Large-Scale Analysis of Modern Code Review Practices and Software Security in Open Source Software

Christopher Thompson

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2017-217
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-217.html

December 14, 2017

Acknowledgement

I want to thank the people who have supported and guided me along my path to eventually finishing this dissertation. My parents, who encouraged me to explore my various wild fascinations while I was growing up, and who supported me through my undergraduate studies. Nick Hopper, who mentored me throughout my undergraduate studies, and who gave me a chance to learn my love of computer security and research. David Wagner, my doctoral advisor, who has been a constant sounding board for my ideas (including what led to this dissertation), for always having something interesting to teach me, and for always wanting to learn from me as I explored new ideas.

And finally, thank you to Hina, who has supported me through graduate school with love and lots of understanding.

Large-Scale Analysis of Modern Code Review Practices and Software Security in Open Source Software

by

Christopher Thompson

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor David Wagner, Chair
Professor Vern Paxson
Associate Professor Coye Cheshire

Fall 2017

Large-Scale Analysis of Modern Code Review Practices and Software Security in Open Source Software

Abstract

Large-Scale Analysis of Modern Code Review Practices and Software Security in Open Source Software

by

Christopher Thompson

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David Wagner, Chair

Modern code review is a lightweight and informal process for integrating changes into a software project, popularized by GitHub and pull requests. However, having a rich empirical understanding of modern code review and its effects on software quality and security can help development teams make intelligent, informed decisions, analyzing the costs and the benefits of implementing code review for their projects, and provide insight on how to support and improve its use.

This dissertation presents the results of our analyses on the relationships between modern code review practice and software quality and security, across a large population of open source software projects. First, we describe our neural network-based quantification model which allows us to efficiently estimate the number of security bugs reported to a software project. Our model builds on prior quantification-optimized models with a novel regularization technique we call random proportion batching. We use our quantification model to perform association analysis of very large samples of code review data, confirming and generalizing prior work on the relationship between code review and software security and quality. We then leverage timeseries changepoint detection techniques to mine for repositories that have implemented code review in the middle of their development. We use this dataset to explore the causal treatment effect of implementing code review on software quality and security. We find that implementing code review may significantly reduce security issues for projects that are already prone to them, but may significantly increase overall issues filed against projects. Finally, we expand our changepoint detection to find and analyze the effect of using automated code review services, finding that their use may significantly decrease issues reported to a project. These findings give evidence for modern code review being an effective tool for improving software quality and security. They also suggest that the development of better tools supporting code review, particularly for software security, could magnify this benefit while decreasing the cost of integrating code review into a team's development process.

Contents

List of Figures

List of Tables

Acknowledgments

I want to thank the people who have supported and guided me along my path to eventually finishing this dissertation. My parents, who encouraged me to explore my various wild fascinations while I was growing up, and who supported me through my undergraduate studies. Nick Hopper, who mentored me throughout my undergraduate studies, and who gave me a chance to learn my love of computer security and research. David Wagner, my doctoral advisor, who has been a constant sounding board for my ideas (including what led to this dissertation), for always having something interesting to teach me, and for always wanting to learn from me as I explored new ideas.

And finally, thank you to Hina, who has supported me through graduate school with love and lots of understanding.

Chapter 1

Introduction

Code inspections are costly and formal, and frequently not performed as often as they maybe should be. In comparison, modern code review is lightweight, informal, and frequently integrated into the tools already used by developers. In modern code review, reviewers leave comments (in the form of feedback, discussion, or suggested changes) on changes *before* they are integrated into the codebase. The goals of modern code review are to ensure that the code is free of defects, that it follows team conventions, that it solves problems in a reasonable way, and is generally of high quality.

All of this makes modern code review a particularly interesting software engineering process to study. There has been much research into code review process and how it relates to software quality in general [2, 3, 42, 65], 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 [10, 43, 68].

Heitzenrater and Simpson [30] called for the development of a "secure software development economics" to provide a foundation for reasoning about investments into software development processes for software security. As Heitzenrater and Simpson write, "Management is often reluctant to make investments in security processes or procedures without a business case demonstrating a tangible return on investment." Having a rich theoretical and empirical understanding of software process and its effects on software quality and security can help development teams make intelligent, informed decisions trading off the costs and the benefits of different practices. As part of this decision, teams must balance security and functionality expenditure. However, modern code review potentially benefits both, while being a lightweight (and lower cost) process compared to formal code inspections. Furthermore, costs to deal with errors increase as development continues [30]. Modern code review can catch issues *before* changes are integrated.

Czerwonka, Greiler, and Tilford [15] argue that code review, as it is currently implemented, can lack usefulness and be an expensive commitment. Their (admittedly exaggerated) title, "Code Reviews Do Not Find Bugs: How the Current Code Review

Practice Slows Us Down", belies a more complex complaint: Based on data within Microsoft, they find that code review can have outsized and often ignored costs, while the effects are less well understood than we might think. For example, code reviews often fail to find any issues that require blocking the change from being integrated, and current practices may not be optimally applied (particularly in terms of review priority and reviewer assignment).

Modern code review can be considered to be expensive, disruptive, or slow. It is often viewed as an obstacle to productivity, at best grudgingly performed. Code review needs more research (what do we know about it) and development (how can we make better tools to help with it). Prioritizing important review (by component or type of change, for example), improving usefulness of review with better tools, and automating the routine review tasks (e.g., style issues) to improve the review process can all help reduce the perceived costs of code review. But development teams can only adequately consider the costs if they also understand the benefits.

Our goal is to better understand how code review coverage and participation affect security. Empirical evidence for the relationship between code review process and software security (and software quality) could help improve code review automation and tools or encourage the adoption of code review process. Prior work in this area has primarily been limited to case studies of a small handful of software projects [42, 44, 65]. To the best of our knowledge, we present the first large-scale analyses of the relationship between code review and software security in open source projects.

Modern code review can be one part of a broader set of secure software development processes, but we need measures of the effectiveness of code review to be able to evaluate investments into code review processes. In this dissertation, we focus on more precisely understanding the effects of code review practice, as commonly implemented in open source software projects using pull requests. We provide evidence that code review coverage is positively correlated with software quality and software security. Furthermore, we show that by implementing thorough code review practices, some open source software projects may improve the security of their software. We believe this evidence adds credence to the usefulness of modern code review practices. Finally, we show that automated code review may greatly improve software quality, motivating further development of tools supporting code review.

The remainder of this thesis is organized as follows. We begin with an overview of the modern code review process in Chapter 2, covering the state of research into code review and research related to this thesis. Chapter 3 presents an overview of *quantification* models, and our neural-network based quantifier for estimating the number of security issues reported to a project. Chapter 4 presents our correlational study of code review practices and software quality and security. Chapter 5 presents our changepoint-based treatment detection technique, and our quasi-experimental study of the causal effects of code review practices on software quality and security. Finally, in Chapter 5.6 we draw conclusions, summarize the main contributions of this thesis, and discuss potential directions for future research.

We have made our datasets, analysis scripts, and software are available online [63].

Thesis Contributions

We show that:

- Quantification models can be a powerful tool for estimating quantities of interest for large-scale studies where hand-labeling would be impossible. Our quantification-optimized neural network can estimate the fraction of security issues in a set of issues with only 4% error. (Chapter 3)
- Projects with fewer merged pull requests that are unreviewed (merged by their creator without any comments) tend to have fewer overall issues filed against them, as well as fewer security bugs reported in their issue trackers. (Chapter 4)
- Projects that on average have more review comments per pull request (comments on specific lines of code) tend to have fewer overall issues filed against them, but we found no relationship with security bugs. (Chapter 4)
- Combining timeseries changepoint analysis and quasi-experimental design methodology allows us to test the causal relationship between code review practices and software quality and security. (Chapter 5)
- The implementation of modern code review may reduce security issues for projects that are prone to them, but can increase the number of overall issues reported to projects. (Chapter 5)
- The use of automated code review services can have a stronger effect than peer code review for reducing overall issues. It may be particularly effective for projects with little existing code review practices. However, we found no such effect on security bugs. (Chapter 5)

Chapter 2

Background

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 [56].

Modern code review is also often about more than finding defects. 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 [7].

GitHub,[1] the largest online repository hosting service, encourages modern code review through its pull request system. For open source projects, a pull request (often shortened to "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"). They also assist with code review by showing diffs of the changes made, and integrate with a host of third-party code analysis and testing tools. Figure 2.1 shows an example of a pull request on a GitHub repository.

An example of a pull request-based review process is to require that all commits go through pull requests and have a minimum amount of review. For example, a project might require any change be approved by at least one reviewer who is knowledgeable about the component being modified before that change is merged into the main branch of the repository. (This is similar to the policy used by the Chromium browser project [13].)

