2015 12th Working Conference on Mining Software Repositories

Do Bugs Foreshadow Vulnerabilities? A Study of the Chromium Project

Felivel Camilo, Andrew Meneely, and Meiyappan Nagappan
Department of Software Engineering
Rochester Institute of Technology,
134 Lomb Memorial Drive
Rochester, NY, USA 14623
1+585-475-7829
{fdc7162,axmvse,mxnvse}@rit.edu

***Abstract*—As developers face ever-increasing pressure to engineer secure software, researchers are building an understanding of security-sensitive bugs (i.e. vulnerabilities). Research into mining software repositories has greatly increased our understanding of software quality via empirical study of bugs. However, conceptually vulnerabilities are different from bugs: they represent abusive functionality as opposed to wrong or insufficient functionality commonly associated with traditional, non-security bugs. In this study, we performed an in-depth analysis of the Chromium project to empirically examine the relationship between bugs and vulnerabilities. We mined 374,686 bugs and 703 post-release vulnerabilities over five Chromium releases that span six years of development. Using logistic regression analysis, we examined how various categories of pre-release bugs (e.g. stability, compatibility, etc.) are associated with post-release vulnerabilities. While we found statistically significant correlations between pre-release bugs and post-release vulnerabilities, we also found the association to be weak. Number of features, SLOC, and number of pre-release security bugs are, in general, more closely associated with post-release vulnerabilities than any of our non-security bug categories. In a separate analysis, we found that the files with highest defect density did not intersect with the files of highest vulnerability density. These results indicate that bugs and vulnerabilities are empirically dissimilar groups, warranting the need for more research targeting vulnerabilities specifically.**

I. Introduction

Developers are facing an ever-increasing pressure to engineer secure software. A simple coding mistake or design flaw can lead to an exploitable vulnerability if discovered by the wrong people. These vulnerabilities, while rare, can have catastrophic and irreversible impact on our increasingly digital lives. Vulnerabilities as recent as Shellshock and Heartbleed are reminders that small mistakes can lead to widespread problems. To engineer secure software, developers need a scientifically rigorous understanding of how to detect and prevent vulnerabilities.

We can build an understanding of vulnerabilities by viewing them as security-sensitive bugs. That is, a vulnerability can be defined as a "software defect that violates an [implicit or explicit] security policy" [1]. Research into mining software repositories has greatly increased our understanding of software quality via empirical study of bugs. Researchers have provided a myriad of metrics, prediction models, hypothesis tests, and other actionable empirical insight that speak to the nature of bugs [2]–[4]. At first glance, research on software quality should translate to software security.

However, vulnerabilities are conceptually different than traditional bugs. Vulnerabilities represent an abuse of functionality as opposed to wrong or insufficient functionality commonly associated with non-security bugs. Vulnerabilities are about allowing "too much" functionality so as to be open to attack. For example, an open permissions policy may function perfectly well for most users, but would be quickly exploited by attackers. Or, a simple memory leak can be coerced into denial-of-service attack. As a result, vulnerabilities are about what the system is supposed to *prevent* from happening beyond the functionality that the customer requires.

Thus, the relationship between bugs and vulnerabilities deserves empirical examination. In particular, can software quality problems foreshadow security problems? If the correlation between bugs and vulnerabilities is strong, then empirical analyses should focus primarily the super-group of bugs. If not, perhaps some subgroups of bugs (e.g. stability bugs) may foreshadow vulnerability problems in the future.

The objective of this research is *improve our fundamental understanding of vulnerabilities by empirically evaluating the connections between bugs and vulnerabilities*. We conducted an in-depth analysis of the Chromium open source project (a.k.a Google Chrome). We collected code reviews, post-release vulnerabilities, version control data, and bug data over six years of the project. We conducted regression analysis to evaluate the strength of association, along with examining subgroups and various rankings of the files. We repeated our analysis across five annual releases to gauge the sensitivity of the results. We focused on the following research questions (and had the following results):

RQ1. Are source code files fixed for bugs likely to be fixed for future vulnerabilities? (We found that files with more pre-release bugs are slightly more likely to present post-release vulnerabilities.)

RQ2. Are some types of bugs more closely related to vulnerabilities than others? (Here we discovered that while of some types of pre-release bugs present a stronger association than others to post-release vulnerabilities, this relation is overall weak.)

978-0-7695-5594-2/15 $31.00 © 2015 IEEE
DOI 10.1109/MSR.2015.32

RQ3. Do the source code files with the most bugs also have the most vulnerabilities? (We found that files with the top count of pre-release bugs have only some post-release vulnerabilities.)

The rest of the paper is organized as follows. Sections II and III cover the terminology and related work introducing the concepts to understand of the paper. Section IV we present our research questions. In Section V we introduce and explain the Chromium browser. Section VI explains our analysis methodology approach, answers the research questions and discusses the results. Section VII we discuss the threats to validity and we finally conclude in Section VIII with a brief summary.

II. Terminology

We use various technical and statistical terms with respect to processing and analyzing the data. We define these terms first ensuring that the essence of the paper can be understood by a broader audience.

A. Data terms

When we refer to a **release**, we are referring to a milestone in the software development life cycle for our case study project. Releases represent an snapshot of all source code files at a specific point in time. The project evolves from release to release through small changes called **commits**. The commits represent unique changes to the source code of the project, and are recorded by a version control system (e.g. Git). We also make the special distinction between non-security related software flaws (**bugs**), that manifest as the lack of expected functionality. Security related software flaws (**vulnerabilities**) manifest as violation of the system's security policies [1]. We categorize files of a release as **vulnerable** if they were fixed as a result of a vulnerability post-release, and other files are **neutral** (i.e. no vulnerabilities were known to be associated with them).

We also use the metric Source lines of code (**SLOC**) as a measurement of file size. In this study source comments or whitespace are not considered SLOC. SLOC forms part of a group called traditional software metrics [5] (Other metrics in this group are Code Churn and ciclomatic complexity).

