An Empirical Investigation of Socio-technical Code Review Metrics and Security Vulnerabilities

Andrew Meneely, Alberto C. Rodriguez Tejeda, Brian Spates, Shannon Trudeau, Danielle Neuberger, Katherine Whitlock, Christopher Ketant, Kayla Davis
Department of Software Engineering
Rochester Institute of Technology
134 Lomb Memorial Drive,
Rochester, NY, USA 14623
1+585-475-7829
{axmvse,acr9218,bxs4361,smt9020,drn5447,www5610,cck9672,kd9205}@rit.edu

ABSTRACT
One of the guiding principles of open source software development is to use crowds of developers to keep a watchful eye on source code. Eric Raymond declared Linus' Law as "many eyes make all bugs shallow", with the socio-technical argument that high quality open source software emerges when developers combine together their collective experience and expertise to review code collaboratively. Vulnerabilities are a particularly nasty set of bugs that can be rare, difficult to reproduce, and require specialized skills to recognize. Does Linus' Law apply to vulnerabilities empirically? In this study, we analyzed 159,254 code reviews, 185,948 Git commits, and 667 post-release vulnerabilities in the Chromium browser project. We formulated, collected, and analyzed various metrics related to Linus' Law to explore the connection between collaborative reviews and vulnerabilities that were missed by the review process. Our statistical association results showed that source code files reviewed by more developers are, counter-intuitively, more likely to be vulnerable (even after accounting for file size). However, files are less likely to be vulnerable if they were reviewed by developers who had experience participating on prior vulnerability-fixing reviews. The results indicate that lack of security experience and lack of collaborator familiarity are key risk factors in considering Linus' Law with vulnerabilities.

Categories and Subject Descriptors
D.2.8 [**Software Engineering**]: Metrics – *process metrics, product metrics*.

General Terms
Measurement, Security, Human Factors.

Keywords
Code review, socio-technical, vulnerability

1. INTRODUCTION
One of the guiding principles of open source software development is to leverage crowds of developers to keep a watchful eye on source code. In his famous 2001 essay [1], Eric Raymond dubbed Linus' Law as:

> "Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix will be obvious to someone. […] Many eyes make all bugs shallow"

According to Raymond's reasoning, a diversity of developer perspectives ought to be embraced as the method of producing high quality open source software. The assertion is both social and technical nature: when developers bring together their collective experience and expertise, bugs are found, fixed, and prevented.

Security vulnerabilities, however, are a particularly nasty type of bug for developers. A vulnerability can be just a simple coding mistake, but it can lead to catastrophic failures that compromise the confidentiality, integrity, and availability of the system. The effectiveness of Linus' Law certainly draws scrutiny after recent severe vulnerability situations such as the Heartbleed vulnerability in OpenSSL, and the many other vulnerabilities regularly found in large open source projects such as Apache, Firefox, and Google Chrome. Thus, we must ask the empirical question: does Linus' Law apply to vulnerabilities?

In prior and related work [2]–[7], researchers have found some connections between Linus' Law and vulnerabilities, but with several counter-intuitive caveats such as having more committers to a file being associated with vulnerabilities [4], [5]. Most of prior and related work, however, quantifies Linus' Law as committers, not code reviewers. In this study, we approach Linus' Law as it applies to vulnerabilities to examine the effectiveness of the code review practice.

The objective of this study is to investigate Linus' Law as it applies to security by measuring and analyzing code review participation. We conducted an historical study of the Chromium browser open source project by mining their Git logs, Rietveld code review logs, source code, and vulnerability disclosures. Based on the Chromium development process, we developed metrics to analyze two properties of code reviews: thoroughness and socio-technical familiarity. We analyzed our metrics using a statistical association analysis, and compared the metrics to each other in terms of risk factors for vulnerabilities. The contributions of this work are:

- The development of nine metrics that quantify factors of thoroughness and socio-technical familiarity;
- Empirical evaluation of association and risk factors of those metrics with regards to post-release vulnerabilities in the Chromium project.

SSE'14, November 17, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3227-9/14/11...$15.00
http://dx.doi.org/10.1145/2661685.2661687

This paper is organized as follows. Sections 2 and 3 cover basic terminology used in this paper and related work in this area. Section 4 describes the Chromium project and the data sources we collected from. Section 5 describes our methodology of processing the data in preparation for analysis. Section 6 covers our analysis of the metrics. Section 7 discusses threats to validity and Section 8 closes the paper with a brief summary.

2. TERMINOLOGY

The metrics we propose are **socio-technical metrics**, a term we borrow from sociology to describe the connection between two people in the context of work-related collaboration[8], which is of a social and technical nature. The term "technical" is not referring to technology-related activities, but to the more general idea of technicality, skill, and labor. Thus, a socio-technical metric focuses on people and their interactions with others.

We use the term **commit** to describe a recorded, individual change to the source code as recorded by the version control system (e.g. Git). Some version control systems call these "revisions", or "change sets". Each commit contains a **patch**, or a compared difference between a source code file prior to the commit and after the commit. A proposed commit that is being reviewed is a **patch set**.