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 a metric for what proportion of changes are reviewed before being integrated

[1] `https://github.com`

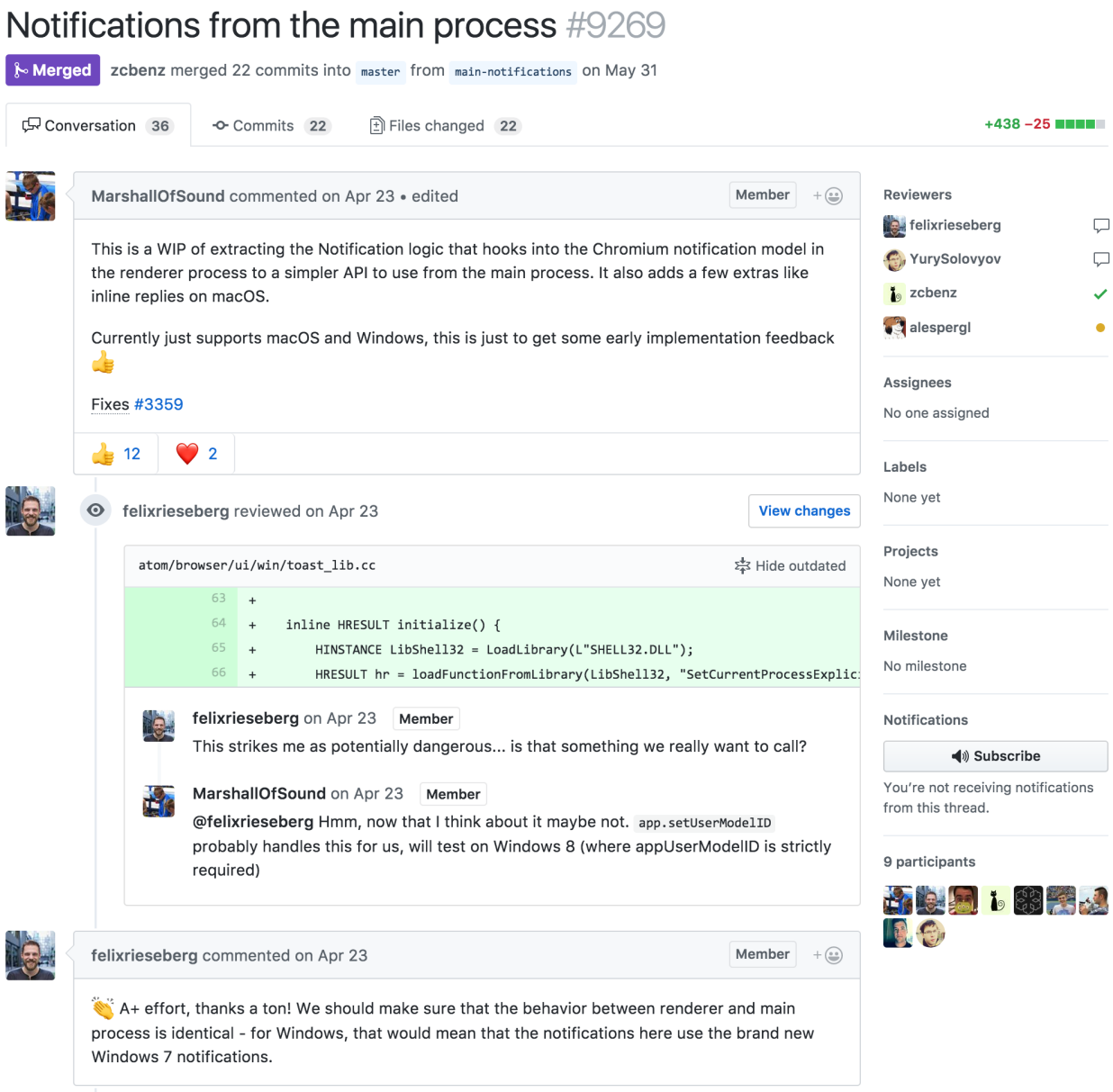

Figure 2.1: A pull request on GitHub. Pull requests provide a single place for discussion about a set of proposed changes, and for comments on specific parts of the code itself. Here, we see a user contributing a new feature for Electron (`https://github.com/electron/electron`). They summarize their changes and refer to a prior issue. Their commits are listed, and a reviewer can load them to view a diff of all of the changes made. Below, a reviewer has written an inline code review comment regarding a snippet of code that is potentially unsafe. This project requires approval from specific reviewers before a pull request can be merged.

into the code base for a project. Previous case studies have examined the effects of code review coverage on software quality [42, 65] and software vulnerabilities [44] 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.

2.1 Related Research

The nature of vulnerabilities

Software security is a form of software quality, but there is reason to believe that software vulnerabilities may be different from general software defects. Camilo et al. [10] examined 374,686 bugs and 703 post-release vulnerabilities in the Chromium project, and found that vulnerabilities tend to be distributed differently throughout software than general defects (that is, files with the highest defect density did not intersect with those with the highest vulnerability density). Meneely et al. [43] examined 68 vulnerabilities in the Apache HTTP Server project and traced them back to the commits that originally contributed the vulnerable code. They found that vulnerabilities were connected to changes with higher churn (more lines of code added and removed; vulnerability-contributing commits had more than twice as much churn on average) and newer developers than those connected to general defects. Zaman et al. [68] did a case study of the Firefox project and found that vulnerabilities tended to be fixed faster than general defects, but their bug reports tended to be reopened more often. Ozment and Schechter [51] 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.

Edmundson et al. [16] 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 the vulnerabilities, that experience did not 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.

Code review and software quality and security

McIntosh et al. [42] studied the connection between code review coverage and participation and software quality in a case study of the Qt, VTK, and ITK projects, which use the Gerrit code review tool.[2] They used multiple linear regression models to explain the relationship between the incidence of post-release defects (defects in official releases of software) and code review coverage (the proportion of self-reviewed changes,

[2] https://www.gerritcodereview.com/

the proportion of hastily reviewed changes, and the proportion of changes without discussion). They used keyword search on the commit messages to determine whether each change fixes a defect (a "bug-fixing commit"). These keywords included words like "bug", "fix", "defect", or "patch". For general defects, they found that both review coverage and review participation are negatively associated with post-release defects.

Meneely et al. [44] analyzed the socio-technical aspects of code review and security vulnerabilities in the Chromium project (looking at a single release). They labeled each source code file as "vulnerable" if it was fixed for a specific CVE[3] after the release, or "neutral" if not. They measured both the thoroughness of reviews of changes to files (the total number of reviews, reviewers, and participants a file had; the average number of reviewers per code review who did not participate; the percentage of reviews with three or more reviewers; and the percentage of reviews that exceeded 200 lines per hour, a threshold for "fast" reviews [36]), and socio-technical familiarity (whether the reviewers had prior experience on fixes to vulnerabilities and how familiar the reviewers and owners were 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. They conclude that security code review is much more nuanced than the "many eyes make all bugs shallow" argument—a diversity of eyes may play a bigger role.

In a similar vein, Rahman et al. [54] found that reviewer expertise was a promising metric to use for recommending useful reviewers for pull requests in GitHub. They proposed a system that leveraged relevant cross-project work history (prior review experience on changes involving the same technologies or external libraries). They evaluated their recommendation tool with a study of 10 commercial projects (with 13,081 pull requests) and 6 open source projects (with 4,034 pull requests), and found that their technique provides 85-92% recommendation accuracy.

Bosu et al. [8] studied factors that improve the quality and usefulness of code reviews. They began by performing qualitative interviews with developers at Microsoft. Their interviews yielded 145 comments rated on their usefulness, which they supplemented by manually analyzing another 844 review comments. They used this dataset to develop an automated classifier that can distinguish between useful and not useful review comments, and then applied their classifier to over a million review comments on five Microsoft projects. They found that reviewer experience and the size of the changesets have an effect on the usefulness of reviews.

Vasilescu et al. [67] investigated the use of continuous integration (CI) services in GitHub repositories. A common use of continuous integration is to automate running the test suite for a piece of software on every commit before integration. On GitHub,

[3]A CVE (Common Vulnerabilities and Exposures) identifier, sometimes simply referred to as "a CVE", is a unique reference to a publicly known security vulnerability.

there are continuous integration services that integrate directly into the pull request interface, showing whether the proposed changes in a pull request can be cleanly merged into the project. Vasilescu et al. looked at a sample of 223 repositories that used the popular Travis-CI service. They selected repositories that were not forks; had not been deleted; were at least one year old; received both direct commits and pull requests; were written in Java, Python, or Ruby; had at least 10 changes (commits or pull requests) during the last month; and had at least 10 contributors. This included several popular open source projects (such as rails and elasticsearch). They found that direct code modifications (those pushed directly to the main repository) were more common than pull requests, and, perhaps surprisingly, were more likely to result in successful builds than pull requests. However, they did not attempt to analyze why this might be the case, or to control for other potentially confounding factors. Continuous integration is promising as a tool for automating many code review tasks, improving the efficacy and consistency of code review while reducing its overall costs. In Chapter 5 we look at the use of automated code review services, one form of continuous integration tooling, to see if they have an effect on software quality and security.