B. Statistics terms

Spearman correlation coefficient: This is an statistical test for measuring the association between two or more variables. This metric has the ability to rank the strength of the correlation between variables. The following scale is used to measure and interpret the result values [6]: *a*) ± 0.00 - 0.30 "Negligible" *b*) ± 0.30 - 0.50 "Low correlation" *c*) ± 0.50 - 0.70 "Moderate correlation" *d*) ± 0.70 - 0.90 "High correlation" *e*) and ± 0.90 - 1.00 "very High correlation".

Mann-Whitney-Wilcoxon test (MWW): This test evaluates the difference in values of two populations, and reveals if the two populations are statistically equal when compared against a null hypothesis. Previous software metrics studies [4], [7] have suggested this as suitable test for validating software metrics.

Cohen's D Statistic: The Cohen's D is an effect size measure that evaluates the strength of a phenomenon [8], [9]. While MWW measures statistical significance, it does not measures the strength of the association. In this study we make use of the Cohen's D to support the results of the MWW test and measure how closely related the vulnerable population is to neutral population. Cohen's D is defined as:

$$\delta = \frac{\text{mean difference}}{\text{standard deviation}}$$

The original paper measures in terms of non-overlap percentage, We use the inverse of that measure [10] in this paper , where a lower Cohen's D indicates that a larger overlap exists. The following scale was proposed originally by the authors [11] to interpret the results in terms of non-overlap: *a*) 0.20 "Small" *b*) 0.50 "Medium" *c*) or 0.80 "Large".

Logistic Regression: Logistic regression analysis predicts the possibility of a binary outcome based on a set of indicator variables. The goal of a logistic regression is to find the equation (combination of the right indicator variables) that best predicts the probability of a true or false outcome [2], [12].

Akaike information criterion (AIC): This is a measure of relative model fitness calculated on generalized models. This metric can be used to rate a group of models and find the best one among them. By itself the AIC is not a measure of how fitted the model is, but instead it is relative rank, suitable for comparing with other models that were trained using the same data set. A lower the AIC value, means a better fit model [2], [13], [14].

Percent of deviance explained (D^2): This measure indicates how well the data fits a statistical model. D-squared is the equivalent measure of R-squared for linear models [2], [15], except that this metric is employed to evaluate the amount of deviance that the model accounts for. This measure can also be used to evaluate the goodness of fit of a group of generalized linear models. A higher D^2 means a better fit model.

Precision and recall: When we use logistic regression to predict the outcome of a binary variable, the model can make two types of mistakes: False Positives (FP) and False Negatives (FN). False positives are when the model identified neutral file as vulnerable, and false negatives are when the model identifies a vulnerable file as neutral. The relevant values are defined as True Positives (TP) and True Negatives (FN), when the model accurately identifies a vulnerable or neutral file.

Precision is defined as the fraction of all the predictions that are relevant , that is $precision = \frac{TP}{(TP+FP)}$. Recall is defined as the fraction of all relevant instances that are retrieved, that is $recall = \frac{TP}{(TP+FN)}$. To serve as example and to explain the application of this metric in our study we present the following scenario: Imagine a data set of 50 files, from which 30 files are known to be "vulnerable". Then we use a model to retrieve 30 files, if the model is able to identify 10 vulnerable files correctly but misidentifies on 20 files, here $precision = \frac{10}{20}$ and $recall = \frac{10}{30}$.

F-measure: This is a metric that can be interpreted the as a weighted average of precision and recall. It is considered a more complete metric that using only precision and recall [16] because it takes into account both values. The F-measure is defined as:

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Area Under the ROC Curve (AUC): Receiver operating characteristic curves can be used to graphically represent the performance of a logistic model by plotting the precision and recall. The area under the ROC curve measures the ability of the model to correctly classify the binary outcome. To understand this concept, consider the following example: We have separated the files in vulnerable and neutral; after that we randomly select two files one from the vulnerable group and one from the neutral group and then use a model to predict the outcome (Vulnerable or Neutral); the AUC represents the percentage of observations in which the model will be able to discriminate correctly between two files.

III. Related Work

Other researchers have studied the relationship between bugs and vulnerabilities, and this work extends those questions with further analysis and a larger data set. Gegick, M. et al. [3], [17] studied the prediction of attack prone components, by identifying and ranking components that are more likely to present vulnerabilities. These rankings were proposed to prioritize the security risk management efforts. In their study they used automated static analysis tools to calculate different metrics and show how these metrics relate with a higher vulnerability risk. This study directly relates with our research, as it tries to evaluate vulnerability-proneness in files based on non-security factors (i.e. code churn, SLOC, and previous manual inspections), we go a step further by taking account multiple releases as factor.

In a prior study by Meneely, A. et al. [4], the authors evaluated code review effectiveness and the Linus laws applying in terms of vulnerabilities. Investigating if "many eyes make all the bugs shallow", also apply to vulnerabilities. The results from this previous study show that contrasted with traditional bugs, vulnerabilities are still missed by many reviewers, indicating an intrinsic difference in the two groups, that we aim to clarify. We have expanded the data set from this previous study and have included four additional releases, introduced the bug report data and recollected the commit logs, code reviews and vulnerability entries.

Studies have shown the potential use of logistic regression analysis as a way to predict future software flaws, based on traditional software metrics [5] and defects. A previous study by T.-H Chen et al. [2] shows the use of logistic regression analysis using various static and historical defects metrics to explain future software defects. This study takes into account the defect history of each topic and evaluate the probability of future defects. We borrow the historical analysis from their approach by apply it to the evaluation of future vulnerability. Also Shihab et al. [18] evaluated different software metrics in look for patterns that lead to high impact defects. These defects break the functionality of production software systems and cause a negative impact on customer satisfaction. This study also makes use of logistic regression as the means of predicting a binary outcome. Their findings indicate that while logistic regression analysis can be used to predict high impact defects, there is a need of further development to bring these predictions techniques closer to an industry adoption.