We also use the term **vulnerability** as an "instance of a fault that violates an [implicit or explicit] security policy" [9]. All vulnerabilities examined in this study are **post-release**, meaning that the system was released with this problem, the system was potentially exposed, and the team had to release a patch to end-users fix the problem.

We also extracted more terms that are specific to the study of the Chromium project: **reviewer**, **patch set, participant**, **security-experienced**, **code review owner**, and **directory owner**. We define these terms in Section 4.1

3. RELATED WORK

The recent related work in this field has been with some work on bugs and code reviews [10], [11] and our own prior work on Linus' Law and vulnerabilities [2]–[5].

McIntosh et al. [10] published a study mining code review logs (very similar to the logs in our study) to examine the relationship between reviews and software quality. The authors mined the Gerrit logs of Qt, VTK, and ITK projects, where code reviews are optional. They found that, overall, low coverage and participation is statistically linked to software quality. This study has several key differences from ours: mainly in coverage, vulnerabilities, and socio-technical familiarity. For Chromium, the concept of code review coverage is not appropriate since all commits are required to have a code review, so the coverage would effectively be 100% for Chromium. Secondly, they examined all types of bugs to study software quality and we examined vulnerabilities to study security in particular. Thirdly, they did not factor developer experience and socio-technical familiarity as we did. We did borrow several concepts from this study, however, in measuring participation and fast reviews. This study serves as an interesting baseline of comparison between bugs and vulnerabilities.

In another recent study, Bosu [11] examined the commits that introduced vulnerabilities and analyzed the experience of the authors who contributed those changes. In that study, the results showed that experienced developers were more likely to contribute vulnerable code changes. This study differs from ours in that the data points in their study are commits and our study examines the history of files. Their measurements of experience are much different than ours as well. This work is similar to our prior work of exploring properties vulnerability-contributing commits [12].

In prior work [2]–[7], we have found a consistent statistical association between metrics that quantify developer activity and vulnerabilities. In one study [3], we investigated code churn along with complexity and developer activity metrics as indicators of security vulnerabilities. We studied the likelihood of a file being vulnerable by collecting the metrics before a release and predicting them against the reported vulnerabilities post-release. We also developed some socio-technical metrics for the study, such as metrics dealing with the contribution to a file and the socio-technical developer network surrounding a file. We conducted studies on the Red Hat Enterprise Linux kernel and Mozilla Firefox. The combined model of code churn, complexity, and developer activity metrics resulted in predicting 80% of known vulnerable files and less than 25% false positives. We also have studied and developed socio-technical metrics in various other case studies, such as interactive churn [2], developer networks [6], [7], and developer network clustering [4].

4. THE CHROMIUM BROWSER PROJECT

The data set we analyzed in this project is the Chromium browser, popularly known as the open source project behind Google Chrome. Chromium is an open source project, and effort is primarily sponsored by and driven by Google employees.

The Chromium browser is a large product consisting of over 4 million Source Lines of Code (SLOC), over 19,000 source code files in 36 different programming languages, primarily C and C++. Our analysis covers 185, 948 Git commits, 159, 254 Rietveld code reviews, and 667 post-release vulnerabilities.

Chromium employs a large dependency list, and developers often contribute to the dependency projects such as Skia, Webkit, and V8. In this study, we did not consider code reviews nor vulnerabilities in Chromium dependencies. Furthermore, the Chromium Browser project has a sister project for an operating system called ChromiumOS, which is not part of this study. In this paper, any reference to "Chromium" implies the "Chromium Browser Project".

4.1 Code Reviews in Chromium

The Chromium project enforces a strict policy that *every* source control commit must be reviewed prior to being integrated into the system. This code review process is designed with dual purposes: to improve the overall quality of the system and to encourage collaboration amongst developers.

The Chromium project uses the Rietveld tool to manage their code reviews. Rietveld is a web application that integrates with Subversion, Git, and Chromium's build server called Trybot into a single place for code review. Developers can upload their proposed changes to the system, invite reviewers, and receive feedback either with in-line comments or just a message at the bottom of the review. Developers can upload revised patch sets throughout the course of the discussion. The review must be approved by developers with the proper permissions. Once finished, Rietveld integrates the patch into the Chromium tree.

To illustrate how the process works, consider the following example scenario. Bonnie is a developer who wishes to push her latest changes into the Chromium tree. Using command-line tools, Bonnie creates a **patch set**, which is a proposed source control commit that gets uploaded to Rietveld. Bonnie then invites two **reviewers** to provide feedback on her patch set: Josiah and Andy.