Programming languages and software quality and security

Ray et al. [55] looked at the effects of programming languages on software quality. For the top 19 programming languages on GitHub, they took the top 50 repositories for each (by number of "stars"), and then filtered out the bottom quartile by number of commits, giving them a dataset of 729 repositories. They categorized languages into different classes by programming paradigm (*Procedural*, *Scripting*, or *Functional*), compilation class (*Static* or *Dynamic*), type class (*Strong* or *Weak*), and memory class (*Managed* or *Unmanaged*). They counted defects by detecting "bug-fix commits"—commits that fix a defect, found by matching error-related keywords, which included "bug", "fix", "issue", "mistake", "incorrect", "fault", "defect", and "flaw". They categorized bugs into seven categories based on their cause and impact. To do this, they first randomly chose 10% of the bug-fix commits and used keyword search to categorize them with potential bug types. Then, they used those categorized commits to train an SVM-based classifier using the bag-of-words features for each commit message. They manually annotated 180 randomly chosen bug fixes, equally distributed across all of their bug categories. They had 75% precision and 83.33% recall for the "Security" category. Their best performance was for the "Concurrency" category, with 100% precision and 90.91% recall. They then used their classifier to predict the categories of their entire corpus of bug-fix commits, and used a negative binomial regression model to model the relationship between the number of bug-fix commits and the language categories of a project. They specifically chose negative binomial regression as it correctly models non-negative integer responses (in their case, the number of bug-fix commits) that may have overdispersion (where the response variance is greater than the mean). We use the same type of model for our analysis of the number of issues and security issues a project

has. 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. For example, procedural, static, weakly-typed, and unmanaged languages (such as C) had a high proportion of memory bugs.

Ruef et al. [58] created a "Build It Break It Fix It" contest design to gather evidence of how team composition, tools, methods, processes, and programming language choices affect the correctness, performance, and security of software. The contests used a fixed programming task, but were open ended in how teams approached and built the software. The second phase of the contests involved teams finding flaws (in correctness and security) in the software produced by other teams. The third phase allowed teams to submit patches to address each flaw found in their software. Ruef et al. designed the contest structure to try to provide incentives for each team to follow the tasks closely and in the expected manner—to maximize the correctness, performance, and security of the software they built. However, there was no assignment of techniques, languages, or process among the teams—they were allowed to freely choose how they developed their software (thus it was not a randomized experiment). They found that while teams using C and C++ had the most efficient software, they were much more likely to have security flaws (and almost all the increased security flaws were related to memory-safety issues). Similarly, memory-safe but statically-typed languages were less likely to have security flaws. Teams with diverse programming language knowledge created more secure code. They also found that breaking teams that were also successful building teams were better at finding security bugs.

Chapter 3

Quantification of Security Issues

An earlier version of the work in this chapter appears in the Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE) [64].

Ultimately, our studies involve statistical techniques to explore the relationship between code review and security using archival data. To measure security outcomes, 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.

The challenge is that this would seem to require examining each issue reported on the GitHub issue tracker for each project in our samples. For our association analysis in the next chapter on 2,881 repositories, we would have to examine 464,280 issues to determine whether each issue is security-related or not. It seems 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 [19] and has since been applied to a variety of fields, from sentiment analysis [20] to political science [31] to operations research [19]. 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 smaller than classification

error (the same models achieved around 8% average absolute quantification error—see "RF CC" in Figure 3.2). Intuitively, the false positives and the false negatives of the classifier can cancel each other out when the proportion is being calculated.[1]

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 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. For example, Gegick et al. [21] developed a predictive model to identify software components with the highest security risk. We refer the reader to Hall et al. [29], which reviewed 208 fault prediction studies from 2000-2010 and provides a good overview of this space. 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 [22], where their text-based bug report classifier was able to successfully classify 78% of security bug reports). 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

[1] In practice, this independence of errors may not hold, but many quantification techniques focus on adjusting for such bias.

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 [19]; 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)* [19], 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 [19] 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 $k = 10$ (a common default for cross-validation folds [38]). 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 [6]. It 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, 17, 46]. Forman was the first to use Kullback-Leibler Divergence (KLD), which measures the difference between two probability distributions, as a measure of quantification error [19]. 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.

In quantification, P represents the true class distribution (e.g., proportion of positive and negative instances) and $\hat{P}$ the predicted distribution (according to the output of the model). 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 [17] use structured prediction (based on SVM_{perf} [34, 35]) to train a SVM classifier that minimizes the KLD loss. They call their quantifier SVM(KLD), and it has been used for sentiment analysis tasks [20]. 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 [50].

3.3 *NNQuant* and Randomized Proportion Batching

Building on the idea of optimizing for quantification error instead of accuracy, we construct *NNQuant*, 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

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

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 random batches, using the gradient of the cross entropy loss between the predicted batch class distribution and the true batch class distribution. A reduced version of the network architecture is shown in Figure 3.1.

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 target 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

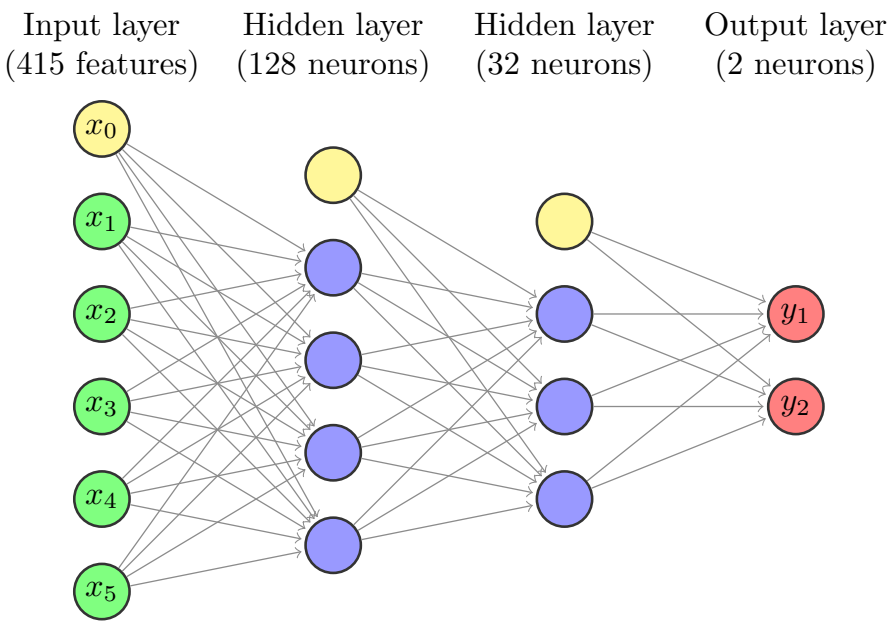

Figure 3.1: A reduced-size version of our neural network architecture, with fewer neurons at each layer but the same network design. Our neural network quantifier is a fully-connected feed forward network with two hidden layers of 128 and 32 neurons, with 415 input features and 2 output neurons. Each layer except the output layer includes a bias neuron (yellow), which is connected only to the neurons in the next layer. The hidden layers use the ReLU activation function. The output layer uses the softmax activation function so that the output is a probability distribution.

KLD, where we are averaging over a range of proportions p for the true proportion of positives. This training procedure acts as a form of regularization, forcing 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. Pseudocode for our training procedure is shown in Algorithm 1.

Our network architecture is kept intentionally simple, and as shown below, it performs very well. The number of hidden layers and their sizes were chosen to be small enough to train quickly while still allowing the model to learn non-linear relationships that a simpler linear model could not, but other applications of our techniques may require tweaking the size of the network. We leave heavy optimization of the network design or testing of alternative architectures to future work.

Data: Training data (X, y) with binary (0, 1) labels, batch size b, proportions ps, max steps n
Result: A trained network
begin
  for $step \leftarrow 1 \ldots n$ **do**
    Shuffle training data;
    Split training data into batches of size b;
    for $batch \in batches$ **do**
      Choose a random proportion $p \in ps$;
      $sub\_batch \leftarrow \text{MaxSizeSubsetWithProportion}(batch, p)$;
      $y_{true} \leftarrow \text{mean}(sub\_batch_y)$;
      $y_{pred} \leftarrow \text{Predict}(sub\_batch_X)$;
      $loss \leftarrow \text{KLD}(y_{true}, y_{pred})$;
      Update the network using $loss$;
    end
  end
end

Algorithm 1: Pseudocode of the random-proportion batching training procedure used in NNQuant.