Other studies have evaluated the occurrence security-related bugs through the analysis of source control repositories and static analysis tools. Mitropoulos, D. et al. [19] studied the relation of software bugs compared to other bugs categories, using maven repositories. Their results indicate security bugs do not have a recognizable pattern in the projects they studied, and encourage the further investigation on this topic. Mitropoulos, D. et al. [20] also studied the evolution of security related bugs by tracking down their introduction. Their results show that the number of security-related software bugs increases as the system evolves and it is influenced by external software dependencies. These two studies use static analysis tools to identify bugs, in contrast We use data from a bug tracking system and bug labels, eliminating bias introduced by the tool.

A recent study by Tantithamthavorn, C. et al. [21] evaluates the impact of mislabeling in the interpretation and performance of prediction models. Their results show that bug labeling mechanisms are a reliable source of bug classification and model training. Their findings increase the relevance of our research as we make use of Google Code labels to classify the pre-release bugs.

IV. Research Questions

We approach our analysis by first conducting an overall analysis with a single variable, then we conducted our analysis by using the categorizations provided by the Chromium project. Finally, we examined how vulnerabilities are spread across files when ranked by defect density. Thus, our three research questions are:

- RQ1. Are source code files fixed for bugs likely to be fixed for future vulnerabilities?
- RQ2. Are some types of bugs more closely related to vulnerabilities than others?
- RQ3. Do the source code files with the most bugs also have the most vulnerabilities?

V. Data: The Chromium Project

The Chromium Browser is the open source project behind Googles Chrome Browser. The Chromium project is a large project consisting of more than 4 million Source Lines of Code (SLOC). We selected this project because its high visibility and popularity in the open source developer community and more significantly because it offers different analysis opportunities and perspectives that can be explored using software repository mining techniques [22]. Our in-depth analysis to produce this data involved analyzing code reviews, investigating vulnerability claims, and maintaining traceability between the artifacts.

A. Chromium Vulnerabilities (NVD, Public Disclosures)

The Chromium team regularly acknowledges vulnerabilities that have been fixed in each new release. These post-release vulnerabilities are recorded in the National Vulnerability Database (NVD) and given a Common Vulnerabilities and Exposures (CVE) identifier. However, not every CVE entry is accurate. Thus, we conducted manual analysis of each vulnerability to ensure that it was acknowledged by Chromium, fixed, and the fix was released. These vulnerabilities are post-release vulnerabilities that are reported to the Chromium Team, and present potential threats users. In this study, we have traceability from 703 CVE entries to the code review and the commit that fixed the vulnerabilities.

A key part of mining the vulnerability data were tracing them to code reviews and git logs. We mined 242,635 Git [23] commits, spanning across six years of development of the Chromium project. These commits contain information about the source files that were change. Additionally the commits contain the ID of the bug report that this change contributes to fix and the ID of the code review that inspected the change before adding it to the main code base. This IDs are used to link the records in a relational database. The commit data is the central point of our data set, combining together the bugs and the vulnerabilities through code reviews.

B. Chromium Bug Tracking System (Google Code)

Chromium uses Google Code as bug tracking system [24], [25]. We have collected 374,686 bug entries with 5,825 different labels, and 3,801,444 bug comments. A bug entry does not necessarily mean an actual flaw in the system: system features, and other tasks are in this database as well. The Google Code labels function as a taxonomy system, and they allow the developers to label specific keywords to the bugs reports, including the "bug" label. The system then uses these labels to categorize, filter and search the bug repository. Special labels are used to attach additional critical information like priority (Pri), product category (Cr), operating system where the bug occurs (OS), possible milestone to release a fix (M-value) and type of bug. Additionally Google Code users can manually add arbitrary labels to the bug reports. To mitigate misspellings of labels, we manually inspected the label choices throughout the 5,825 labels that were used.

We aggregated these labels and examined how often they were used. Among the 5,825 labels, only a few were labeled consistently. For our RQ2 question about categories, we examined labels that were used over 1,000 times and identified categories that were not specific to Chromium (e.g. "stability" is specific to Chromium, but "Milestone18 Migration" is not).

C. Processing By Releases

To account for the evolution of the project, we separated the data into **releases**. Chromium utilizes a rapid release cycle, with a major release coming out approximately every two months. In prior work [26] we found that vulnerabilities on average have remained in the system for approximately two years prior to their fix. Using those results as a guide, a vulnerability fixed within one year of a release may have reasonably existed in the system for up to one year prior a release. Additionally, we conducted our analysis in time segments that are non-overlapping to avoid double counting and ultimately auto-correlation. Thus, we chose five major releases that were approximately one year apart: versions 5.0, 11.0, 19.0, 27.0, and 35.0.

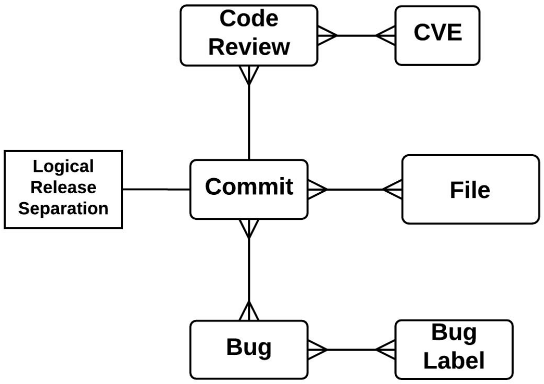

Fig. 2. Conceptual diagram of data set

Throughout this paper, we use the phrase "pre-release bugs". In the context of Chromium's rapid release cycles, this is not an entirely apt name. Many bugs are exposed to users by some major release since the product is constantly being release. In Chromium, we did not see any consistent labeling referring to pre-release vs. post-release bugs. Our concept of "pre-release bugs" comes from our own selection of five releases, not necessarily from bugs that were never part of production release.

We used this pre-release bug data and post-release vulnerability data so that no overlap exists between the groups. As shown in Figure 1, bugs between 5.0 and 11.0 are used exactly once: in the Release 11.0 analysis. The only overlap is in our analysis of pre-release "security bugs" and post-release vulnerabilities: these groups are not entirely the same as some vulnerabilities are not recorded in the bug-tracking system and some security-labeled bug entries were never part of a release to users.

The Git repository contained specific release information via the tag feature. We performed a manual investigation for each specific release dates to corroborate the dates, comparing the actual release of the product to the public and the date of the final version commit.

