UC Berkeley
UC Berkeley Previously Published Works

Title
A Large-Scale Study of Modern Code Review and Security in Open Source Projects

Permalink
https://escholarship.org/uc/item/86d5w0gn

ISBN
9781450353052

Authors
Thompson, Christopher
Wagner, David

Publication Date
2017-11-08

DOI
10.1145/3127005.3127014

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

A Large-Scale Study of Modern Code Review and Security in Open Source Projects

Christopher Thompson
University of California, Berkeley
cthompson@cs.berkeley.edu

David Wagner
University of California, Berkeley
daw@cs.berkeley.edu

ABSTRACT

Background: Evidence for the relationship between code review process and software security (and software quality) has the potential to help improve code review automation and tools, as well as provide a better understanding of the economics for improving software security and quality. Prior work in this area has primarily been limited to case studies of a small handful of software projects. *Aims:* We investigate the effect of modern code review on software security. We extend and generalize prior work that has looked at code review and software quality. *Method:* We gather a very large dataset from GitHub (3,126 projects in 149 languages, with 489,038 issues and 382,771 pull requests), and use a combination of quantification techniques and multiple regression modeling to study the relationship between code review coverage and participation and software quality and security. *Results:* We find that code review coverage has a significant effect on software security. We confirm prior results that found a relationship between code review coverage and software defects. Most notably, we find evidence of a negative relationship between code review of pull requests and the number of security bugs reported in a project. *Conclusions:* Our results suggest that implementing code review policies within the pull request model of development may have a positive effect on the quality and security of software.

CCS CONCEPTS

• **Security and privacy → Economics of security and privacy**; *Software security engineering*; • **Software and its engineering → Software development methods**; *Collaboration in software development*;

KEYWORDS

mining software repositories, software quality, software security, code review, multiple regression models, quantification models

1 INTRODUCTION

Modern code review takes the heavyweight process of formal code inspection, simplifies it, and supports it with tools, allowing peers and others to review code as it is being added to a project [41]. Formal software inspection generally involves a separate team of inspectors examining a portion of the code to generate a list of defects to later be fixed [2]. In contrast, modern code review is much more lightweight, focusing on reviewing small sets of changes before they are integrated into the project. In addition, modern code review can be much more collaborative, with both reviewer and author working to find the best fix for a defect or solution to an architectural problem [41]. Code review can help transfer knowledge, improve team awareness, or improve the quality of solutions to software problems [3]. It also has a positive effect on understandability and collective ownership of code [6].

There has been much research into code review process and how it relates to software quality in general [2, 3, 28, 45], but the connection to the security of software has been less thoroughly explored. Software security is a form of software quality, but there is reason to believe that software vulnerabilities may be different from general software defects [8, 29, 46].

Empirical evidence for the relationship between code review process and software security (and software quality) could help improve decision making around code review process. Prior work in this area has primarily been limited to case studies of a small handful of software projects [28, 30, 45]. To the best of our knowledge, we present the first large-scale (in terms of number of repositories studied) analysis of the relationship between code review and software security in open source projects.

GitHub, the largest online repository hosting service, encourages modern code review through its pull request system. A pull request (or "PR") is an easy way to accept contributions from outside developers. Pull requests provide a single place for discussion about a set of proposed changes ("discussion comments"), and for comments on specific parts of the code itself ("review comments").

However, not all projects have such a strictly defined or enforced development process, or even consistently review changes made to their code base. Code review coverage is the proportion of changes that are reviewed before being integrated into the code base for a project. Code review participation is the degree of reviewer involvement in review. Previous studies have examined the effects of code review coverage and participation on software quality [28, 45] and software security [30] among a handful of large software projects. We extend and generalize these prior studies by performing quantitative analysis of these effects in a very large corpus of open source software repositories on GitHub.

Our contributions in this paper are as follows:

- We create a novel neural network-based quantification model (a model for predicting the class distribution of a dataset) which outperforms a range of existing quantification techniques at the task of estimating the number of security issues in a project's issue tracker.
- We perform a large-scale analysis of projects on GitHub, using multiple regression analysis to study the relationship between code review coverage and participation and: (1) the number of issues in a project, and (2) the number of issues that are security bugs in a project, while controlling for a range of confounding factors.

PROMISE'17, November 8, 2017, Toronto, Canada
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 DATA PROCESSING

We focused our investigation on the population of GitHub repositories that had at least 10 pushes, 5 issues, and 4 contributors from 2012 to 2014. This is a conservatively low threshold for projects that have had at least some active development, some active use, and more than one developer, and thus a conservatively low threshold for projects that might benefit from having a set code review process (we hypothesize, but do not investigate, that a higher threshold would yield a population that exhibits *stronger* effects from code review). We used the GitHub Archive [24], a collection of all public GitHub events, to generate a list of all such repositories. This gave us 48,612 candidate repositories in total. From this candidate set, we randomly sampled 5000 repositories.

We wrote a scraper to pull all non-commit data (such as descriptions and issue and pull request text and metadata) for a GitHub repository through the GitHub API [20], and used it to gather data for each repository in our sample. After scraping, we had 4,937 repositories (due to some churn in GitHub repositories).

We queried GitHub to obtain the top three languages used by each repository. For each such language, we manually labeled it on two independent axes: Whether it is a programming language (versus a markup language like HTML, etc.), and whether it is memory-safe or not. A programming language is "memory-safe" if its programs are protected from a variety of defects relating to memory accesses (see Szekeres et al. [44] for a systematization of memory safety issues).