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 [53] lemmatizer provided by the Natural Language Toolkit (NLTK) [40], compute token counts, and apply a term-frequency inverse document frequency (TF-IDF) transform [59] to the token counts. Separately, we also extract all of the labels (tags) from each issue, normalize them to lowercase,[2] 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.

[2]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.

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 1,097 issues carefully, 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". 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, use-after-free, XSS, CSRF), 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

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. This dataset is available in our accompanying online materials [63].

Table 3.1 shows the top repositories included in the set of "security"-tagged issues we labeled. Despite not constraining the number of issues per repository, no single repository appears to dominate our sample of security issues.

Table 3.1: Top 20 repositories in our set of "security"-tagged issues that we ended up hand-labeling and including in our quantifier dataset. The largest contributor, `owncloud/core` only accounted for 39 "security"-tagged issues. It does not appear that any single repository dominated our dataset.

Repository	# Issues
owncloud/core	39
zcash/zcash	32
grpc/grpc	31
uclibs/scholar_uc	25
NixOS/nixpkgs	21
pavel-pimenov/flylinkdc-r5xx	18
GEWIS/gewisweb	14
defuse/php-encryption	12
AdguardTeam/AdguardForWindows	12
okulbilisim/ojs	10
chocolatey/choco	10
thaliproject/Thali_CordovaPlugin	9
brave/browser-laptop	9
GDG-Ukraine/gdg.org.ua	9
Electric-Coin-Company/zcash	9
oraj-360/oraj360tool	8
mozilla/id.webmaker.org	8
education/classroom	8
UprootLabs/gngr	8
Nubisproject/nubis-meta	8

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 [12] classifiers under a variety of settings, along with various configurations of our neural network-based quantifier. Figure 3.2 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 analyses.

Finally, Figure 3.4 shows the relative error over only low proportions $p \in [0.0, 0.2]$ for the same quantifiers. The relative performance of the quantifiers is similar to the performance for the full range, but the full range neural network quantifier outperforms the others in this restricted interval.

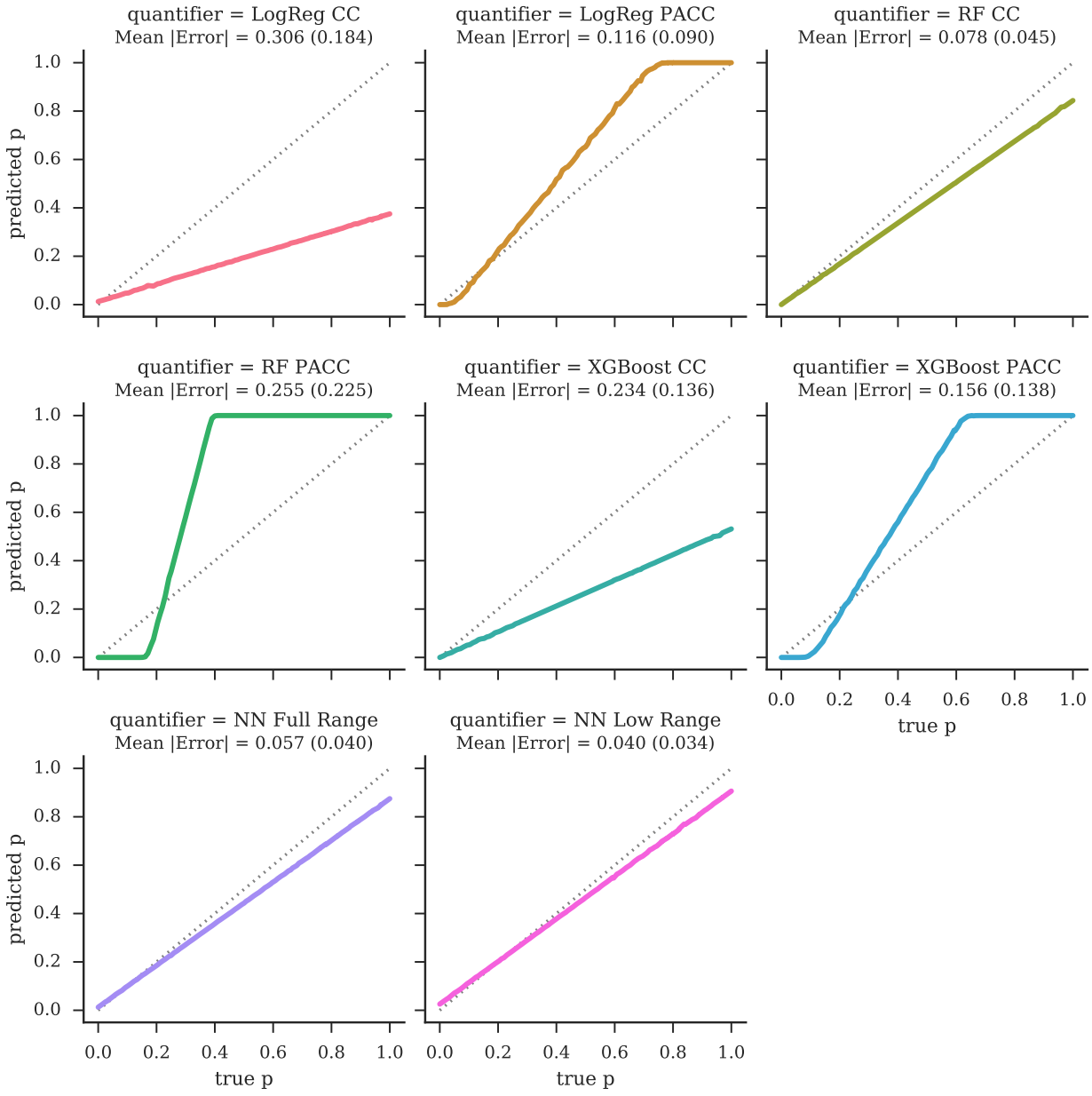

Figure 3.2: Plots of predicted proportion vs. true proportion for our quantifiers on our reserved test set. The dotted line marks the line $y = x$, which represents the ideal (a quantifier with no error); closer to the dotted line is better. Each quantifier is labeled with the mean absolute error over 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%.