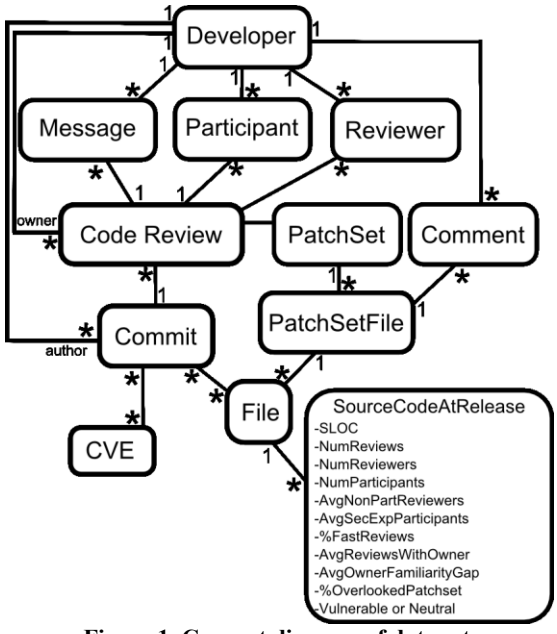

Figure 1. Concept diagram of data set

Josiah is also a **directory owner**, which means that he has the authority to approve a code review related to the source code directories that Bonnie is committing to. Three months prior to this review, Josiah participated in a review of an integer overflow vulnerability, so he brings his experience to this discussion. Josiah then writes some more comments, and Bonnie responds by revising her patch set again. After some more discourse via messages, Josiah writes "LGTM" (Looks Good To Me), which flags the review as "approved" because Josiah is a directory owner. At that point, Bonnie can make any last-minute changes to her patch set. She then hits the "commit" button and Rietveld integrates the patch set into the Chromium tree. Andy, while asked to review, did not participate but the code is integrated anyway. From this above scenario, we provide the following definitions:

- **Code Review Owner**: the developer who developed the patch set and initiated the code review. i.e. Bonnie
- **Directory Owner**: a developer who has the authority to approve changes to a given source code directory via the LGTM command, i.e. Josiah.
- **Reviewer**: a developer who was explicitly asked to provide feedback on the code review. While Rietveld allows it, we do not count the code review owner to be reviewers on their own code review. In the example scenario, Josiah and Andy are reviewers, but not Bonnie.
- **Participant**: a developer who provided at least one in-line comment or a message to the code review. In the example scenario, Josiah and Bonnie are participants, but not Andy.
- **Security-Experienced**: a participant who, at the time of the review, had been a participant on a prior review that was the fix for a post-release vulnerability. Josiah was security-experience in the scenario. See Section 6.1.2 for further discussion and analysis.
- **Reviews with Owner**: a count for a given participant of the number of prior reviews they had co-participated on with the owner at the time of the review. See Section 6.1.2 for further discussion and analysis.

4.2 Vulnerabilities in Chromium

Internet browsers have a very high expectation for security as they are the front lines between users and the untrusted data of the World Wide Web. Over the past six years of development the Chromium project has relentlessly pursued the fixing and preventing of vulnerabilities in their system, such as instituting a bounty program for finding vulnerabilities. Each vulnerability is reported by the Chromium team and is registered in the National Vulnerability Database (NVD) and is given a Common Vulnerability Enumeration identifier (CVE). Vulnerabilities registered in the CVE are post-release vulnerabilities, which means that they represent potential exposures to users.

Furthermore, the Chromium team has released traceability data from CVEs to either the commit or the code review that fixed the vulnerability. In situations where this linkage did not exist, we investigated the data and found the linkage manually. In many situations, vulnerabilities were reported by the Chromium team that existed in dependencies such as Webkit. These vulnerabilities were ignored from this study as they did not actually exist in the Chromium source tree.

5. DATA PREPROCESSING

In this project, we collected data from the following sources:

- Rietveld code review data
- Git log of commits integrated into the system
- Vulnerability reports from the NVD, and acknowledged by the Chromium security response team.
- SLOC data from the CLOC tool (http://cloc.sourceforge.net)

We collected this data from the beginning of the project in September 2008 through November 2013.

5.1 Scraping Data

We began our data collection with the Git logs of Chromium. The Git logs of Chromium represent the main line of development over time. Chromium makes major releases often and sometimes maintains older releases, but in this study we focused only on the main line of development. Each commit was auto-generated by the Rietveld tool, with attribution to the original author of the patch set, and a message that contained a code review ID and a description of the change. Reviewers are mentioned in commits sometimes, but we ignored this field in favor of the Rietveld data itself.

We obtained our Rietveld code review data by parsing the code review IDs from the commits, and scraped the Rietveld logs from code reviews integrated into the system. Code reviews that never were successful in being integrated into the system, therefore, are not included in this study.

The vulnerability data originated from the NVD and required an acknowledgement of the vulnerability from Google Chrome's release log to be considered credible. This data directly links to bug reports, commits, and code reviews. In some cases, the link was missing so we conducted a manual investigation to complete the traceability. Finally, empirical research on source code files must account for size as potential confounding factor, so we collected source code counts. In this study, we consider files with the following extension to be source code: `.h, .cc, .js, .cpp, .gyp, .py, .c, .make, .sh, .S, .scons, .sb, Makefile`.

5.2 Integrating the Data Sources

We used the Ruby on Rails ActiveRecord libraries to parse, model, verify, and query the data. We employed a nightly build scheme to check long-running queries and parsers to ensure that our metrics are accurate. The Entity-Relationship Diagram (sans attributes for most entities) can be found in Figure 1.