D. Computing Metrics

We used Ruby and the Ruby on Rails ActiveRecord libraries to parse, model, verify and query the data. We used R to conduct our analysis. We made use of a nightly build process to integrate and verify long running queries. The conceptual data diagram can be found in Figure 2.

We set up a process to calculate different file-level metrics in a release per release basis. We present the file-level metrics

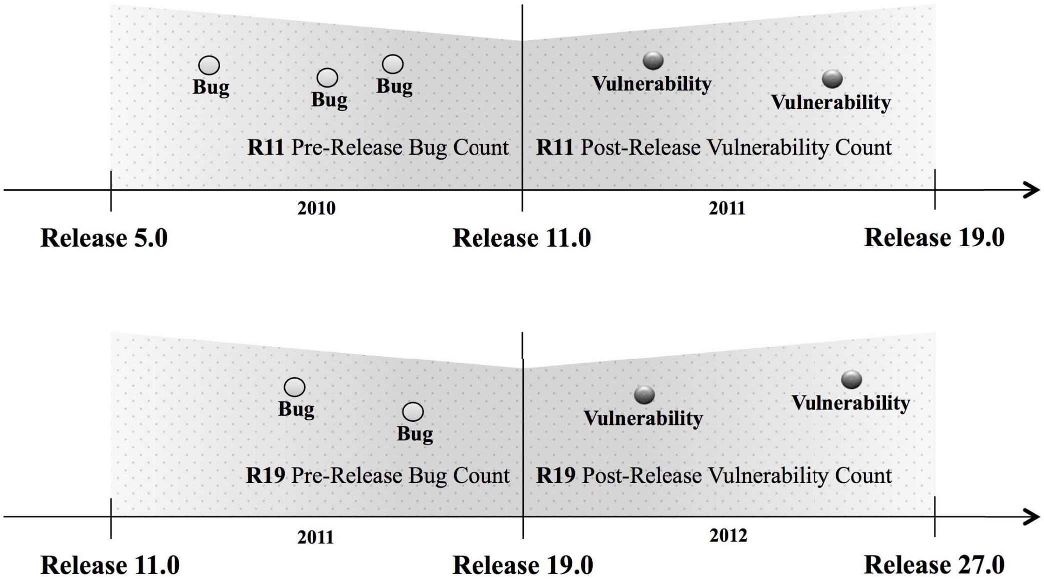

Fig. 1. Release time line explaining bug and vulnerability selection criteria. For Release 11: the population of possible pre-release bugs is equal to the bugs reports opened in (roughly) 2010. The population of possible post-release vulnerabilities is equal to the vulnerabilities reported in the next year.

TABLE I
SELECTED RELEASES BEFORE AND POST PROCESS. NUMBER OF FILES (VULNERABILITY PERCENTAGE)

Release	Date	Pre-Process	Post-Process
Chromium 5.0	Jan 2010	9,142 (2.55%)	2,276 (11.08%)
Chromium 11.0	Jan 2011	17,005 (3.35%)	4,760 (12.80%)
Chromium 19.0	Feb 2012	21,818 (1.40%)	6,826 (4.61%)
Chromium 27.0	Feb 2013	30,087 (0.98%)	8,451 (3.58%)
Chromium 35.0	Feb 2014	35,871 (0.28%)	8,125 (1.23%)

in Table II. For each file we count the number of pre-release bugs and the number of post-release vulnerabilities reported in a one year. We only use source code files, which are primarily C/C++ file extensions.

To measure the pre-release bug count we evaluated if the specific file was modified in a commit that linked to a bug report, if a match is found we add one to the bug count. Only bug reports opened from a year prior to the specific release date are taken into account. Additionally in this step we make use of the Chromium bug labels to categorize the bug count in different types of bugs. We created 7 bug categories based on the 7 most vulnerability relevant labels. These 7 labels were selected from the top most used labels.

To measure the post-release vulnerabilities we evaluated if the specific file was modified in a commit that linked to a code review that fixed a vulnerability. In comparison to the criteria for bugs this is a forward search, only selecting the vulnerabilities that were reported from the release date to a year after the release date, this information is selected as **num-post-release-vulnerabilities**. The boolean **becomes-vulnerable** is set to true if any vulnerabilities are found in that future date range.

A graphical representation of both selection processes can be found in Figure 1.

Finally, we removed files that were never fixed for any bugs nor vulnerabilities. Since no quality nor security data data exist for these files, no inference can be made about their quality or security. Table I shows the a summary of the state of the releases.

VI. ANALYSIS APPROACH & RESULTS

We began an in-depth exploratory statistical analysis using the metrics calculated at the file level. We used the pre-release bugs as independent variables and future vulnerabilities as dependent variable. In this section we explain in detail the analysis methodology and show the experimental results.

Our first goal was to evaluate the quality of our independent variables. The validity of our future results depends on our ability to distinguish between these the pre-release bug types. Thus, having two or more variables with heavy correlation will increase the comparison difficulty of the next analysis steps.

We applied the Spearman correlation coefficient analysis on our 7 pre-release bug variables looking to remove variables that present high co-linearity. The results from the Spearman correlation coefficient indicate that the pre-release bug variables show a negligible to low correlation with each other. In Table III we present the maximum Spearman correlation coefficient of each variable when we run the test against all

TABLE II
File-level Metrics Computed Per Release

Metric	Description
num-pre-bugs	Number of bugs discovered from a year before the selected release.
num-pre-features	Number of bugs labeled as *"type-feature"* discovered from a year before the selected release.
num-pre-compatibility-bugs	Number of bugs labeled as *"type-compat"* discovered from a year before the selected release.
num-pre-regression-bugs	Number of bugs labeled as *"type-bug-regression"* discovered from a year before the selected release.
num-pre-security-bugs	Number of bugs labeled as *"type-bug-security"* discovered from a year before the selected release.
num-pre-tests-fails-bugs	Number of bugs labeled as *"cr-tests-fails"* discovered from a year before the selected release.
num-pre-stability-bug	Number of bugs labeled as *"stability-crash"* discovered from a year before the selected release.
num-pre-build-bugs	Number of bugs labeled as *"build"* discovered from a year before the selected release.
num-post-vulnerabilities	Number of vulnerabilities discovered from a year after the selected release.
becomes-vulnerable	Denotes if a module will become vulnerable in the future year.