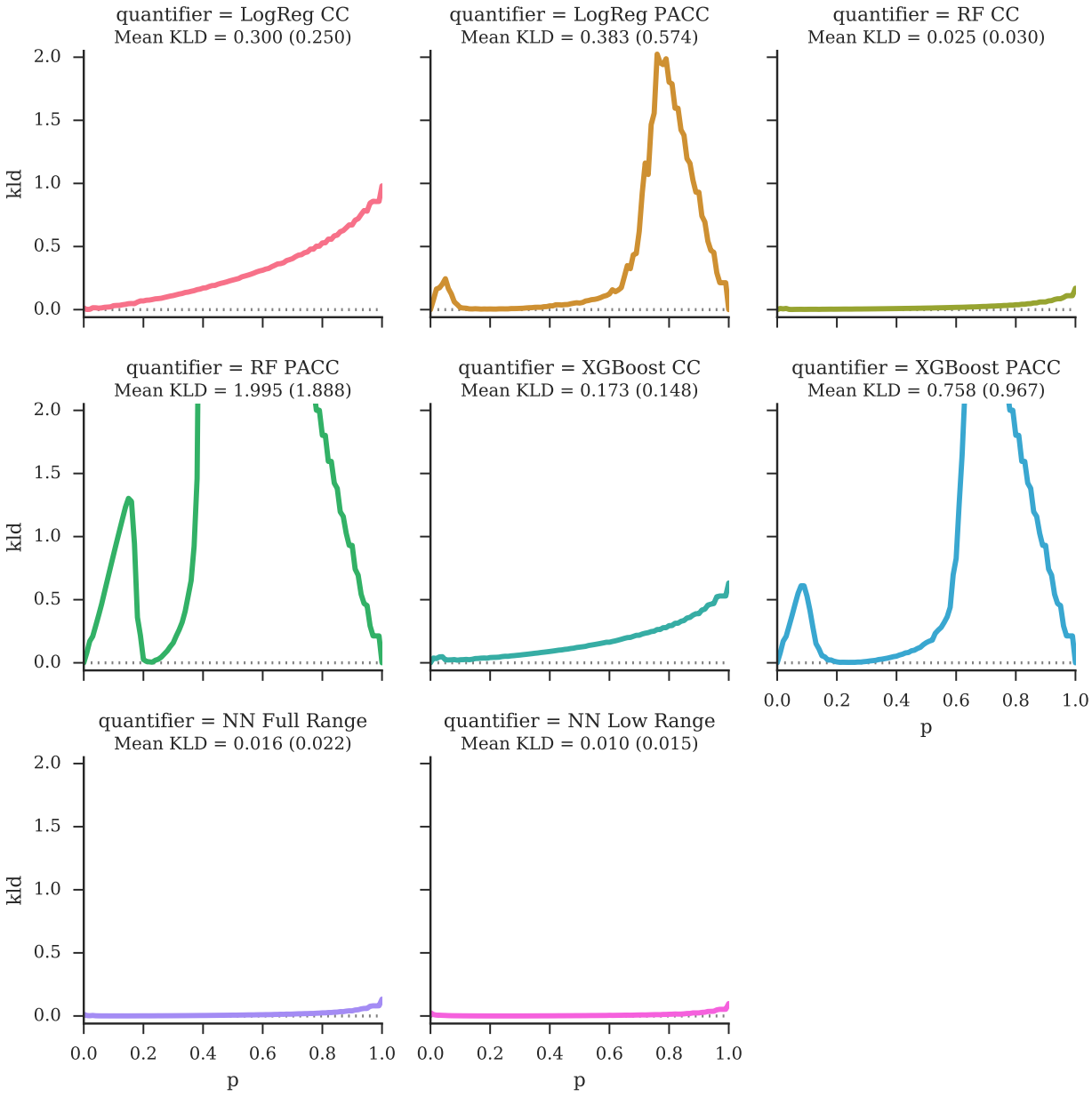

Figure 3.3: Plots of KLD loss vs. true proportion for our quantifiers on our reserved test set. Lower is better. Each quantifier is labeled with the mean KLD loss over all proportions and the standard error of that mean. Our "low range" neural network quantifier trained over $p \in [0.0, 0.1]$ shows the best performance, with a mean KLD of only 0.01.

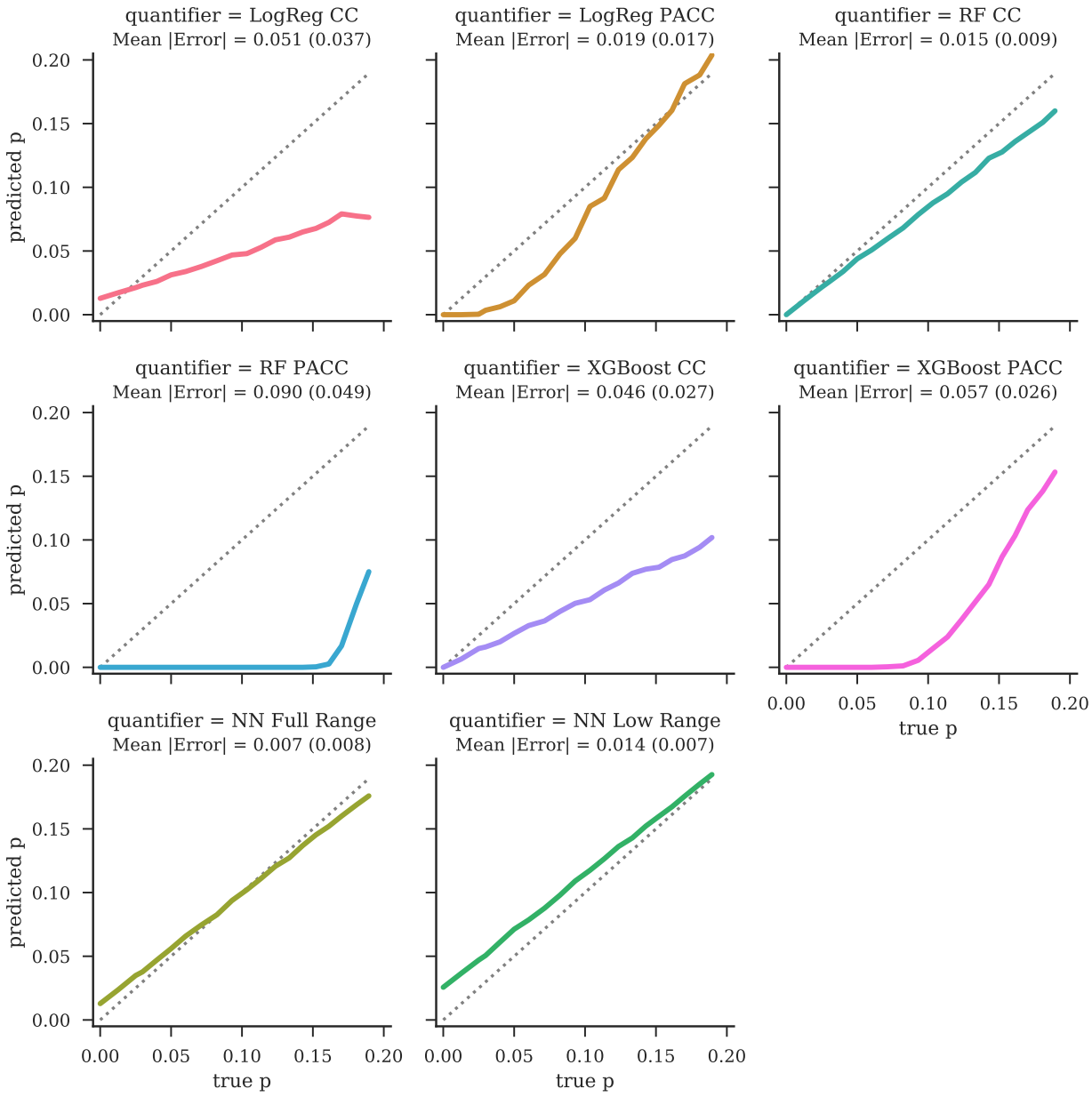

Figure 3.4: Plots of predicted proportion vs. true proportion for our quantifiers on our reserved test set for only low proportions $p \in [0.0, 0.2]$. The dotted line marks the line $y = x$, which represents the ideal (a quantifier with no error); closer to the dotted line is better. Each quantifier is labeled with the mean absolute error over all proportions in that interval and the standard error of that mean.

Chapter 4

Association Analysis of Code Review

An earlier version of the work in this chapter appears in the Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE) [64].

Empirical 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 to encourage the adoption of code review processes. Prior work in this area has primarily been limited to case studies of a small handful of software projects.

In this chapter, we extend and generalize that prior work in order to better understand the relationship between code review coverage and participation and software quality and software security. To do this, we gather a very large dataset from GitHub (2,881 projects in 139 languages, with 464,280 issues and 354,548 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. We control for confounding effects such as team size, project size, project age, project popularity, and memory safety. To be able to perform such a large-scale association analysis on GitHub repositories, where hand-labeling the almost 500,000 issues in our sample would be prohibitively expensive, we use our quantification techniques from the previous chapter to estimate the number of security bugs reported to each repository in our sample.

We find that code review coverage has a significant relationship with 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. 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.

4.1 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 used the GitHub Archive [28], a collection of all public GitHub events (including new commits, forks, pull requests, and issues), which is hosted on Google BigQuery [27], to generate a list of all such repositories. This gave us 48,612 candidate repositories in total. From this candidate set, we randomly sampled 5,000 repositories.[1]

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 [25], 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 being moved or deleted).

We queried GitHub to obtain the top three languages used by each repository, according to GitHub's own heuristics [26]. For each such language, we manually labeled it on two independent axes:

- Whether it is a programming language (versus a markup language like HTML, a configuration language like YAML, etc.) or not
- 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. [62] for a systematization of memory safety issues). For example, Python is considered a "memory-safe" language as all array accesses are bounds-checked at runtime.

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; 1,008 repositories)
- Repositories with no pull requests (406 repositories)

[1] Due to the limitations of scraping the live GitHub API for the data required, we chose take a large sample rather than analyzing the full set of candidate repositories.

Table 4.1: Summary of our sample of GitHub repositories. See Tables 4.3, 4.4, and 4.5 for more details about these metrics. We note that here "security issues" refers to the estimate of our quantifier.