The analysis for this paper originates at **SourceCodeAtRelease**, which is a snapshot of a given source code file at a single release. While we only consider source code files in our metric analysis, however, the File entity in our database includes other types of textual data such as documentation. Regarding the release, Chromium utilizes a rapid release cycle of a major release approximately every two months. For this study we consider one release of Chromium: Release 11 on January 31st 2011. We chose to set our snapshot at release 11 because it was the nearest halfway major release between the present time and the beginning of the project. For all fields on **SourceCodeAtRelease** except for the **VulnerableOrNeutral** field, we collect metrics from the genesis of the project to January 31st 2011.

In this study we, examine the vulnerabilities in Chromium by way of their fixes. Each vulnerability can be traced to one or more Git commits, and therefore one or more code reviews. We utilize the CVE data in two ways:

- A vulnerability after 2011-01-31 represents a severe mistake that was missed by the team and escaped to the field (i.e. **VulnerableOrNeutral**)
- Vulnerabilities fixed prior to 2011-01-31 are used in our security experience metric described in Section 6.1.2.

For the **VulnerableOrNeutral** field in **SourceCodeAtRelease**, we count a given source code file as "vulnerable" if it was fixed for an acknowledged CVE after January 31st 2011, and we use the label "neutral" if not as in prior work [4], [5]. From the **SourceCodeFileAtRelease** entity, we compute our metrics based on queries to the other associated entities. For Release 11, we had 17,005 source code files, 745 (4.4%) of which were vulnerable. The other metrics are discussed in Section 6.1

5.3 Filtering Developer Email Addresses

One of the most important elements of extracting information about developer collaboration is to properly identify the developers themselves. A Chromium developer can interact with the various artifacts of the project through a variety of tools, such as Git, the Rietveld tool, or via email. We found that, in many situations, developers would use multiple emails to identify themselves throughout the project. To mitigate this, we identify each developer in our database using a unique integer. Through manual inspection of every email address mentioned in Chromium history, we conducted the following filters to condense multiple emails down to a single ID for situations where the same developer used multiple emails.

- All emails were converted to lowercase letters
- Plus-tags were removed (e.g. bonnie+reviews@foo.com → bonnie@foo.com)
- Google Temp Accounts were converted to their original form (e.g. bonnie%foo.com@gtempaccount.com →bonnie@foo.com)
- Any @google.com accounts were converted to @chromium.org addresses
- Common misspellings of "chromium.org" were corrected (13 variations found in the data)
- Automated emails were not given a developer ID, such as commit-bot@chromium.org, and therefore not counted

Finally, the overall list was manually inspected by two authors on this paper to determine if any other determinations could be made. This disambiguation filtering was a useful one, as the original

Table 1. Metrics computed for each source code file of Chromium release 11

Metric	Description	Rationale
SLOC	Number of source lines of code, not counting comments or whitespace	More code, more problems (used as a confounding factor in analysis).
NumReviews	The number of code reviews a source code file has had	Code that is reviewed more may be less likely to have vulnerabilities
NumReviewers	Number of different people who were invited to provide feedback on a file	Code with more reviewers may be less likely to have vulnerabilities
NumParticipants	Number of different people who provided feedback on a file over all of its code reviews	Code with more participants may be less likely to have vulnerabilities
AvgNonPartReviewer	Average number of reviewers per code review who did not participate	Lack of feedback may mean that a vulnerability was missed
%ManyReviewers	Percentage of a reviews with three or more reviewers	Chromium guidelines recommend only one or two reviewers per review so responsibility is clear (i.e. no bystander apathy)
%FastReviews	Percentage of reviews that exceeded 200 lines per hour.	Code reviews exceeding 200 lines per hour is correlated with lower quality software [13]
AvgSecurityExperienced-Participants	Average number of code review participants who had previously participated on a vulnerability fix	Inspecting the fix of a post-release vulnerability is a valuable experience that developers can bring to the next
AvgPriorReviewsWithOwner	The average number of prior reviews co-participated with the review owner.	Working with the same people over time can provide more socio-technical familiarity, providing more opportunity to bring up security concerns
AvgOwnerFamiliarityGap	Per code review, the maximum number of prior reviews with owner minus the minimum number of prior reviews with owner. Averaged over all reviews for a file	Teams with people who are both familiar and unfamiliar with the owner can, in aggregate, provide both familiarity and objectivity to the review

distinct email count was over 4,000 developers and was reduced to 2,464 developers.

6. ANALYSIS & RESULTS

With the data parsed and processed, we formulated our metrics based on exploring the concepts of Linus' Law and open source development. We conducted two forms of analysis: association and risk factors. Association is to examine if a metric is related to vulnerabilities in a statistically significant way, and risk factors is used to compare the metrics to each other to discover which was the largest risk to security.

6.1 Code Review Metrics

Descriptions of our metrics can be found in Table 1, providing a definition and a brief rationale as to why that metric may be related to security according to the reasoning of Linus' Law. We approach these metrics in two categories: *thoroughness* and *socio-technical familiarity.*

6.1.1 Thoroughness: "Many Eyes"

Question: Do vulnerable and neutral files differ regarding code review thoroughness?