TABLE III
Maximum Spearman Rank Correlation Coefficient Between Pre-Release Bug Variables Per Release

Metric	R5	R11	R19	R27	R35
num-pre-features	-0.12	-0.18	-0.23	-0.29	-0.32
num-pre-compatibility-bugs	0.15	0.08	0.08	0.05	0.03
num-pre-regression-bugs	0.19	0.29	0.19	0.13	0.32
num-pre-security-bugs	0.15	0.18	0.15	0.11	0.12
num-pre-tests-fails-bugs	0.05	0.09	0.06	0.04	0.03
num-pre-stability-bugs	0.19	0.29	0.19	0.13	0.13
num-pre-build-bugs	-0.12	0.18	0.23	0.29	0.13

other pre-release variables. Even the highest values indicate low correlations, so we did not see co-linearity that would interfere with multiple regression.

RQ1. Are source code files fixed for bugs likely to be fixed for future vulnerabilities?

Motivation. Our goal here is to evaluate, in a broad sense, if files that have been frequently fixed for bugs also have a higher probability of being fixed for a vulnerability. This research question serves as an overall measure of quality and security, but does not delve into comparing other factors such as features and security-related bugs (as RQ2 does).

TABLE IV
MWW Test results for SLOC and num-pre-bugs per Release. All Result Were Statistically Significant ($p - value < 0.05$).

	Metric	Median Vuln.	Median Neutral
Release 5.0	SLOC	5.30	4.63
	num-pre-bugs	1.95	1.39
Release 11.0	SLOC	5.27	4.26
	num-pre-bugs	1.61	1.10
Release 19.0	SLOC	5.31	4.42
	num-pre-bugs	1.61	1.39
Release 27.0	SLOC	5.43	4.67
	num-pre-bugs	1.79	1.39
Release 35.0	SLOC	5.73	4.76
	num-pre-bugs	1.94	1.61

TABLE V
Cohen's D Effect Size Statistic for num-pre-bugs in vulnerable and neutral populations per Release.

	Cohen's D	Cohen's U3	% Overlap
Release 5.0	0.74	76.92%	71.28%
Release 11.0	0.57	71.55%	77.58%
Release 19.0	0.29	61.23%	88.66%
Release 27.0	0.36	64.24%	85.52%
Release 35.0	0.39	65.02%	84.70%

Analysis. To examine how the pre-release bugs are related to an increased chance of future vulnerability, we used the non-parametric **Mann-Whitney-Wilcoxon test (MWW)** to examine if a statistically significant correlation exists. We tested the number of pre-release bugs, separating the population in vulnerable and neutral files, and evaluated the becomes vulnerable hypothesis. To examine the amount of overlap between neutral and vulnerable files with respect to number of bugs, We used the Cohen's D [8] statistic.

The MWW results in Table IV show that in comparison vulnerable files have a larger median pre-release bug count than neutral files; this difference is small but constant for all of our releases. All of our MWW test results were statistically significant ($p < 0.05$). We also included in this analysis the SLOC population to reconfirm finding from related and prior work that vulnerable files tend to be larger in size than neutral files.

The Cohen's D statistic is useful for gauging the strength of our MWW results by evaluating the amount of overlap between the vulnerable and neutral populations. This statistic is computed by comparing the two population means and factoring in how many of each population are above the other population's mean. Our results for Cohen's D, presented in Table V, show that the lowest amount of overlap between vulnerable and the neutral populations with respect to bugs is 71.28% on Release 5.0. As the project matures, the overall overlap stays above 75%. The Cohen's D literature indicates that this overlap is considered to be medium-to-large, indicating that the association is relatively weak. That is, many neutral files have many bugs and many vulnerable files have few bugs.

These results indicate that, broadly speaking, files with a history of bugs are likely to be fixed for vulnerabilities in the future. However, this association has many counterexamples, leading to a weak association overall.

RQ2. Are some types of bugs more closely related to vulnerabilities than others?

Motivation. From RQ1, we learned that, in aggregate, a connection between bugs and vulnerabilities exists in Chromium across multiple releases. However, bugs come in many different forms. Some bugs can be related to maintaining compatibility across operating systems, other might be related to stability problems that could foreshadow future vulnerabilities. For example, code with a history bugs related to 32-bit and 64-bit builds might have integer overflow problems that become exploitable.

Furthermore, we use this sub-category analysis to gauge the strength of the connection between bugs and vulnerabilities. In particular, we use SLOC, features, and security-related bugs as bases of comparison against non-security bugs.

Thus, our objective in this question is twofold: (a) identify trends between specific types of bugs and the occurrence of future vulnerabilities, and (b) compare those types to baselines of SLOC, security-related bugs, and features.

Analysis. We performed logistic regression analysis to evaluate these patterns by comparing the model quality. We based the model quality in two statistical tests: **model goodness of fit** (how well the created model fits the data) and **model performance** (how well model is able explain the data). We must point out, however, that these analyses for to *comparing categories* of bugs, not to create an overall vulnerability prediction model as has been done in other literature [2]–[4], [18].

To implement this approach first we built a base model based on SLOC, to serve as the baseline for evaluating the improvement in model quality that each pre-bug-metrics produces. Our RQ1 results, as well as other studies [27] have shown that files with higher SLOC are more likely to be vulnerable.

We continued this process by performing a forward selection of the variables, building models adding one pre-release bug type at a time, we managed to identify 4 groups of pre-release bug variables that can be represented in logical bug categories and show model quality improvement over the individual pre-release bug types. The description of each of these category groups can be found on Table VI.

Finally we compared the variance on the overall quality of the output that each category group introduces.

Goodness of fit: These metrics present a way to evaluate how well a model fits the data when compared to other models, in our case the base model. In this step we measured: *a*) The AIC of each model *b*) and the D^2.