Starting from the set of repositories we scraped, we filtered out those that failed to meet our minimum criteria for analysis:

- Repositories with a `created_at` date later than their `pushed_at` date (these had not been active since being created on GitHub; 13 repositories)
- Repositories with fewer than 5 issues (these typically had their issue trackers moved to a different location; 264 repositories)
- Repositories with fewer than 4 contributors (these typically were borderline cases where our initial filter on distinct committer e-mail addresses over-estimated the number of GitHub users involved; 1008 repositories)
- Repositories with no pull requests (406 repositories)
- Repositories where the primary language is empty (GitHub did not detect any language for the main content of the repository, so we ruled these as not being software project repositories; 83 repositories)
- Repositories where none of the top three languages are programming languages (a conservative heuristic for a repository not being a software project; 37 repositories)

This left us with 3,126 repositories in 149 languages, containing 489,038 issues and 382,771 pull requests. Table 1 shows a summary of the repositories in our sample. Table 2 lists the top 10 languages used in our repository sample and the total number of bytes in files of each. We use this dataset for our regression analysis in Section 4.

We found that, in our entire sample, security bugs make up 4.6% of all issues (as predicted by our final trained quantifier, see Section 3 for how we derived this estimate). This proportion is close to the results of Ray et al. [40] where 2% of bug fixing commits were categorized as security-related.

Table 1: Summary of our sample of GitHub repositories.

	median	min	max
Stars	42.5	0	338880
Pull Requests	34	1	9060
Contributors	11	4	435
Issues	46	5	8381
Size (B)	4163	22	6090985
Age (days)	1103	0.3078	3048
Avg Commenters per PR	0.9274	0	5
Unreviewed PRs	2	0	1189
% Unreviewed PRs	5.56	0	100
Security Bugs	1	0	175
% Security Bugs	0	0	68

Table 2: Top primary languages in our repository sample.

Language	# Repositories	Total Primary Size (MB)
JavaScript	660	1083.27
Python	418	282.98
Ruby	315	93.62
Java	305	755.75
PHP	268	540.87
C++	172	1147.48
C	140	2929.24
CSS	110	23.13
HTML	107	476.02
C#	83	177.83

3 QUANTIFYING SECURITY ISSUES

Ultimately, our basic approach involves applying multiple linear regression to study the relationship between code review and security. We count the number of security bugs reported against a particular project and use this as a proxy measure of the security of the project; we then construct a regression model using our code review metrics and other variables as the independent variables. The challenge is that this would seem to require examining each issue reported on the GitHub issue tracker for each of the 3,126 repositories in our sample, to determine whether each issue is security-related or not.

Unfortunately, with 489,038 issues in our repository dataset, it is infeasible to manually label each issue as a security bug or non-security bug. Instead, we use machine learning techniques to construct a *quantifier* that can estimate, for each project, the proportion of issues that are security-related. Our approach is an instance of *quantification*, which is concerned with estimating the distribution of classes in some pool of instances: e.g., estimating the fraction of positive instances, or in our case, estimating the fraction

of issues that are security-related. Quantification was originally formalized by Forman [14] and has since been applied to a variety of fields, from sentiment analysis [16] to political science [23] to operations research [14]. We build on the techniques in the literature and extend them to construct an accurate quantifier for our purposes.

One of the insights of the quantification literature is that it can be easier to estimate the fraction of instances (out of some large pool) that are positive than to classify individual instances. In our setting, we found that accurately classifying whether an individual issue is a security bug is a difficult task (reaching at best 80-85% classification accuracy). In contrast, quantification error can be much smaller than classification error (the same models achieved around 8% average absolute quantification error—see "RF CC" in Figure 1). Intuitively, the false positives and the false negatives of the classifier cancel each other out when the proportion is being calculated.

For our research goals, the crucial insight is that we are only concerned with estimating the *aggregate* proportion of security issues in a repository, rather than any of the individual labels (or predicting the label of a new issue). In particular, our regression models only require knowing a count of how many security issues were reported against a particular project, but not the ability to identify which specific issues were security-related. Thus, quantification makes it possible to analyze very large data sets and achieve more accurate and generalizable results from our linear regression models.

Using our best methods described below, we were able to build a quantifier that estimates the fraction of issues that are security-related with an average absolute error of only 4%.

We distinguish our task of quantification from prior work in vulnerability *prediction* models (as in work by Gegick et al. [17] and Hall et al. [21]). We are concerned about textual issues reported in a project's issue tracker, rather than identifying vulnerable components in the project's source code. Our goal is not *prediction*, where we would want to correctly label each new instance we see (such as has been addressed in work by Gegick et al. [18]). Instead, the goal of our models is to estimate the proportion of positive (security-related) instances (issues) in an existing population.

3.1 Basic Quantification Techniques

We start by reviewing background material on quantification. Quantification is a supervised machine learning task: we are given a training set of labeled instances (x_i, y_i). Now, given a test set S, the goal is to estimate what fraction of instances in S are from each class. Quantification differs from standard supervised learning methods in that the class distribution of the training set might differ from the class distribution of the test set: e.g., the proportion of positive instances might not be the same.

Many classifiers work best when the proportion of positive instances in the test set is the same as the proportion of positive instances in the training set (i.e., the test set and training set have the same underlying distribution). However, in quantification, this assumption is violated: we train a single model on a training set with some fixed proportion of positives, and then we will apply it to different test sets, each of which might have a different proportion of positives. This can cause biased results, if care is not taken. Techniques for quantification are typically designed to address this challenge and to tolerate differences in class distribution between the training set and test set [14]; a good quantification approach should be robust to variations in class distribution.

Several methods for quantification have been studied in the literature. The "naive" approach to quantification, called *Classify and Count (CC)* [14], predicts the class distribution of a test set by using a classifier to predict the label y_i for each instance and then counting the number of instances with each label to estimate the proportion of positive instances:

$$\hat{p} = \frac{1}{N} \sum_i y_i .$$

In other words, we simply classify each instance in the test set and then count what fraction of them were classified as positive.