Metrics: **`NumReviews`**, **`NumReviewers`**, **`NumParticipants`**, **`AvgNonPartReviewers`**, **`%ManyReviewers`**, **`%FastReviews`**.

Linus' Law dictates that open source software products improve with "many eyes" because the more people who watch, the more problems will become apparent. Thus, a thorough review of source code would involve many people over time. We quantify this idea is a variety of ways based on the schema of our data set and the Chromium development process.

More people can mean more reviews, so we count **`NumReviews`**. In Chromium, however, each commit requires its own code review, so the **`NumReviews`** metric is closely related to the well-studied code churn metrics [2], [3], [14], [15] that have already been shown to be correlated with vulnerabilities [2], [3]. Thus, we also examine "many eyes" by the number of different developers involved. In some situations, many different people review the same code, in other situations only a handful of people may be repeatedly reviewing code over time. Our definitions in Section 4.1 indicate that Reviewers are asked to provide feedback and Participants are developers who actually do provide feedback. Since Reviewers and Participants are two overlapping groups of developers, we count each of those as a separate metric (**`NumReviewers`** and **`NumParticipants`**).

With vulnerabilities, the group of non-participating reviewers can be problematic: a developer was asked to provide feedback and did not follow through. For a single code review or a single developer this may not be a big problem. However, when aggregating the code reviews over the entire history of a file then significant problems in the code may be overlooked. To aggregate, we use **`AvgNonParticipatingReviewers`**, or the average number of non-participating reviewers per code review over the history of a given source code file.

Furthermore, based on our readings of the Chromium guidelines published on their website, we noticed that the team discourages code reviews with three or more reviewers. This somewhat counter-intuitive recommendation comes from their own experience: when reviews have three or more reviewers, the reviewers get confused about what their role in the review is. This concept is similar in nature to the psychological effect of "bystander apathy" [16] observed in the psychology and sociology disciplines where collaborators be less likely to contribute to a project when more people are involved. Another term used in this area is "diffusion of responsibility". We note that Chromium's guideline of having one or two reviewers does not directly contradict Linus' Law, since participants are not limited. Our metric **`%ManyReviewers`** for a source code file is the percentage of reviews that had three or more reviewers.

Beyond the number of people participating or not participating in a code review, we examine one final indicator of a code review lacking thoroughness: **`%FastReviews`**. We borrow this idea from related work [10], [13] that indicates a team of developers cannot adequately review code at more than 200 LOC per hour. For this metric, we compute the total size of the patch sets involved and compute the time from the opening of the code review to the final approval. If the code review was opened and closed at a rate greater than 200 LOC per hour, we consider it "fast" since code may be un-reviewed.

6.1.2 Socio-Technical Familiarity: "The Fix is Obvious to Someone"

Question: Do vulnerable and neutral files differ regarding the socio-technical familiarity of the developers?

Metrics: **`AvgSecurityExperiencedParticipants`**, **`AvgPriorReviewsWithOwner`**, **`AvgOwnerFamiliarityGap`**.

In Linus' Law, Eric Raymond was not simply making a statement about crowds, he was making a statement about the advantages of socio-technical diversity amongst colleagues. Each code review is essentially a miniature team with a singular task of finding any problems they can fix. As with any team, dynamics can come into play that are both social and technical in nature, such as the experience that one developer brings to the proverbial table or the history between two participants on giving each other feedback. Metrics that are both social and technical in nature are often referred to as socio-technical [17], and historically have been correlated with bugs and vulnerabilities [3], [5], [6], [12]. In this study, we measure socio-technical familiarity as it applies to two concepts: *familiarity with security* and *familiarity with people.*

Security experts [18] talk about the "hacker mindset" as being a difficult set of skills to develop that go beyond standard software development acumen. To find a vulnerability, developers must put aside their assumptions about how the system should work and start thinking about how the system actually does work, and if it can be exploited. Understanding vulnerabilities conceptually is also much different than finding them in a complex system such as Chromium. In Chromium, only 4.4% of the source code files have had a vulnerability, meaning a developer may never encounter a vulnerability in their day-to-day work. In a recent study of Apache HTTPD [12], we found that all of vulnerabilities in Apache HTTPD can be traced to 124 vulnerability-contributing commits, out of a population of over 25,000 commits. Yet, the experience of fixing a vulnerability, or inspecting the fix of a vulnerability, can also be a valuable one as the discussion must touch upon a key question: "how can we prevent this from ever happening again?" We define the metric **`AvgSecurity-ExperiencedParts`** as the average number of participants who had participated in inspecting a fix for a vulnerability prior to their participation on a given code review.

For example, suppose a developer Kelly participates in three reviews in three days, where the second code review was fixing a vulnerability. As a participant on the first two reviews, Kelly is not security-experienced, but for the third review she is (since she potentially brought her security-related experience to the review).

We also note that this metric utilizes vulnerability data from *prior* to the Chromium 11 release (see Section 5.2), and the metric is analyzed against a vulnerable/neutral flag for *after* the release of Chromium 11, so no two vulnerabilities are correlated with themselves with this metric even though they both use security data. For the entire population and six-year history of Chromium, only 299 (12%) of developers have participated on a code review that fixed a vulnerability.