TABLE VI
CATEGORY MODELS BASED ON BUG TYPES

Metric	Description
fit-base	Our base model based only on SLOC.
fit-num-pre-bugs	Based on SLOC + num-pre-bugs.
fit-features	Bug category based on SLOC + num-pre-features.
fit-security	Bug category based on SLOC + num-pre-security-bugs.
fit-stability	Bug category based on SLOC + num-pre-stability-bugs + num-pre-compatibility-bugs + num-pre-regression-bugs.
fit-build	Bug category based on SLOC + num-pre-build-bugs + num-pre-tests-fails-bugs.

Our results show that adding the pre-release bug metrics have a positive effect overall on the quality of the models, but we found this relation to be weak and not constant.

On Release 5.0 and 11.0, the model **fit-num-pre-bugs**, show the strongest correlation with improvement of model goodness of fit.

We also found that **fit-security** and **fit-features** present the stronger positive association when compared to the other category models on almost all releases, averaging -2.78% and -0.84% of AIC reduction respectively. One exception to this trend is Release 27, where the model **fit-features** is associated with a lower quality model (increased AIC). While other category models are also associated with lower AIC, we found that the improvement is not as significant when compared to the baseline SLOC model.

In terms of D^2 again the category **fit-security** and category **fit-features** show the strongest correlation averaging 73.47% and 22.66% improvement respectively over the base model. Reconfirming the trend that among the evaluated bug categories **fit-security** and **fit-features** present the strongest correlation with chances of vulnerabilities.

The complete result set for the goodness of fit metrics is shown in Table VII. The models with the best fit are shown in bold, and represent the strongest association with post-release vulnerabilities.

Model Performance: In this step we evaluate the variance in explanatory power that each bug category produces, when the category models are used to predict the probability of vulnerabilities and compared against next release data. We used next-release validation on four releases, meaning that we used the data from one release to train a model, then apply that model to the subsequent release. Next-release validation is particularly helpful because it simulates the data that the team could have feasibly collected at that time in history.

In our validation, we examining the following statistics: *a*) Precision and recall, *b*) F-measure, *c*) Area under the ROC (AUC).

We compare these statistics to the output of the base model. We present our findings in Table VIII showing the performance improvement of each model.

TABLE VII
Goodness of fit Metrics for Pre-Release of Bug based category Models Per Release

		Release 5.0		
Model	**AIC**	**AIC DEC**	**D^2**	**D^2 INC**
fit-base	1424.40	-	0.04	-
fit-num-pre-bugs	**1377.40**	-3.30%	**0.07**	86.39%
fit-security	1402.20	-1.56%	0.05	42.68%
fit-features	1423.90	-0.04%	0.04	4.34%
fit-stability	1419.50	-0.34%	0.05	19.25%
fit-build	1423.80	-0.04%	0.04	8.09%

		Release 11.0		
Model	**AIC**	**AIC DEC**	**D^2**	**D^2 INC**
fit-base	3098.40	-	0.08	-
fit-num-pre-bugs	3093.00	-0.17%	0.08	2.69%
fit-security	3095.90	-0.08%	0.08	1.65%
fit-features	**3071.10**	-0.88%	**0.09**	10.73%
fit-stability	3090.50	-0.25%	0.09	5.08%
fit-build	3099.20	0.03%	0.08	1.15%

		Release 19.0		
Model	**AIC**	**AIC DEC**	**D^2**	**D^2 INC**
fit-base	2365.60	-	0.04	-
fit-num-pre-bugs	2367.00	0.06%	0.04	0.60%
fit-security	2358.60	-0.30%	0.05	8.50%
fit-features	**2337.90**	-1.17%	**0.05**	28.06%
fit-stability	2361.30	-0.18%	0.05	9.75%
fit-build	2350.10	-0.66%	0.05	18.41%

		Release 27.0		
Model	**AIC**	**AIC DEC**	**D^2**	**D^2 INC**
fit-base	2432.50	-	0.04	-
fit-num-pre-bugs	2434.40	0.08%	0.04	0.02%
fit-security	**2286.10**	-6.02%	**0.10**	134.10%
fit-features	2434.10	0.07%	0.04	0.34%
fit-stability	2427.50	-0.21%	0.05	9.88%
fit-build	2421.90	-0.44%	0.05	13.11%

		Release 35.0		
Model	**AIC**	**AIC DEC**	**D^2**	**D^2 INC**
fit-base	1038.20	-	0.03	-
fit-num-pre-bugs	1040	0.17%	0.03	0.47%
fit-security	**976.42**	-5.95%	**0.09**	180.41%
fit-features	1015.50	-2.19%	0.06	69.82%
fit-stability	1038.70	0.05%	0.04	15.50%
fit-build	1038.70	0.05%	0.04	25.01%

TABLE VIII
Performance Metrics for Pre-Release Bug Based Category Model Per Release

		Release 11.0		
Model	**Precision**	**Recall**	**F-measure**	**AUC**
fit-base	**0.29**	0.35	0.32	70.89%
fit-num-pre-bugs	0.24	0.49	0.32	66.63%
fit-security	0.28	0.34	0.31	70.41%
fit-features	0.25	**0.50**	**0.33**	**71.97%**
fit-stability	0.26	0.41	0.32	69.92%
fit-build	0.25	0.48	**0.33**	70.86%

		Release 19.0		
Model	**Precision**	**Recall**	**F-measure**	**AUC**
fit-base	**0.11**	0.31	0.16	66.84%
fit-num-pre-bugs	0.09	**0.48**	0.15	66.12%
fit-security	**0.11**	0.34	**0.17**	67.06%
fit-features	0.10	0.46	0.16	**68.77%**
fit-stability	0.09	0.41	0.15	66.40%
fit-build	0.09	0.44	0.16	67.61%

		Release 27.0		
Model	**Precision**	**Recall**	**F-measure**	**AUC**
fit-base	0.11	0.32	0.16	66.48%
fit-num-pre-bugs	0.07	**0.51**	0.13	66.57%
fit-security	**0.15**	0.40	**0.21**	**70.03%**
fit-features	0.07	0.50	0.11	65.54%
fit-stability	0.08	0.46	0.13	66.35%
fit-build	0.08	0.48	0.14	68.42%

		Release 35.0		
Model	**Precision**	**Recall**	**F-measure**	**AUC**
fit-base	0.04	0.35	0.07	66.50%
fit-num-pre-bugs	0.03	**0.54**	0.05	66.48%
fit-security	**0.06**	0.51	**0.10**	**75.34%**
fit-features	0.03	0.47	0.06	67.31%
fit-stability	0.03	0.45	0.06	65.90%
fit-build	0.03	0.45	0.06	67.41%

In these results we notice that the category model **fit-security** is associated with improvement of the precision in 3 out of 4 evaluated releases. The category model **fit-num-pre-bugs** is associated with an increase in the recall measurement in in 3 out of 4 evaluated releases. The results show that again **fit-security** is associated with an improvement in F-measure. In terms of AUC we found that the models **fit-security** and **fit-features** have a stronger positive relation than the other models.