The *Adjusted Count (AC)* method [14] tries to estimate the bias of the underlying classifier and adjust for it. Using k-fold cross-validation, the classifier's true positive rate (tpr) and false positive rate (fpr) can be estimated. For our experiments, we used $k = 10$. The adjusted predicted proportion is then

$$\hat{p}_{AC} = \frac{\hat{p} - fpr}{tpr - fpr}.$$

Some classifiers (such as logistic regression) output not only a predicted class y, but also a probability score—an estimate of the probability that the instance has class y. The *Probabilistic Adjusted Classify and Count (PACC)* method builds on the AC method by using the probability estimates from the classifier instead of the predicted labels [5]. It also uses estimates of the expected true positive and false positive rates (computed using cross-validation, as with AC). The adjusted predicted proportion is then

$$\hat{p}_{PACC} = \frac{\hat{p} - \mathrm{E}[fpr]}{\mathrm{E}[tpr] - \mathrm{E}[fpr]}.$$

3.2 Quantification Error Optimization

More recently, researchers have proposed training models to optimize the quantification error directly instead of optimizing the classification error and then correcting it post-facto [4, 13, 32]. Forman was the first to use Kullback-Leibler Divergence (KLD), which measures the difference between two probability distributions, as a measure of quantification error [14]. For quantification, KLD measures the difference between the true class distribution and the predicted class distribution. Given two discrete probability distributions P and $\hat{P}$, the KLD is defined as

$$\mathrm{KLD}(P|\hat{P}) = \sum_i P(i) \log \frac{P(i)}{\hat{P}(i)}.$$

The KLD is the amount of information lost when $\hat{P}$ is used to approximate P. A lower KLD indicates that the model will be more accurate at the quantification task. Thus, rather than training a model to maximize accuracy (as is typically done for classification), for quantification we can train the model to minimize KLD.

Esuli and Sebastiani [13] use structured prediction (based on $\mathrm{SVM}_{\mathrm{perf}}$ [25, 26]) to train an SVM classifier that minimizes the KLD loss. They call their quantifier SVM(KLD), and it has been used for

sentiment analysis tasks [16]. However, we were unable to reproduce comparable KLD scores on simple test datasets, and found the existing implementation difficult to use. Other researchers report subpar performance from SVM(KLD) compared to the simpler CC, AC, or PACC quantification methods for sentiment analysis tasks [36].

3.3 Our Quantifier

Building on the idea of optimizing for quantification error instead of accuracy, we construct a neural network quantifier trained using TensorFlow [1] to minimize the KLD loss. TensorFlow allows us to create and optimize custom machine learning models and has built-in support for minimizing the cross-entropy loss. We express the KLD in terms of the cross entropy via

$$\mathrm{KLD}(P|\hat{P}) = H(P, \hat{P}) - H(P),$$

that is, the difference of the cross entropy of P and $\hat{P}$ and the entropy of P. Because for any given training iteration the entropy of the true class distribution $H(P)$ will be constant, minimizing the cross entropy $H(P, \hat{P})$ will also minimize the KLD.

We implement a fully-connected feed-forward network with two hidden layers of 128 and 32 neurons, respectively. The hidden layers use the ReLU activation function. The final linear output layer is computed using a softmax function so that the output is a probability distribution. Training uses stochastic gradient descent with mini-batches, using the gradient of the cross entropy loss between the predicted batch class distribution and the true batch class distribution.

Naive random batching can cause the neural network to simply learn the class distribution of the training set. To combat this, we implemented random proportion batching: for each batch, we initially set the batch to contain a random sample of instances from the training set; then we randomly select a proportion of positives p (from some range of proportions) and select a maximum-size subset of the initial set such that the sub-batch proportion is p; finally, we evaluate the model's KLD on that sub-batch. This objective function is equivalent to minimizing the model's average KLD, where we are averaging over a range of proportions p for the true proportion of positives. This training procedure forces the model to be accurate at the quantification task over a wide range of values for the true proportion p of positives, and thus provides robustness to variations in the class distribution.

Our network architecture is kept intentionally simple, and as shown below, it performs very well. We leave heavy optimization of the network design or testing of alternative architectures to future work.

3.4 Feature Extraction

In all of our quantification models we used the following features. We extract features from the text of the issue using the "bag-of-words" approach over all 1- and 2-grams. We extract all of the text from each issue, remove all HTML markup and punctuation from the text, stem each word (remove affixes, such as plurals or "-ing") using the WordNet [39] lemmatizer provided by the Natural Language Toolkit (NLTK) [27], compute token counts, and apply a term-frequency inverse document frequency (TF-IDF) transform [42] to the token counts. Separately, we also extract all of the labels (tags) from each issue, normalize them to lowercase,[1] and apply a TF-IDF transform to obtain additional features. We also count the number of comments on each issue, and extract the primary language of the repository. The combination of all of these were used as our features for our quantifiers.

3.5 Methodology

To train and evaluate our quantifiers, we hand-labeled 1,097 issues to indicate which ones were security issues and which were not. We reserved 10% of them (110 issues) as a test set, and used the remaining 987 issues as a training set.

We selected the issues in our dataset to reduce class imbalance. Because security issues are such a small fraction of the total population of issues, simply selecting a random subset of issues would have left us with too few security issues in the training and test sets. Therefore, we used the tags on each issue as a heuristic to help us find more issues that might be security-related. In particular, we collected a set of issues with the "security" tag, and a set of issues that lacked the "security" tag; both sets were taken from issues created from January 2015 to April 2016, using the GitHub Archive (issue events in the GitHub Archive before 2015 did not have tag information). We restricted the non-security-tagged issues to one per repository, in order to prevent any single project from dominating our dataset. We did not limit the security-tagged issues, due to the limited number of such issues. This left us with 84,652 issues without the "security" tag and 1,015 issues with the "security" tag. We took all of the security-tagged issues along with a random sample of 2,000 of the non-security-tagged issues and scraped all of the text and metadata for each issue using the GitHub API:

- The owner and name of the repository
- The name of the user who created the issue
- The text of the issue
- The list of any tags assigned to the issue
- The text of all comments on the issue
- The usernames of the commenters
- The time the issue was created
- The time the issue was last updated
- The time the issue was closed (if applicable)

We then hand-labeled 1,097 of these issues, manually inspecting each to determine it was a "security bug" (the uneven number is an artifact of our data processing pipeline and issues in our overall archival sample that had since been deleted). We considered an issue filed against the repository to be a security bug if it demonstrated a defect in the software that had security implications or fell into a known security bug class (such as buffer overruns, XSS, CSRF, hard-coded credentials, etc. [10]), even if it was not specifically described in that way in the bug report. We treated the following as not being security bugs:

- Out-of-date or insecure dependencies
- Documentation issues
- Enhancement requests not related to fundamental insecurity of existing software

[1] To avoid the potential for overfitting due to interaction with how we selected issues to be hand-labeled, we remove the tag "security" if present.