	Mean	Min	25%	Median	75%	Max
Stars	425.230	0	8	46	237	33 880
Forks	107.655	0	6	23	77	5989
Contributors	24.506	4	6	11	25	435
Age (days)	1132.282	0.308	736.426	1112.877	1518.748	3047.529
Code Size (KB)	3364.896	0.089	77.627	271.952	1100.207	471 603.800
Issues						
Overall	161.152	5	19	47	143	8381
Security	7.173	0	1	2	6	273
Security / Overall	0.051	0	0.009	0.039	0.070	0.68
Churn						
Total	79 353.810	0	681	4710	27 406	46 293 722
Unreviewed	36 484.250	0	0	205	3412	46 291 762
Pull Requests						
Total	123.064	1	12	34	95	9060
Unreviewed	15.913	0	0	2	10	1189
Per Pull Request						
Review comments	0.026	0	0	0	0	17
Commenters	0.749	0	0	1	1	5

- 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 the primary language is not a programming language (a conservative heuristic for a repository not being a software project; 245 repositories)
- Repositories with a dedicated channel for security issues (1 repository)

This left us with 2,881 repositories in 139 languages, containing 464,280 issues and 354,548 pull requests. Table 4.1 shows a summary of the repositories in our sample. Table 4.2 lists the top 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 4.2.

We found that, in our entire sample, security bugs make up 4.5% of all issues (as predicted by our final trained quantifier; see Chapter 3 for how we derived this es-

Table 4.2: Top primary languages in our repository sample.

Language	# Repositories	Median Primary Size (KB)
JavaScript	660	192.81
Python	418	217.67
Ruby	315	99.69
Java	305	563.47
PHP	268	302.13
C++	172	1122.69
C	140	1626.69
C#	83	513.72
Shell	62	29.75
Objective-C	58	296.60

timate). This proportion is similar to but slightly higher than the results of Ray et al. [55] where 2% of bug fixing commits were categorized as security-related.[2]

4.2 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?*

We use negative binomial regression modeling to describe the relationship between code review coverage (the proportion of changes receiving at least some review) and participation (the extent of review on average) and both the number of issues and the number of security bugs filed on each project (our response variables). In our models

[2]Ray et al. [55] categorized bug fixing commits by training a support vector machine using a subset of commit messages selected using keyword search (for a variety of categories) and predicting the categories of the remaining commits. They reported having between 70-100% accuracy and 69-91% recall for the different categories they studied.

Table 4.3: Description of the control metrics.

Metric	Description	Rationale
Forks	Number of repository forks	The more forks a repository has, the greater number of 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	A proxy for the popularity of a project. On GitHub, users interested in a project can "star" the repository. More popular projects, with more users, will tend to have more bug reports and more active development.
Code Size	Size of code in 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 [48, 49].
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 [51] 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 [47, 48] and software security [61].
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 [62]. 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.

Table 4.4: Description of the code review 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 [42] and security bugs [44].
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.

Table 4.5: Description of the code review participation metrics (RQ 3, 4).

Metric	Description	Rationale
Average Commenters	Median 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 [44].
Average Discussion Comments	Median 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.
Average Review Comments	Median 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 also include a number of control explanatory variables (such as the age, size, churn, number of contributors, and stars for each repository). Tables 4.3, 4.4, and 4.5 explain each of our explanatory variables.

Negative binomial models are well suited for discrete counts, allow for non-normal residuals, and constrain the predictions to non-negative values, improving the fit and power of our analysis [41, 57]. Similarly, Ray et al. [55] used negative binomial models for their analysis of programming languages and bug-fix commits. Negative binomial models are also able to handle overdispersion, where the response variance is higher than other discrete models would assume (such as a Poisson model, where the mean and variance of the response are equal). Simple tests by fitting a quasi-Poisson model to our data show that our response variables are clearly overdispersed. Bcause of this overdispersion, a standard Poisson model would underestimate the standard errors of its coefficients, while using a negative binomial model gives us more accurate standard error estimates. We refer the reader to [57] and [41] for a more in-depth look at the background and applications of discrete count regression modeling and the negative binomial model.

Precisely, our models are negative binomial models with a log link function, which means that the *log* of the dependent variable ($\ln(y)$) is a linear combination of the predictors, plus an error term. This means that the coefficients have a *multiplicative* effect in the y-scale. These multiplicative effects are often referred to as "incidence rate ratios" or "risk ratios" (the latter is often used in epidemiology, and is the term we will use going forward).

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 ($|\rho| > 0.7$ [45]), we only include one of the two in our model. This resulted in us dropping the number of forks (too highly correlated with the number of stars) and the number of watchers (also too highly correlated with the number of stars). For our review coverage models, the amount of unreviewed churn was highly correlated with the number of number of unreviewed pull requests, so we chose to keep only the number of unreviewed pull requests.

To determine whether the coefficients for each explanatory variable are significantly different from zero, we perform a Wald z-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 4.6: Negative binomial association analysis model risk ratios. Listed are the risk ratios for each of the explanatory variables, when significant. These are the exponentiated coefficients of the models, and they indicate the relative (multiplicative) change in the dependent variable (number of overall issues or number of security issues) given a one-unit change of the explanatory variable. Each model also includes the code size, total churn, age, number of contributors, number of stars, number of pull requests, and memory safety for each repository. The full models, including all covariates, are listed in Appendix A.

	Dependent variable			
	Issues		Security Issues	
	(1)	(2)	(3)	(4)
unreviewed pull requests	1.0030***		1.0017***	
commenters per pr		◇		◇
review comments per pr		0.9087*		◇

◇ $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

4.3 Results

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

Prior work has found significant effects between code review coverage and defects in an analysis of three large software projects [42]. To investigate this relationship on a more general sample, we build a negative binomial model using the number of overall issues in a repository as the response variable, and the number of unreviewed (but integrated) pull requests as the explanatory variable. The model is presented in Table 4.6 as model (1).

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 number of unreviewed pull requests in a project and the number of issues the project has. Projects

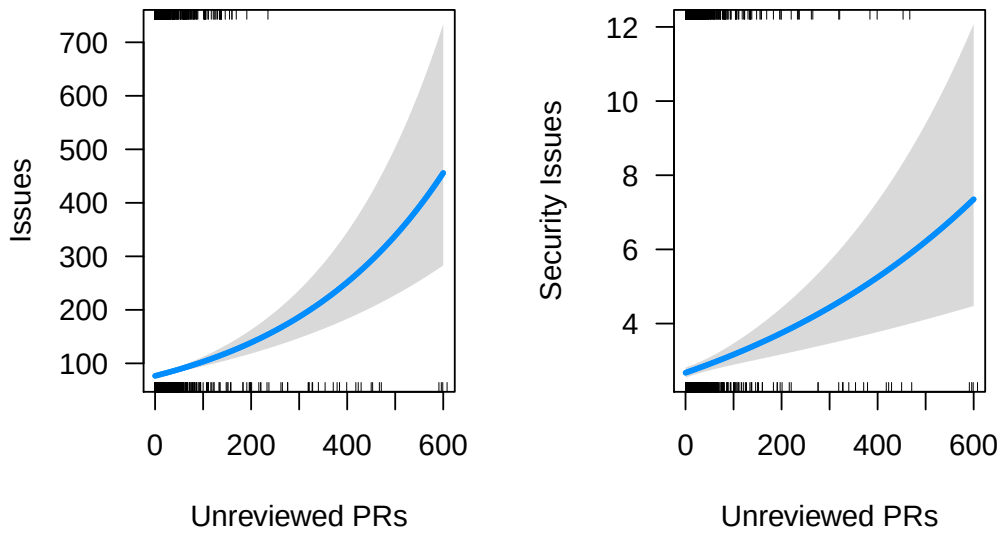

Figure 4.1: The marginal relationship between unreviewed pull requests and overall issues (left) and security issues (right). The relationship with security issues is weaker than the relationship with overall issues (having a risk ratio of 1.0017 versus 1.0030).

with more unreviewed pull requests tend to have more issues. Holding other variables constant, with an increase of 10 unreviewed pull requests we would expect to see 3.02% more issues. The relationship between the number of unreviewed pull requests and the number of overall issues is shown in Figure 4.1.

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

To explore this question, we replace the response variable of our previous model with the number of security bugs (as predicted by our quantifier for each project). The model is presented in Table 4.6 as model (3).

We find a small but significant positive relationship between the number of integrated pull requests that are unreviewed and the 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 an increase of 10 unreviewed pull requests, we would expect to see 1.70% more security bugs. The relationship between the number of unreviewed

pull requests and the number of security issues is shown in Figure 4.1.

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 number of issues in a project, and we replace our main explanatory variable with the median number of commenters on pull requests and the median number of review comments per pull request in each project. The model is presented in Table 4.6 as model (2). We did not include the median number of discussion comments in the model as it was highly correlated with the median number of commenters. We chose to keep the median number of commenters and the median number of review comments, to capture both aspects of participation (diversity of participation and quality of participation).

We do not find a significant relationship between the median number of commenters on pull requests and the total number of issues. However, we did find a small but significant negative relationship between the median 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, increasing the median review comments per pull request by 1 we would expect to see 9.13% fewer issues. The size and uncertainty of these relationships are shown in Figure 4.2.