Results Interpretation: Summarizing the results from both aspects "model goodness fit" and "model performance" we notice that, when compared to baselines of features, SLOC, and security bugs, traditional non-security bugs have a positive but weak association with vulnerabilities. Among the results, the strongest positive correlation with overall model quality was found in the variables pre-release feature bugs and pre-release security bugs. But even with this existing association, the improvement over our baseline model is very small and the overall predictive power is relatively small compared to the literature [3], [12], [27], [28].

This result is particularly interesting given that several sub-categories of bugs are related to security properties. System stability is key to providing availability (i.e. preventing denial-of-service attacks), so one might assume that stability problems in a file's past may lead to vulnerability problems in the future, but this effect was small.

This weak association indicate that bugs and vulnerabilities are empirically dissimilar groups. And it can be tough call to identify future vulnerabilities based solely on the pre-release bug history. We advice to include other vulnerability identification patterns where needed [28].

RQ3. Do the source code files with the most bugs also have the most vulnerabilities?

Motivation. In this question, we want to simulate how bugs could be used in practice. Consider the situation of a last-minute security audit where we have limited resources to perform a thorough code inspection. If we used the concept of "bugginess" as our only guide to prioritize those inspections, how well will that guide work at inspecting the files that would later need to be fixed for vulnerabilities?

Analysis. To examine this question, we present a **lift curve** for each release demonstrating how many vulnerabilities exist in the top-ranked files by defect density. Figure 3 shows these results by release. As an example interpretation of the charts, in Release 11.0, 60% of the post-release vulnerabilities can be found in the top 30% of the buggiest files. While this ranking is better than random (i.e. roughly a diagonal line), the prediction capabilities of defect density is significantly worse than what is found in the vulnerability prediction literature [27], [28].

Consider also these results in a scenario. Suppose we are only able to inspect 20 files that are non-trivial, say 25 lines of code or more. (While these numbers are arbitrary, we chose them as reasonable simulations of a rushed security audit.) If we rank the files at each release as by defect density (i.e. num-pre-bugs / SLOC), then those files would, at best, contain 1.4% of the vulnerabilities. An optimal top-20 file ranking would account for an average of 12% of vulnerabilities, so ranking by defect density is far from optimal.

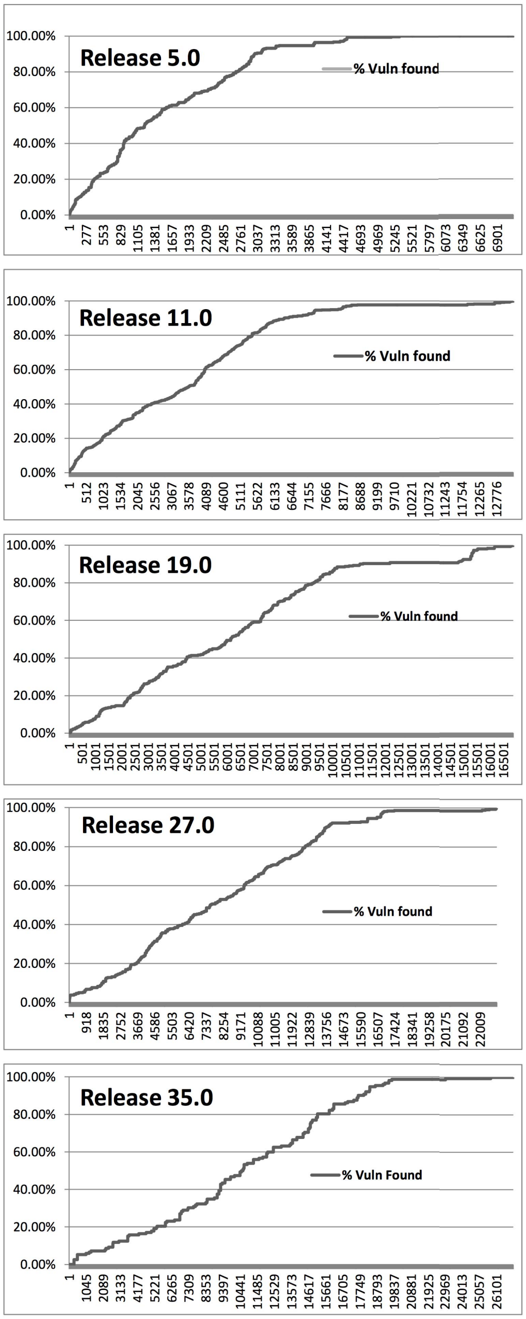

Fig. 3. Lift curves of % vulnerabilities found when ranked by num-pre-bugs shows a weak association

The results above show that the buggiest files have some (but not many) vulnerabilities. The situation gets worse, however, when we examine the question from the other direction: ranking by files with the most *vulnerabilities* per SLOC. In that situation, across the five releases, none of the Top 20 files with the most vulnerabilities per SLOC appear in the Top 20 of the buggiest files.

VII. Threats to Validity

We chose the Chromium project as a large, open source project to be representative of many large software projects. But, as with any empirical study, these results may be specific to the Chromium project. Labels such as the ones we used were based on our own investigation of Chromium, and may not generalize to other project (e.g. different development teams may define "stability bugs" differently).