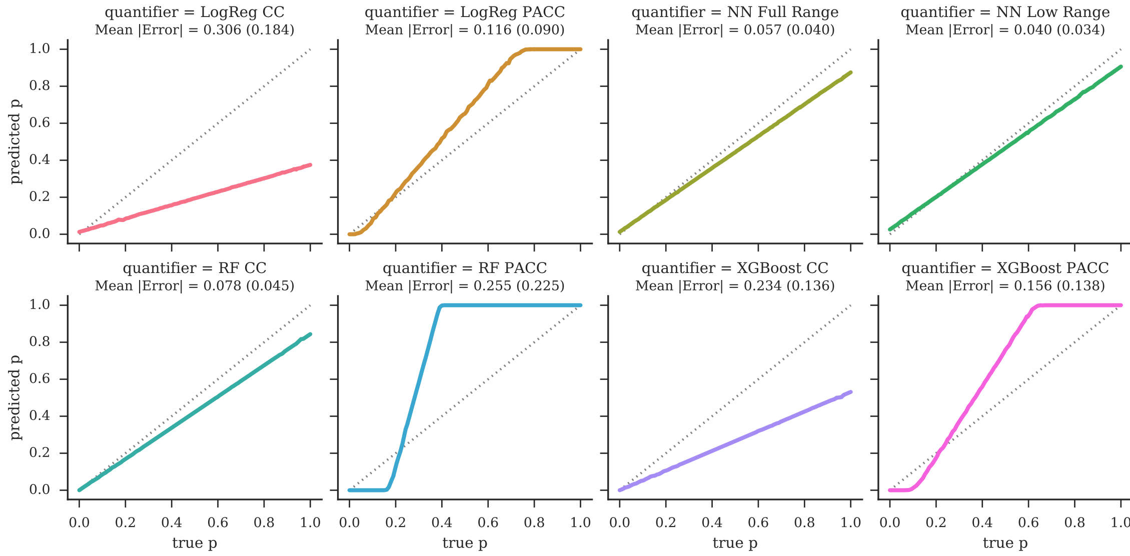

Figure 1: Plots of predicted proportion vs. true proportion for our quantifiers on our reserved test set. The dotted line marks the line $y = x$, which represents the ideal (a quantifier with no error); closer to the dotted line is better. Each quantifier is labeled with the mean absolute error over all proportions and the standard error of that mean. Our "low range" neural network quantifier trained over $p \in [0.0, 0.1]$ shows the best performance, with a mean absolute error of only 4%.

We compensated for the generally low prevalence of security bugs by hand-labeling more of the issues from our "security"-tagged set. After hand-labeling we had 224 security bug issues and 873 non-security bug issues.

The "security" tag on GitHub had a precision of 37% and a recall of 99% when compared to our hand-labeling. This very low precision validates our decision to hand-label issues and develop quantification models to analyze our main repository corpus.

3.6 Evaluation

We implemented and tested a variety of quantifiers. We tested CC, AC, and PACC with logistic regression, SVM, random forest, and XGBoost [9] classifiers under a variety of settings, along with various configurations of our neural network-based quantifier. Fig. 1 shows the relative error over all proportions $p \in [0.0, 1.0]$ for the top-performing quantifiers. Our neural network quantifier, when trained on proportions of positives in the range $[0.0, 0.1]$ (the "low range"), performed the best on our test set, with the lowest mean absolute error (0.04) and the lowest mean KLD (0.01), so we adopt it for all our subsequent analysis.

4 REGRESSION DESIGN

In our study design, we seek to answer the following four research questions:

(RQ1) *Is there a relationship between code review coverage and the number of issues in a project?*

(RQ2) *Is there a relationship between code review coverage and the number of security bugs reported to a project?*

(RQ3) *Is there a relationship between code review participation and the number of issues in a project?*

(RQ4) *Is there a relationship between code review participation and the number of security bugs reported to a project?*

Our research questions are similar to McIntosh et al. [28] and Ray et al. [40], and we use similar model construction techniques. We use multiple linear regression modeling to describe the relationship between code review coverage and participation (our explanatory variables) and both the number of issues and the number of security bugs filed on each project (our response variables). In our models we also include a number of control explanatory variables (such as the age, size, churn, number of contributors, and stars for each repository). Table 3 explains each of our explanatory variables.

We manually inspect the pairwise relationships between our response variable and each explanatory variable for non-linearity. Following standard regression analysis techniques for improving linearity [15], we apply a log transformation ($\log(x + 1)$) to each metric with natural number values.

To reduce collinearity, before building our regression models we check the pairwise Spearman rank correlation (ρ) between our explanatory variables. We use Spearman rank correlation since our explanatory variables are not necessarily normally distributed. For any pair that is highly correlated ($|\rho| > 0.7$ [31]), we only include one of the two in our model. Additionally, after building our regression models, we calculate the Variance Inflation Factor (VIF), a measure of multicollinearity, for each explanatory variable in the model. No variables in our models exceeded a VIF of 5, which is considered a conservative threshold [37].