To measure the concept of familiarity with people, we also examine developer participation history, but in relation to each other. When people work together on a code review, they must provide feedback to each other. Over time, colleagues who repeatedly work together may gain a rapport with each other that can lead to more efficient communication and effective code reviews that find and fix vulnerabilities. For this concept, we compute the **`AvgPriorReviewsWithOwner`**, which is the average number of reviews that participants had co-participated with the owner of the code review. Thus, if developers continue working with each other, this **`AvgPriorReviewsWithOwner`** will increase.

However, as developers gain familiarity with each other they may also lose objectivity. New participants on a review can provide a fresh perspective that is invaluable to finding vulnerabilities. To account for *both* familiarity and objectivity, we do the following:

1. For each review, compute the number of prior reviews each participant had with the owner
2. Compute the maximum and minimum number of prior reviews with the owner for the code review.
3. OwnerFamiliarityGap is the maximum minus the minimum for the code review.
4. **`AvgOwnerFamiliarityGap`** is the average OwnerFamiliarityGap for a source code file

A high gap then indicates that a team had *both* low prior reviews and high prior reviews, indicating a potentially good mixture of objectivity and familiarity. A low gap means that all participants had roughly the same number of prior reviews with the owner as each other.

6.2 Association Analysis

To examine if each of the metrics summarized in Table 1 are related to security vulnerabilities, we examine the difference between the vulnerable files and the neutral files in terms of each metric. As suggested in other metrics validation studies [19] for not having a normality assumption, we use the non-parametric **Mann-Whitney-Wilcoxon (MWW)** test and compare the medians. Three outcomes are possible from this test:

- The metric is statistically higher for vulnerable files than neutral files;
- The metric is statistically lower for vulnerable files than for neutral files; or
- The metric is not different between neutral and vulnerable files at a statistically significant level ($p<0.005$, see below).

Since we are examining 10 metrics (9 metrics and the control metric SLOC), we must account for multiple hypothesis testing errors. We use the conservative Bonferoni correction by dividing the alpha threshold by the number of hypotheses, thus we check for $p<0.005$ instead of the standard $p<0.05$.

Furthermore, we must account for the variability of source code file size. Studies have already shown that files with high lines of code are likely to have vulnerabilities [3]. Given the variability of SLOC across files, SLOC can be a confounding factor in many analyses. For example, a file with high SLOC may be more likely to have more reviewers simply because it has more code to review. To account for file size, we divide our metrics by SLOC first and compute medians and MWW test based on the normalized data. We present the results of our association analysis in Table 2.

Table 2. Association analysis results using medians of vulnerable vs. neutral files and MWW test

Metric	Median		Median Per SLOC			Interpretation
	vuln	neut	vuln	neut	MWW p<0.005?	
SLOC	189	68	--	--	Yes*	Vulnerable files are larger than neutral files
NumReviews	14	2	0.08	0.04	Yes	Vulnerable files have had more reviews per SLOC
NumReviewers	11	3	0.06	0.05	Yes	Vulnerable files have had more people review them per SLOC
NumParticipants	14	4	0.08	0.06	Yes	Vulnerable files had more participants per SLOC
AvgNonPartReviewer	0.18	0.14	0.0004	0.0006	No	No conclusive evidence of association
%ManyReviewers	4.0%	0	0.0001	0	Yes	Vulnerable files had higher % of reviews with 3 or more reviewers per SLOC
%FastReviews	28%	33%	0.1%	0.2%	Yes	Vulnerable files had a lower % of reviews over 200 lines/hour, per SLOC
AvgSecurityExpPart	1.0	1.0	0.003	0.006	Yes	Vulnerable files had a lower occurrence of security-experienced participants per SLOC
AvgPriorReviewsWithOwner,	21.6	21.5	0.10	0.21	Yes	Vulnerable files had a lower occurrence of prior reviews with owner per SLOC
AvgOwnerFamiliarityGap	20.2	20	0.09	0.21	Yes	Vulnerable files had less of a mixture between high a number of prior reviews with the review owner and low number of prior reviews with owner per SLOC
*Except for SLOC itself, all MWW tests were conducted after first dividing by SLOC to account for varying file size effects. The p-values were compared against 0.05/10 for a Bonferoni correction for multiple hypothesis tests						

Among the results, only one difference was not considered to be conclusive: **AvgNonParticipatingReviewer**. From this result, we cannot conclude one way or another that enough evidence exists that vulnerable files are any different from neutral files with respect to non-participating reviewers.

For all other metrics, the results were statistically significant. In some situations, however the association was in an interesting direction. Vulnerable files tended to have *many* more reviews than neutral files (**NumReviews**) per SLOC, however since every commit is required to have a code review this measurement may simply be a proxy measurement of code churn which is already known to be associated with vulnerabilities [2], [3].