By presenting the variance in the model fitness and performance metrics, we caution the reader about the overall predictability of vulnerabilities. Other studies (even our own [27]) have shown prediction models of vulnerabilities that outperform the models here. The multiple regression models are formed as a comparison of multiple groups of metrics.

We compared the quality of the models from one release with data of the next release. It is also possible to combine models from multiple releases to identify bug patterns that take more time to reveal its effects on vulnerabilities. We could for example average the prediction results of Release 5.0, 11.0, 19.0 and 27.0 to evaluate the probability of vulnerabilities in Release 35.0. However we postpone this investigation for future work.

VIII. Summary

In this study we evaluated the correlation between pre-release bugs and post-release vulnerabilities on the Chromium project. The results show that, while an empirical connection between bugs and vulnerabilities exist, the connection is considerably weak. The strongest indicators of vulnerability are past security-related bugs and new features - neither of which are non-security bugs. Furthermore, the buggiest files do not intersect with the files with many vulnerabilities. This evidence underscores the conceptual difference between bugs and vulnerabilities, and indicates that additional empirical research must be directed at vulnerability data specifically.

Acknowledgments

This research is supported by the National Science Foundation (grant CCF-1441444). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We thank the Software Archeology group at RIT for their valuable contributions to this work.

References

[1] I. V. Krsul, "Software vulnerability analysis," Ph.D. dissertation, Purdue University, 1998.

[2] T.-H. Chen, S. Thomas, M. Nagappan, and A. Hassan, "Explaining software defects using topic models," in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, June 2012, pp. 189–198.

[3] M. Gegick, P. Rotella, and L. Williams, "Predicting attack-prone components," in *Software Testing Verification and Validation, 2009. ICST '09. International Conference on*, April 2009, pp. 181–190.

[4] A. Meneely, A. C. R. Tejeda, B. Spates, S. Trudeau, D. Neuberger, K. Whitlock, C. Ketant, and K. Davis, "An empirical investigation of socio-technical code review metrics and security vulnerabilities," in *Proceedings of the 6th International Workshop on Social Software Engineering*, ser. SSE 2014. New York, NY, USA: ACM, 2014, pp. 37–44. [Online]. Available: http://doi.acm.org/10.1145/2661685.2661687

[5] D. Tegarden, S. Sheetz, and D. Monarchi, "Effectiveness of traditional software metrics for object-oriented systems," in *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on*, vol. iv, Jan 1992, pp. 359–368 vol.4.

[6] M. Mukaka, "A guide to appropriate use of correlation coefficient in medical research," *Malawi Medical Journal*, vol. 24, no. 3, pp. 69–71, 2012.

[7] N. F. Schneidewind, "Methodology for validating software metrics," *Software Engineering, IEEE Transactions on*, vol. 18, no. 5, pp. 410–422, 1992.

[8] J. Cohen, *Statistical power analysis for the behavioral sciences*. Academic press, 2013.

[9] J. Ruscio, "A probability-based measure of effect size: Robustness to base rates and other factors." *Psychological Methods*, vol. 13, no. 1, p. 19, 2008.

[10] K. K. Zakzanis, "Statistics to tell the truth, the whole truth, and nothing but the truth: formulae, illustrative numerical examples, and heuristic interpretation of effect size analyses for neuropsychological researchers," *Archives of clinical neuropsychology*, vol. 16, no. 7, pp. 653–667, 2001.

[11] J. Cohen, "Statistical power analysis," *Current directions in psychological science*, pp. 98–101, 1992.

[12] A. Cruz and K. Ochimizu, "Towards logistic regression models for predicting fault-prone code across software projects," in *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, Oct 2009, pp. 460–463.

[13] A. E. Raftery, "Bayesian model selection in social research," *Sociological methodology*, vol. 25, pp. 111–164, 1995.

[14] K. P. Burnham and D. R. Anderson, "Multimodel inference understanding aic and bic in model selection," *Sociological methods & research*, vol. 33, no. 2, pp. 261–304, 2004.

[15] A. Guisan and N. E. Zimmermann, "Predictive habitat distribution models in ecology," *Ecological modelling*, vol. 135, no. 2, pp. 147–186, 2000.

[16] T. Y. Chen, F.-C. Kuo, and R. Merkel, "On the statistical properties of the f-measure," in *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*, Sept 2004, pp. 146–153.

[17] M. Gegick, L. Williams, J. Osborne, and M. Vouk, "Prioritizing software security fortification throughcode-level metrics," in *Proceedings of the 4th ACM workshop on Quality of protection*. ACM, 2008, pp. 31–38.

[18] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan, "High-impact defects: a study of breakage and surprise defects," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 300–310.

[19] D. Mitropoulos, V. Karakoidas, P. Louridas, G. Gousios, and D. Spinellis, "Dismal code: Studying the evolution of security bugs," in *Proceedings of the LASER 2013 (LASER 2013)*. Arlington, VA: USENIX, 2013, pp. 37–48. [Online]. Available: https://www.usenix.org/laser2013/program/mitropoulos

[20] D. Mitropoulos, G. Gousios, and D. Spinellis, "Measuring the occurrence of security-related bugs through software evolution," in *Informatics (PCI), 2012 16th Panhellenic Conference on*, Oct 2012, pp. 117–122.

[21] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. ichi Matsumoto, "The impact of mislabelling on the performance and interpretation of defect prediction models," in *Proc. of the 37th Int'l Conf. on Software Engineering (ICSE)*, 2015, p. To appear.

[22] W. Poncin, A. Serebrenik, and M. van den Brand, "Process mining software repositories," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, March 2011, pp. 5–14.

[23] C. Bird, P. Rigby, E. Barr, D. Hamilton, D. German, and P. Devanbu, "The promises and perils of mining git," in *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, May 2009, pp. 1–10.

[24] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. IEEE, 2003, pp. 23–32.

[25] D. Behl, S. Handa, and A. Arora, "A bug mining tool to identify and analyze security bugs using naive bayes and tf-idf," in *Optimization, Reliabilty, and Information Technology (ICROIT), 2014 International Conference on*, Feb 2014, pp. 294–299.

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

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

[28] 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.