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 number of security bugs reported in a project. We do not find a significant relationship between the median number of commenters on pull requests and the number of security bugs. This result is in contrast with that found by Meneely et al. [44]. 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 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 median number of review comments per pull request and the number of security bugs reported.

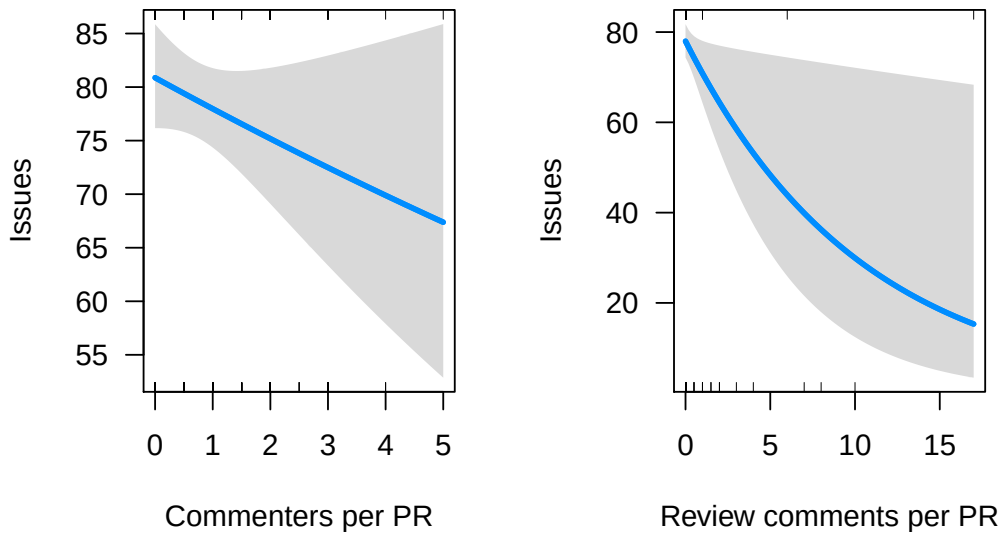

Figure 4.2: The marginal relationship between commenters per pull request and overall issues (left) and review comments per pull request and overall issues (right). We did not find a significant relationship between the average number of commenters and overall issues.

4.4 Threats to Validity

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. 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 4.6).

We manually label issues as “security bugs” to train our quantifier. We have a spe-

cific notion of what a security bug is (see Section 4.1), 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).

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. It is also possible that some code review occurs out-of-band and we would not measure it.

Some of our control explanatory variables are proxies for the underlying constructs we are trying to control for. The number of stars and watchers are a proxy for the popularity and user base of a project. The number of contributors to a project are a proxy for its activity and team size. These proxies may be incomplete in capturing the underlying constructs.

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.

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 [9, 24]. 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 3.2). The regression models produced using the predictions from this quantifier had the same conclusions as our results in Section 4.3, 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.

4.5 Discussion

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. [42] and Meneely et al. [44]. 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. In the next chapter we will explore how to determine if there is a causative effect of code review on quality and security, and how effective it is as a treatment.

Chapter 5

Causal Treatment Effects of Code Review

Previously, we explored the relationships between code review practices and both software security and software quality using correlative methods. In this chapter, we tease apart the questions from Chapter 4 to consider code review process as a "treatment" that can be applied to software projects generally, exploring the causal effects of these practices on software security and software quality. As in medicine, measuring the treatment effect allows a comparison of treatments, and a weighing of their effectiveness against their costs (and potential side effects). We hope that understanding the treatment effect of implementing code review processes allows for more informed (and economically motivated) decisions for improving software quality and security.

The gold-standard for measuring treatment effects is the randomized controlled trial. In an ideal experiment, we would:

- Recruit active software projects that currently only use direct commits,
- Randomly assign half of the projects to start integrating all changes via pull requests with at least one peer reviewer (the other half would continue without change),
- Measure the development behavior before and after the intervention,
- Measure the state of the security and quality of each project before and after the intervention.

However, performing real-world, randomized experiments on software process is difficult and potentially cost-prohibitive. Even studies on smaller artificial tasks tend to lack random assignment of the observed variables of interest (such as the "Build It Break It Fix It" contests [58]). Real world software development often happens at large scales, where experimental units are bigger and more complex, and harder for researchers to recruit and manipulate. Meanwhile, retrospective analysis of changes

in industrial settings have proven useful for exploring the effects of changes in software process (such as Microsoft's analysis of switching to use formal unit testing processes [39]), but they fall into the broad category of non-experiments.

We tackle this problem by taking advantage of the longitudinal data provided by GitHub and the GitHub Archive to perform quasi-experiments on the effects of code review as a software security and software quality intervention. Our insight is the use of timeseries *changepoint detection* to allow us to find cases of code review "treatment" among a very large set of GitHub projects.

Some variations of quasi-experiments are sometimes called *natural experiments*, where the intervention was exogenously applied, outside the control of the researchers or the units. For interventions that are difficult (or potentially impossible) for the researchers to assign, natural experiments have allowed research into the effects of changes in areas such as government structure [23] or specific policy implementations [32]. Other quasi-experiment designs take advantage of observational or archival data (as we do), or are used in cases where it is difficult or unethical for the researchers to randomly assign treatment to units [11, 14, 18]. For example, research into educational interventions has had ethical challenges with random treatment assignment and some have focused designs on giving interventions to lower achieving students [33].

One benefit of quasi-experimental designs like ours over true experiments (such as randomized control trials) is that they minimize threats to ecological validity, as we are observing software projects as they occur in the real world, rather than in an artificial setting. Additionally, the use of archival data reduces any *testing effect* on our sample. In other designs, the participants can know what they are being tested on, potentially influencing their behavior. In contrast with a non-experimental observational study, our quasi-experimental design meets both the time-causality and treatment/control comparisons requirements for causal inference.

5.1 Changepoint Detection

We use a timeseries *changepoint detection* approach to identify candidate repositories for our treatment group by detecting repositories that demonstrate a significant change in code review coverage (that is, repositories where there is a single point in time where the level of code review coverage before and after are significantly different). Similarly, we can use the absence of change to pick a control group of repositories where little or no change in code review coverage occurred.

Changepoint detection (sometimes called *change detection* or *step analysis*) is a method to statistically identify times when a parameter underlying a stochastic process or timeseries changes [5]. This allows us to segment timeseries by changes in their underlying processes. While online change detection is used in many detection applications, in our setting we are able to use offline techniques such as statistical hy-

pothesis testing or maximum likelihood estimation, finding the best possible splits in the available data. This can yield a better trade-off for false alarm versus misdetection rates.

One (simplified) way of thinking about single changepoint detection is that we are searching the space of all possible splits in the timeseries for the split which maximizes the difference of the means before and after the split. We can then test whether the change is statistically significant.

More complicated techniques reduce the number of split points that we have to check or allow finding multiple changepoints (finding all significant changes in mean). For our purposes, we only require finding a single, maximal changepoint in our archival timeseries data. This technique is sometimes called "At-Most-One-Change" (or AMOC) changepoint detection. We use a cumulative sum (CUSUM) statistic [52], which does not rely on assumptions about the data being normally distributed. For an overview of changepoint detection and how the various techniques function, we refer the reader Bassevile's book on the subject [5], as well as Killick's description of the `changepoint` R package [37].

Figure 5.1 shows an example of a changepoint in the mean of code review coverage on a repository from one of our treatment groups. We apply changepoint detection techniques to our code review coverage metric of GitHub repositories over time (the fraction of all pushes and pull requests that receive some code review). Repositories that show a large and statistically significant change in code review coverage can thus be efficiently detected and selected for inclusion in our treatment groups. Similarly, we find repositories that demonstrate no detectable change in code review (and stay under a "no treatment" threshold of such activity) for inclusion in our control group.

5.2 Data Collection

To select the repositories for our treatment and control groups, we use the data in the GitHub Archive [28]. We build our treatment and control groups from the population of repositories that were active in 2015 through July 2017. For this selection, we define "active" as repositories having in both 2015 and 2016 at least 50 issues, 50 pushes, and 10 actors;[1] and in 2017, the equivalent activity (for $7/12$ of the year) of 29 issues, 29 pushes, and 6 actors.

We use BigQuery to select repositories that meet these requirements, giving us a subpopulation of 4,004 repositories. Further, we then use BigQuery to gather each merged pull request and push to each repository, to measure their code review coverage over time. We apply the same criteria for "unreviewed" pull requests as in Chapter 4, except here we also count direct pushes of commits to any default branch as unreviewed changes. To avoid double-counting pushes, we remove any pushes that resulted from

[1] We use the term "actor" to mean unique user causing an event we counted. For our selection here, this means unique users creating an issue or generating a push.