Vulnerable files also tended to have more people (**NumReviewers, NumParticipants, %ManyReviewers**), even after controlling for SLOC. We note that these results are different than in McIntosh et al. [10], which indicated that bugs are less likely to occur in highly-reviewed files with high participation. We interpret these results as characteristic of vulnerabilities contrasted to traditional bugs: even with more people, vulnerabilities are still missed. On the face, this result also contradicts Linus' Law, at least the "many eyes" portion. In the situation of **%ManyReviewers**, the result upholds the Chromium guidelines of not having too many people on reviews, providing more evidence of a potential "bystander apathy" effect found in social sciences [16].

For fast reviews (**%FastReviews**), the result is also counter-intuitive: vulnerable files had a lower history of having hastily-reviewed code. Again, this is the opposite of McIntosh et al. [10] in their study of bugs. We interpret this result as indicating that vulnerabilities are, again, characteristically different from traditional bugs, meaning that even when they have more reviews, are reviewed more thoroughly, with more people, vulnerabilities are still missed even if the overall quality improves.

However, the results also demonstrate that socio-technical factors are important for vulnerabilities as well. Vulnerable files, while having more participants (**NumParticipants**), had fewer security-experienced participants per SLOC (**AvgSecurity-ExperiencedParts**). This result upholds the socio-technical part of Linus' Law: that when diversity of experience is leveraged properly, fixes are obvious to someone.

The "familiarity with people" metrics were associated such that vulnerable files had a lack of familiarity with the owner of the code review. Vulnerable files had a lower occurrence of prior reviews with the owner per SLOC, and the gap was much higher for neutral files than vulnerable files (**AvgPriorReviewsWith-Owner**, **AvgOwnerFamiliarityGap**), indicating that vulnerable files had a wider mixture prior experience with each other.

Overall, these results indicate that security is related to factors related to Linus' Law, but the relationship is more nuanced than the blanket statement of "many eyes make all bugs shallow".

6.3 Risk Increase Analysis

The benefit of an association analysis is that we can see whether or not a given metric is related to security vulnerabilities in a statistically significant way on a metric-by-metric basis. However, the drawback of association is that it does not indicate which metrics are more strongly linked to vulnerabilities. For example, we saw that lack of security experience is a risk to a file being vulnerable, but is that risk larger compared to, say, number of participants?

Table 3. Risk increase analysis results

Metric m, per SLOC* Threshold (t)	p(vuln | m>t)	p(vuln |m<=t)	Risk increase**
NumReviews (0.06)	6.7%	3.9%	2.24
NumReviewers (0.05)	5.1%	3.9%	1.30
NumParticipants (0.07)	5.1%	4.0%	1.26
%ManyReviewers (5.4e-5)	6.2%	3.2%	1.93
%FastReviews (0.002)	3.6%	5.5%	1.50
AvgSecurityExpPart (5e-3)	4.0%	6.1%	1.51
AvgPriorReviewsWithOwner (0.16)	3.8%	7.5%	2.28
AvgOwnerFamiliarityGap (0.15)	3.3%	5.9%	1.77

* All metrics are divided by SLOC to account for varying file size effects

** "Increase" is defined as from the lower p(vuln) to the larger p(vuln). e.g. A file is 2.24 times more likely to be vulnerable if it had more than 6 reviews per 100 lines of code

To examine relative risk, we adapt the methods of Schneidewind again [19] to produce discriminators of metrics. We determine a fixed threshold of the metric, make it a classifier, then compute the increase in risk that a file is vulnerable based on that classifier. We base our analysis on a standard "increase in risk" [20] formula. To determine the threshold, we compute the midpoint of the two medians for vulnerable and neutral files, as in Equation 1.

$$t = \frac{median(m_{\text{vuln}}) + median(m_{\text{neut}})}{2} \quad (1)$$

This threshold is arbitrary and other, better thresholds likely exist in the space of each metric. However, the purpose of this Risk Increase Analysis is to gauge the *relative* risk increase from one metric to another. The method in Equation 1of identifying a threshold can be consistently applied to all of our metrics.

Next, we compute the ratio of vulnerable to neutral files for being above and below the threshold, then examine the amount of risk increased. Equation 2 shows the formula:

$$a = \frac{m_{vuln} \cap m_{m>t}}{m_{neut} \cap m_{m>t}}$$
$$b = \frac{m_{vuln} \cap m_{m\leq t}}{m_{neut} \cap m_{m\leq t}}$$
$$r = \frac{\max(a, b)}{\min(a, b)} \quad (2)$$

Table 3 contains our results, showing thresholds, proportions, and risk factors. We only analyze metrics that were shown to be statistically significant in the association analysis in Section 6.2.

Among the results, the most strongly associated risk factor was prior experience with the owner (**AvgPriorReviews-WithOwner)**. Historically, the proportion of vulnerable files was 2.28 times larger when **AvgPriorReviewsWithOwner** was over under 0.16 per SLOC, indicating that, among the 9 metrics in this study, this factor was the largest risk. Stated more broadly, a file is 2.28 more times likely to be vulnerable if the code review participants are inexperienced with working with each other.

The second highest risk factor in our analysis was **NumReviews**, which is in line with prior results on churn. The next highest risk factors were **%ManyReviewers** and **AvgOwnerFamiliarity-**

Gap, with risk factors that nearly double when the metric exceeds its threshold.