To determine whether the coefficients for each explanatory variable are significantly different from zero, we perform a t-test on each to determine a p-value. If a coefficient is not significantly different from zero ($p > 0.05$), we do not report the coefficient in our model summary.

Table 3: Description of the control (a), code review coverage (b), and code review participation (c) metrics.

(a) Control Metrics

Metric	Description	Rationale
Forks	Number of repository forks	The more forks a repository has, the more users are contributing pull requests to the project.
Watchers	Number of repository watchers	Watchers are users who get notifications about activity on a project. The more watchers a repository has, the more active eyes and contributors it likely has.
Stars	Number of repository stars	On GitHub, users interested in a project can "star" the repository, making the number of stars a good proxy for the popularity of a project. More popular projects, with more users, will tend to have more bug reports and more active development.
Size	Size of repository (in bytes)	Larger projects have more code. Larger code bases have a greater attack surface, and more places in which defects can occur.
Churn	Sum of added and removed lines of code among all merged pull requests	Code churn has been associated with defects [34, 35].
Age	Age of repository (seconds)	The difference (in seconds) between the time the repository was created and the time of the latest commit to the repository. Ozment and Schechter [38] found evidence that the number of foundational vulnerabilities reported in OpenBSD decreased as a project aged, but new vulnerabilities are reported as new code is added.
Pull Requests	Total number of pull requests in a project	The number of pull requests is used as a proxy for the churn in the code base, which has been associated with both software quality [33, 34] and software security [43].
Memory-Safety	Whether all three of the top languages for a project are memory-safe	Software written in non-memory-safe languages (e.g., C, C++, Objective-C) are vulnerable to entire classes of security bugs (e.g., buffer-overflow, use-after-free, etc.) that software written in memory-safe languages are not [44]. Therefore, we might expect that such software would inherently have more security bugs.
Contributors	Number of authors that have committed to a project	The number of contributors to a project can increase the heterogeneity of the code base, but can also increase the number and quality of code reviews and architectural decisions.

(b) Coverage Metrics (RQ 1, 2)

Metric	Description	Rationale
Unreviewed Pull Requests	The number of pull requests in a project that were merged without any code review	A pull request merged by the same author who created it, without any discussion, implies that the changes have not been code reviewed. Such changes may be more likely to result in both general defects [28] and security bugs [30].
Unreviewed Churn	The total churn in a project from pull requests that were merged without any code review	While churn may induce defects in software, code review may help prevent some defects introduced by churn. We would expect that the lower the amount of unreviewed churn, the lower the number of defects introduced.

(c) Participation Metrics (RQ 3, 4)

Metric	Description	Rationale
Average Commenters	Mean number of commenters on pull requests in a project	Prior work has shown that too many distinct commenters on change requests can actually have a negative impact on software quality [30].
Mean Discussion Comments	Mean number of general discussion comments on pull requests in a project	We expect that increased discussion on a pull request may be indicative of more thorough code review.
Mean Review Comments	Mean number of comments on specific lines of code in pull requests in a project	We expect that more review comments mean more specific changes are being requested during code review, which may be indicative of more thorough code review.

We report effect sizes as the regression model coefficients, as they are more readily interpretable than abstract effect size measures.

5 RESULTS

5.1 RQ1: Is there a relationship between code review coverage and the number of issues in a project?

Table 4: Review coverage and overall issues model.

Adjusted R^2	0.5459	
F(8,3117)	470.6^{***}	
log security issues	Coef.	Std. Err.
(Intercept)		⋄
log forks		†
log watchers		†
log size	0.1972	$(0.0087)^{***}$
log churn	-0.0209	$(0.0089)^{*}$
log age	0.0464	$(0.0198)^{*}$
log contributors		⋄
log stars	0.2065	$(0.0105)^{***}$
log pull requests	0.2822	$(0.0238)^{***}$
memory safety	0.2306	$(0.0435)^{***}$
log unreviewed pull requests	0.0880	$(0.0169)^{***}$
log unreviewed churn		†

† Discarded during correlation analysis ($|\rho| > 0.7$)
⋄ $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

Prior work has found significant effects between code review coverage and defects in an analysis of three large software projects [28]. To investigate this relationship on a more general sample, we build a model using the log number of overall issues in a repository as the response variable, and the log number of unreviewed (but integrated) pull requests as the explanatory variable. The model is presented in Table 4.

The amount of unreviewed churn is too highly correlated with the number of unreviewed pull requests to include in the model. We chose to keep the number of unreviewed pull requests as it is a simpler metric, and we argue is easier to reason about as part of an operational code review process. For completeness, we analyzed the model that used the amount of unreviewed churn instead and found that it had no noticeable effect on model performance. The same was true for RQ2.

We find a small but significant positive relationship between the log number of unreviewed pull requests in a project and the log number of issues the project has. Projects with more unreviewed pull requests tend to have more issues. Holding other variables constant, with a 1% decrease in the number of unreviewed pull requests we would expect to see a 0.08% decrease in the number of issues. Halving the number of unreviewed pull requests we would expect to see 5% fewer issues.

5.2 RQ2: Is there a relationship between code review coverage and the number of security bugs reported to a project?

Table 5: Review coverage and security issues model.

Adjusted R^2	0.3663	
F(8,3117)	226.8^{***}	
log security issues	Coef.	Std. Err.
(Intercept)	-2.1603	$(0.3555)^{***}$
log forks		†
log watchers		†
log size	0.1691	$(0.0086)^{***}$
log churn		⋄
log age	0.0396	$(0.0198)^{*}$
log contributors		⋄
log stars	0.0883	$(0.0103)^{***}$
log pull requests	0.2046	$(0.0234)^{***}$
memory safety	0.2322	$(0.0428)^{***}$
log unreviewed pull requests	0.0957	$(0.0167)^{***}$
log unreviewed churn		†

† Discarded during correlation analysis ($|\rho| > 0.7$)
⋄ $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

To explore this question, we replace the response variable of our previous model with the log number of security bugs (as predicted by our quantifier for each project). The model is presented in Table 5.

