2017]. [Online]. Available: http://heartbleed.com/

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013.

[8] "Flawfinder homepage," [Accessed 20-March-2017]. [Online]. Available: http://www.dwheeler.com/flawfinder/

[9] 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.

[10] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates, "When a patch goes bad: Exploring the properties of vulnerability-contributing commits," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 65–74.

[11] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, "Identifying the characteristics of vulnerable code changes: An empirical study," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 257–268.

[12] "Mozilla foundations security advisories," [Accessed 31-March-2017]. [Online]. Available: https://www.mozilla.org/en-US/security/advisories/

[13] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *ACM sigsoft software engineering notes*, vol. 30, no. 4. ACM, 2005, pp. 1–5.

[14] F. Massacci, S. Neuhaus, and V. H. Nguyen, "After-life vulnerabilities: a study on firefox evolution, its vulnerabilities, and fixes," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2011, pp. 195–208.

[15] E. Giger, M. Pinzger, and H. C. Gall, "Comparing fine-grained source code changes and code churn for bug prediction," in *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 2011, pp. 83–92.

[16] "Github api v3 | github developer guide," [Accessed 12-March-2017]. [Online]. Available: https://developer.github.com/v3/

[17] A. G. Koru, D. Zhang, K. El Emam, and H. Liu, "An investigation into the functional form of the size-defect relationship for software modules," *IEEE Transactions on Software Engineering*, vol. 35, no. 2, pp. 293–304, 2009.

[18] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[19] M. Piancó, B. Fonseca, and N. Antunes, "Code change history and software vulnerabilities," in *Dependable Systems and Networks Workshop, 2016 46th Annual IEEE/IFIP International Conference on*. IEEE, 2016, pp. 6–9.

[20] M. P. Fay and M. A. Proschan, "Wilcoxon-mann-whitney or t-test? on assumptions for hypothesis tests and multiple interpretations of decision rules," *Statistics surveys*, vol. 4, p. 1, 2010.

[21] R. E. McGrath and G. J. Meyer, "When effect sizes disagree: the case of r and d." *Psychological methods*, vol. 11, no. 4, p. 386, 2006.

[22] J. L. Myers, A. Well, and R. F. Lorch, *Research design and statistical analysis*. Routledge, 2010.

[23] "scikit-learn: Machine learning in python," [Accessed 30-March-2017]. [Online]. Available: http://scikit-learn.org/stable/

[24] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*. IEEE, 2010, pp. 107–116.

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