7. THREATS TO VALIDITY

For construct validity, we approached our data collection and analysis process from a rigorous test-driven approach. Using unit tests, sample data, and nightly builds we were able to keep our system stable and counts accurate. However, as with all empirical research projects, our scripts could have mistakes in them. Also, our analysis assumes that once a file has a post-release vulnerability, it's considered vulnerable with no discrimination in weights. In reality, vulnerabilities differ greatly in severity, but in this study we consider them equal.

For internal validity, our metric for **%FastReviews** is based on when the code review was approved as calendar date, which can be misleading for some reviews. Many fast code reviews may be missed if they were not approved until, say, after a weekend. No other information is provided on how much time reviewers actually spend reviewing code.

For external validity, this is one particular case study so we cannot say that our results hold for other projects. We chose this case study because of its scale, popularity of the product, and the diligence with which they maintain their vulnerability and code review data. To see if these results generalize, more case studies are needed to triangulate what is specific to Chromium and what is general.

8. SUMMARY

The objective of this study is to investigate Linus' Law as it applies to security by measuring and analyzing code review participation. Overall, the results clearly indicate the counter-intuitive result that more reviews and reviewers are statistically linked to vulnerable files, contradicting Linus' Law while and differentiating vulnerabilities from bugs in other studies [10]. However, the socio-technical side of the argument behind Linus' Law still holds for vulnerabilities, in that developer experience and team experience are statistically linked to vulnerabilities. More case studies and more metrics will illuminate other nuances to such a complex statement like Linus' Law.

9. ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation (#1441444) and the Collaborative Research Experiences for Undergraduates program at the Computing Research Association Committee on the Status of Women in Computing Research. Any opinions, finding, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

10. REFERENCES

[1] E. S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, 1st ed. O'Reilly Media, 2010.

[2] A. Meneely and O. Williams, "Interactive Churn: Socio-Technical Variants on Code Churn Metrics," in *Int'l Workshop on Software Quality*, 2012, pp. 1–10.

[3] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, 2011.

[4] A. Meneely and L. Williams, "Strengthening the Empirical Analysis of the Relationship Between Linus' Law and Software Security," in *Empirical Software Engineering and Measurement*, Bolzano-Bozen, Italy, 2010, pp. 1–10.

[5] A. Meneely and L. Williams, "Secure Open Source Collaboration: an Empirical Study of Linus' Law," in *Int'l Conference on Computer and Communications Security (CCS)*, Chicago, Illinois, USA, 2009, pp. 453–462.

[6] A. Meneely and L. Williams, "Socio-Technical Developer Networks: Should We Trust Our Measurements?," presented at the International Conference on Software Engineering, Waikiki, Hawaii, USA, 2011, p. to appear.

[7] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting Failures with Developer Networks and Social Network Analysis," in *16th ACM SIGSOFT International Symposium on Foundations of software engineering*, Atlanta, Georgia, 2008, pp. 13–23.

[8] E. L. Trist and K. W. Bamforth, "Some social and psychological consequences of the longwall method of coal-getting," *Technol. Organ. Innov. Early Debates*, p. 79, 2000.

[9] I. V. Krsual, "Software Vulnerability Analysis." PhD Dissertation, Purdue University, 1998.

[10] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, VTK, and ITK Projects," in *11th Working Conference on Mining Software Repositories*, New York, NY, USA, 2014, pp. 192–201.

[11] A. Bosu, "Characteristics of the Vulnerable Code Changes Identified Through Peer Code Review," in *36th International Conference on Software Engineering*, New York, NY, USA, 2014, pp. 736–738.

[12] 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," *Proc. 2013 ACM-IEEE Int. Symp. Empir. Softw. Eng. Meas.*, p. to appear, 2013.

[13] C. F. Kemerer and M. C. Paulk, "The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data," *IEEE Trans Softw Eng*, vol. 35, no. 4, pp. 534–550, Jul. 2009.

[14] N. Nagappan and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density," in *27th international Conference on Software Engineering (ICSE)*, St. Louis, MO, USA, 2005, pp. 284–292.

[15] J. C. Munson and S. G. Elbaum, "Code churn: a measure for estimating the impact of code change," in *Software Maintenance, 1998. Proceedings. International Conference on*, 1998, pp. 24 –31.

[16] S. M. Garcia, K. Weaver, G. B. Moskowitz, and J. M. Darley, "Crowded minds: The implicit bystander effect," *J. Pers. Soc. Psychol.*, vol. 83, no. 4, pp. 843–853, 2002.

[17] E. Coakes, "Socio-technical thinking: an holistic viewpoint," in *Socio-technical and human cognition elements of information systems*, IGI Publishing, 2003, pp. 1–4.

[18] G. McGraw, *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[19] N. F. Schneidewind, "Methodology for Validating Software Metrics," *IEEE Trans. Softw. Eng. TSE*, vol. 18, no. 5, pp. 410–422, 1992.

[20] P. V. Rao, *Statistical Research Methods in the Life Sciences*, 1st ed. Duxbury Press, 1997.