We find a small but significant positive relationship between the log number of integrated pull requests that are unreviewed and the log number of security bugs a project has. Projects with more unreviewed pull requests tend to have a greater number of security bugs, when controlling for the total numbers of pull requests and issues. Holding other variables constant, with a 1% decrease in the number of unreviewed pull requests we would expect to see a 0.09% decrease in the number of security bugs. Halving the number of unreviewed pull requests we would expect to see 6% fewer security bugs.

5.3 RQ3: Is there a relationship between code review participation and the number of issues in a project?

To explore this question, we alter our model to use a response variable of the log number of issues in a project, and we replace our main explanatory variable with the log mean number of commenters on pull requests and the log mean number of review comments per pull request in each project. The model is presented in Table 6. As

Table 6: Review participation and overall issues model.

Adjusted R^2	0.543	
F(9,3116)	413***	
log issues	Coef.	Std. Err.
(Intercept)		⋄
log forks		†
log watchers		†
log size	0.2036	(0.0090)***
log churn		⋄
log age		⋄
log contributors		⋄
log stars	0.1881	(0.0107)***
log pull requests	0.3471	(0.0212)***
memory safety	0.2436	(0.0436)***
log mean commenters per pr		⋄
log mean review comments per pr	-0.1105	(0.0477)*
log mean discussion comments per pr		†

† Discarded during correlation analysis ($|\rho| > 0.7$)
⋄ $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

Table 7: Review participation and security issues model.

Adjusted R^2	0.36	
F(9,3116)	196.3***	
log security bugs	Coef.	Std. Err.
(Intercept)	-2.0655	(0.3569)***
log forks		†
log watchers		†
log size	0.1704	(0.0088)***
log churn		⋄
log age		⋄
log contributors		⋄
log stars	0.0797	(0.0105)***
log pull requests	0.2718	(0.0209)***
memory safety	0.2416	(0.0429)***
log mean commenters per pr		⋄
log mean review comments per pr		⋄
lot mean discussion comments per pr		†

† Discarded during correlation analysis ($|\rho| > 0.7$)
⋄ $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

noted in Table 6, we did not include the mean number of discussion comments in the model as it was highly correlated with the mean number of commenters. We chose to keep the mean number of commenters and the mean number of review comments, to capture both aspects of participation.

We do not find a significant relationship between the average number of commenters on pull requests and the total number of issues. However, we did find a small but significant negative relationship between the log mean number of review comments per pull request and the log number of issues a project has. Projects that, on average, have more review comments per pull request tend to have fewer issues. Holding other variables constant, with a 1% increase in the average number of review comments per pull request we would expect to see a 0.11% decrease in the number of issues. Doubling the average number of review comments per pull request we would expect to see 5.5% fewer issues.

5.4 RQ4: Is there a relationship between code review participation and the number of security bugs reported to a project?

To explore this question, we change the response variable in our previous model to be the log number of security bugs reported in a project. The model is presented in Table 7.

We do not find a significant relationship between the average number of commenters on pull requests and the number of security bugs. This result is in contrast with that found by Meneely et al. [30]. While they found that vulnerable files in the Chromium project tended to have more reviewers per SLOC and more reviewers per review, we were unable to replicate the effect. (We note that we looked for an effect across projects, taking a single average for each project, instead of across files within a single project.)

We also did not find a significant relationship between the average number of review comments per pull request and the number of security bugs reported.

6 THREATS TO VALIDITY

6.1 Construct validity

In order to handle the scale of our sample, we used a machine-learning based quantifier to estimate the dependent variable in our security models (the number of security bugs reported in a project). A quantifier with low precision (high variance in its error) or one that was skewed to over-predict higher proportions could cause spurious effects in our analysis. We tested our quantifiers on a sample of real issues from GitHub repositories and selected a quantifier model that has good accuracy and high precision (low variance) in its estimations across a wide range of proportions.

This also means that the dependent variable includes some noise (due to quantifier error). We do not expect errors in quantification to be biased in a way that is correlated to code review practices. Linear regression models are able to tolerate this kind of noise. Statistical hypothesis testing takes this noise into account; the association we found is significant at the $p < 0.001$ level (Table 5).

We manually label issues as "security bugs" to train our quantifier. We have a specific notion of what a security bug is (see Section 2), but we have weak ground truth. We used one coder, and there is some grey area in our definition. Our use of quantification should

mitigate this somewhat (particularly if the grey area issues are equally likely to be false positives as false negatives, and thus cancel out in the aggregate).

It could be possible to test the predictive validity of our operationalization (the quantifier) on a real-world dataset with known ground truth (such as the Chromium issue tracker [11]).

Ideally, we would prefer to be able to measure the security of a piece of software directly, but this is likely impossible. Security metrics are still an area of active research. In this study we use the number of security issues as an indirect measure of the security of a project, rather than trying to directly assess the security of the software itself. This limits the conclusions that can be drawn from our results, as we cannot directly measure and analyze the security of the projects in our dataset.

Our main metric of review coverage (whether a pull request has had any participation from a second party) is somewhat simplistic. One concern is if open source projects tend to "rubber stamp" pull requests (a second participant merges or signs off on a pull request without actually reviewing it): our metric would count this as code review, while it should not be counted. Our metric is an upper bound on code review coverage.

Some of our control explanatory variables are proxies for the underlying constructs we are trying to control for. These proxies may be incomplete in capturing the underlying constructs.

6.2 External validity

We intentionally chose a broad population of GitHub repositories in order to try to generalize prior case study-based research on code review and software quality. Our population includes many small or inactive repositories, so our sample may not be representative of security critical software or very popular software. Looking at top GitHub projects might be enlightening, but would limit the generalizability of the results, and might limit the ability to gather a large enough sample.

While GitHub is the largest online software repository hosting service, there may be a bias in open source projects hosted on GitHub, making our sample not truly representative of open source software projects. One concern is that many security critical or very large projects are not on GitHub, or only mirrored there (and their issue tracking and change requests happen elsewhere). For example, the Chromium and Firefox browsers, the WebKit rendering engine, the Apache Foundation projects, and many other large projects fall into this category. Additionally, sampling from GitHub limits us to open source software projects. Commercial or closed source projects may exhibit different characteristics.

6.3 Effect of Choice of Quantifier

Prior work in defect prediction has found that the choice of machine learning model can have a significant effect on the results of defect prediction studies [7, 19]. We repeated our regression analysis using the naive classify-and-count technique with a random forest classifier model ("RF CC"). This was the best performing of our non-neural network quantification models (see "RF CC" in Figure 1). The regression models produced using the predictions from this quantifier had the same conclusions as our results in Section 5, but with smaller effect sizes on the explanatory variables, and some differences in the effects of the controls. This is likely due to the fact that the RF CC model tends to under-predict the number of security issues compared to our chosen neural network model.

7 RELATED WORK

Heitzenrater and Simpson [22] call for the development of an economics of secure software development, give an overview of how information security economics can lay the foundations, and put forth a detailed research agenda to allow for reasoned investment decisions into software security outcomes. We believe this paper helps to elucidate a small piece of the secure software development economics problem.

Edmundson et al. [12] examined the effects of manual code inspection on a piece of web software with known and injected vulnerabilities. They found that no reviewer was able to find all of vulnerabilities, that experience didn't necessarily reflect accuracy or effectiveness (the effects were not statistically significant), and that false positives were correlated with true positives ($r = 0.39$). It seems difficult to predict the effectiveness of targeted code inspection for finding vulnerabilities.

McIntosh et al. [28] studied the connection between code review coverage and participation and software quality in a case study of the Qt, VTK, and ITK projects. For general defects, they found that both review coverage and review participation are negatively associated with post-release defects.

Meneely et al. [30] analyzed the socio-technical aspects of code review and security vulnerabilities in the Chromium project (looking at a single release). They measured both the thoroughness of reviews of changes to files, and socio-technical familiarity—whether the reviewers had prior experience on fixes to vulnerabilities and how familiar the reviewers and owners are with each other. They performed an association analysis among all these metrics, and found that vulnerable files tended to have many more reviews. In contrast to the results of McIntosh et al., vulnerable files also had more reviewers and participants, which may be evidence of a "bystander apathy" effect. These files also had fewer security-experienced participants.

Ray et al. [40] looked at the effects of programming languages on software quality. They counted defects by detecting "bug fix commits"—commits that fix a defect, found by matching error-related keywords. They found that some languages have a greater association with defects than other languages, but the effect is small. They also found that language has a greater impact on specific categories of defects than it does on defects in general.

8 CONCLUSIONS

We have presented the results of a large-scale study of code review coverage and participation as they relate to software quality and software security. Our results indicate that code review coverage has a small but significant effect on both the total number of issues a project has and the number of security bugs. Additionally, our results indicate that code review participation has a small but significant effect on the total number of issues a project has, but it does not appear to have an effect on the number of security bugs. Overall, code review appears to reduce the number of bugs and number of security bugs.

These findings partially validate the prior case study work of McIntosh et al. [28] and Meneely et al. [30]. However, we did not replicate Meneely's finding of increased review participation having a positive relationship with vulnerabilities. More work would be required to determine if this is a difference in our metrics or a difference in the populations we study. Our results suggest that implementing code review policies within the pull request model of development may have a positive effect on the quality and security of software. However, our analysis only shows correlation—further work is needed to show if there is a causative effect of code review on quality and security.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). Software available from tensorflow.org.

[2] Aybuke Aurum, Håkan Petersson, and Claes Wohlin. 2002. State-of-the-art: software inspections after 25 years. *Software Testing, Verification and Reliability* 12, 3 (2002). https://doi.org/10.1002/stvr.243

[3] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *International Conference on Software Engineering*.

[4] Jose Barranquero, Jorge Díez, and Juan José del Coz. 2015. Quantification-oriented learning based on reliable classifiers. *Pattern Recognition* 48, 2 (2015). https://doi.org/10.1016/j.patcog.2014.07.032

[5] Antonio Bella, Cesar Ferri, Jose Hernandez-Orallo, and Maria Jose Ramirez-Quintana. 2010. Quantification via probability estimators. In *IEEE International Conference on Data Mining*.

[6] M. Bernhart and T. Grechenig. 2013. On the understanding of programs with continuous code reviews. In *International Conference on Program Comprehension*. https://doi.org/10.1109/ICPC.2013.6613847

[7] David Bowes, Tracy Hall, and Jean Petrić. 2017. Software defect prediction: do different classifiers find the same defects? *Software Quality Journal* (2017). https://doi.org/10.1007/s11219-016-9353-3

[8] F. Camilo, A. Meneely, and M. Nagappan. 2015. Do Bugs Foreshadow Vulnerabilities? A Study of the Chromium Project. In *Mining Software Repositories*. https://doi.org/10.1109/MSR.2015.32

[9] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. *CoRR* abs/1603.02754 (2016). http://arxiv.org/abs/1603.02754

[10] Steve Christey. 2011. CWE/SANS Top 25 Most Dangerous Software Errors. (2011). https://cwe.mitre.org/top25/

[11] Chromium Project. 2017. Chromium issue tracker. (2017). https://bugs.chromium.org/p/chromium

[12] Anne Edmundson, Brian Holtkamp, Emanuel Rivera, Matthew Finifter, Adrian Mettler, and David Wagner. 2013. An Empirical Study on the Effectiveness of Security Code Review. In *International Symposium on Engineering Secure Software and Systems*. 197–212. https://doi.org/10.1007/978-3-642-36563-8_14

[13] Andrea Esuli and Fabrizio Sebastiani. 2015. Optimizing text quantifiers for multivariate loss functions. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 9, 4 (2015).

[14] George Forman. 2005. Counting positives accurately despite inaccurate classification. In *European Conference on Machine Learning*.

[15] John Fox. 2008. *Applied regression analysis and generalized linear models* (2nd ed.). Sage.

[16] Wei Gao and Fabrizio Sebastiani. 2015. Tweet sentiment: from classification to quantification. In *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*.

[17] M. Gegick, P. Rotella, and L. Williams. 2009. Predicting Attack-prone Components. In *International Conference on Software Testing Verification and Validation*. https://doi.org/10.1109/ICST.2009.36

[18] M. Gegick, P. Rotella, and T. Xie. 2010. Identifying security bug reports via text mining: An industrial case study. In *Mining Software Repositories*. https://doi.org/10.1109/MSR.2010.5463340

[19] Baljinder Ghotra, Shane McIntosh, and Ahmed E. Hassan. 2015. Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models. In *International Conference on Software Engineering*. 12.

[20] GitHub. 2017. GitHub API. (2017). https://developer.github.com/v3/

[21] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. 2012. A Systematic Literature Review on Fault Prediction Performance in Software Engineering. *IEEE Transactions on Software Engineering* 38, 6 (Nov 2012). https://doi.org/10.1109/TSE.2011.103

[22] Chad Heitzenrater and Andrew Simpson. 2016. A Case for the Economics of Secure Software Development. In *New Security Paradigms Workshop (NSPW '16)*. https://doi.org/10.1145/3011883.3011884

[23] Daniel J Hopkins and Gary King. 2010. A method of automated nonparametric content analysis for social science. *American Journal of Political Science* 54, 1 (2010).

[24] Ilya Grigorik. 2017. GitHub Archive. (2017). https://www.githubarchive.org/

[25] Thorsten Joachims, Thomas Finley, and Chun-Nam John Yu. 2009. Cutting-plane training of structural SVMs. *Machine Learning* 77, 1 (2009).

[26] Thorsten Joachims, Thomas Hofmann, Yisong Yue, and Chun-Nam Yu. 2009. Predicting structured objects with support vector machines. *Commun. ACM* 52, 11 (2009).

[27] Edward Loper and Steven Bird. 2002. NLTK: The Natural Language Toolkit. In *Proceedings of the Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics*. https://doi.org/10.3115/1118108.1118117 Software available from nltk.org.

[28] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2014. The impact of code review coverage and code review participation on software quality: a case study of the Qt, VTK, and ITK projects.. In *Mining Software Repositories*. https://doi.org/10.1145/2597073.2597076

[29] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates. 2013. When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 65–74. https://doi.org/10.1109/ESEM.2013.19

[30] A Meneely, ACR Tejeda, B Spates, and S Trudeau. 2014. An empirical investigation of socio-technical code review metrics and security vulnerabilities. In *International Workshop on Social Software Engineering*. https://doi.org/10.1145/2661685.2661687

[31] Lawrence S. Meyers, Glenn Gamst, and Anthony J. Guarino. 2006. *Applied Multivariate Research: Design and Interpretation* (1st ed.). Sage.

[32] L. Milli, A. Monreale, G. Rossetti, F. Giannotti, D. Pedreschi, and F. Sebastiani. 2013. Quantification Trees. In *IEEE International Conference on Data Mining*. https://doi.org/10.1109/ICDM.2013.122

[33] J. C. Munson and S. G. Elbaum. 1998. Code churn: a measure for estimating the impact of code change. In *International Conference on Software Maintenance*. https://doi.org/10.1109/ICSM.1998.738486

[34] N. Nagappan and T. Ball. 2005. Use of relative code churn measures to predict system defect density. In *International Conference on Software Engineering*. https://doi.org/10.1109/ICSE.2005.1553571

[35] Nachiappan Nagappan and Thomas Ball. 2007. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. In *International Symposium on Empirical Software Engineering and Measurement*. https://doi.org/10.1109/ESEM.2007.13

[36] Preslav Nakov, Alan Ritter, Sara Rosenthal, Veselin Stoyanov, and Fabrizio Sebastiani. 2016. SemEval-2016 Task 4: Sentiment Analysis in Twitter. In *Proceedings of the 10th International Workshop on Semantic Evaluation*.

[37] Robert M. O'brien. 2007. A Caution Regarding Rules of Thumb for Variance Inflation Factors. *Quality & Quantity* 41, 5 (2007). https://doi.org/10.1007/s11135-006-9018-6

[38] Andy Ozment and Stuart E Schechter. 2006. Milk or Wine: Does software security improve with age?. In *USENIX Security*.

[39] Princeton University. 2010. About WordNet. (2010). http://wordnet.princeton.edu

[40] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in GitHub. In *International Symposium on Foundations of Software Engineering*. https://doi.org/10.1145/2635868.2635922

[41] Peter Rigby, Brendan Cleary, Frederic Painchaud, Margaret-Anne Storey, and Daniel German. 2012. Contemporary peer review in action: Lessons from open source development. *IEEE Software* 29, 6 (2012).

[42] Gerard Salton and Christopher Buckley. 1988. Term-weighting approaches in automatic text retrieval. *Information Processing & Management* 24, 5 (1988).

[43] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. 2011. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Transactions on Software Engineering* 37, 6 (Nov 2011). https://doi.org/10.1109/TSE.2010.81

[44] L. Szekeres, M. Payer, T. Wei, and D. Song. 2013. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP.2013.13

[45] P Thongtanunam and S McIntosh. 2015. Investigating code review practices in defective files: an empirical study of the Qt system. In *Mining Software Repositories*. https://doi.org/10.1109/msr.2015.23

[46] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. 2011. Security Versus Performance Bugs: A Case Study on Firefox. In *Mining Software Repositories*. 10. https://doi.org/10.1145/1985441.1985457