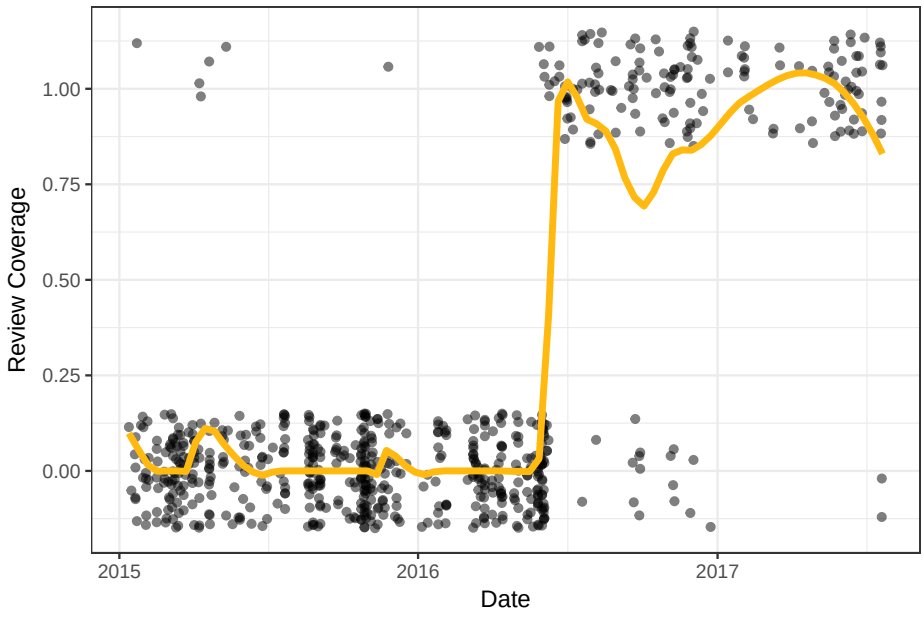

Figure 5.1: An example of a code review coverage timeseries plot showing a clear changepoint in coverage in mid-2016. This is the repository `silx-kit/pyFAI`, which our technique detects. Before the changepoint, it has 1.6% review coverage. After the changepoint, it has 86.5% review coverage. The points in the plot show each push and merged pull request for the repository, and whether it was reviewed or not (with added jitter to show density—each individual pull request is either reviewed or not). The yellow line is a LOESS fit line (on the points before jitter was added) to demonstrate the local trend in coverage and the sharp change at the changepoint.

a pull request being merged, by matching the SHA of the head commit of pushes with the SHA of the merge commit of pull requests.

Using that aggregate code review data, we use the "AMOC" changepoint detection method (with a non-parametric test statistic) from the R `changepoint` package [37] to detect changepoints in the code review coverage in each repository. We use a manual penalty value of 0.1 (chosen as a low threshold for change and manually validated as reasonable against a small sample of projects) and a minimum segment length of 1/4 the total number of events in each repository (so that we have a minimum number of events and time to compare on either side of the changepoint).

From the results of our changepoint detection, we select three groups:

- A **control group** of repositories that showed no significant change in code review coverage, and had an overall coverage under 25% (1,689 cases).
- A **high treatment group** of repositories in which we found a significant change in code review coverage, and the coverage before the change was under 25% and

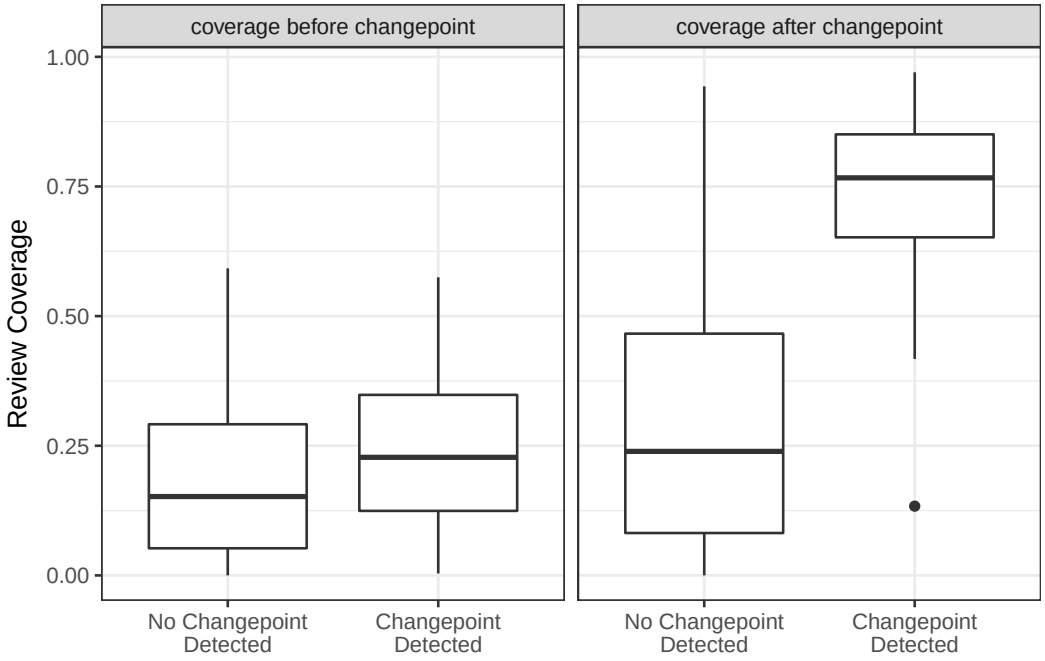

Figure 5.2: The distribution of code review coverage for the two groups: repositories that did not have a changepoint, and those that did. The review coverage is split into before- and after-changepoint periods. For the no changepoint group (when split at their mid-point in time) over 50% of such repositories had review coverage under 25% both before and after. For repositories with changepoints, over 50% of such repositories began with under 25% code review coverage, but over 50% ended with more than 75% coverage. These distributions motivated our control and treatment cut-off thresholds.

the coverage after the change was at least 75% (45 cases).

- A **medium treatment group** of repositories in which we found a significant change in code review coverage, and the coverage before the change was under 25%, but the coverage after the change was at least 50% but less than 75% (113 cases).

Figure 5.2 shows the distribution of review coverage for repositories in which we found a changepoint and those we did not. The medians before and after for each group motivate our control and high-treatment group cutoff thresholds.

Once we have selected our treatment and control groups, we then gather the full text of all issues, the full text of the README file, and the general metadata for each repository. We split the issues into before- and after-changepoint periods (we refer to these as "pre-test" and "post-test" going forward). For our control group, we split at the midpoint in time of our collection period (16 April 2016). For the issues in each time

period we apply the same security issue quantifier as used in Chapter 4 to estimate the number of security bugs reported before and after the changepoint.

We then clean and filter our selected repositories as follows:

- Remove repositories with a dedicated channel for security issues (such as a `security@` email address—we would not be able to find security issues in their issue tracker; 13 repositories from our control group)
- Remove repositories with a blank primary language or where the primary language is not a programming language (these are generally non-software repositories such as documents, datasets, or static webpages; 12 repositories with blank primary language, 90 repositories with a non-programming primary language)
- Remove repositories that are forks of another repository, where both parent and child are currently active (we lose the ability to capture all development on the project; 14 repositories)

After cleaning and filtering, we were left with 1,576 control cases, 102 medium treatment cases, and 42 high treatment cases. They encompass 145 languages, with 1,063,342 issues (43,907 of them security bugs, or 4% of all issues). Table 5.1 gives a summary of each group, and Table 5.2 shows the top primary languages for repositories in our selection.

Scatterplots showing the relationship between each explanatory variable we included in our models and the number of post-test security and overall issues are shown in Figure 5.3 and Figure 5.4.

Automated Code Review

To further demonstrate the broader usefulness of our changepoint treatment detection technique, we examined the use of automated code review services on GitHub. A number of services are available that integrate with pull requests, which automatically review the changes for style problems or common mistakes. We looked at three automated code review tools: Hound,[2] Codacy,[3] and Ebert.[4] These three tools all create comments on the original pull request with bot accounts, which allows us to measure comments created by these services using the GitHub Archive data.

We measured each pull request in our repository population for whether they have any bot comments from these services, and computed changepoints using our same detection technique as before. For our control group, we selected all repositories that never had any bot comments over our entire collection period. For our treatment group, we selected all repositories that had no bot comments before their changepoint, with

[2]https://houndci.com
[3]https://www.codacy.com
[4]https://ebertapp.io/

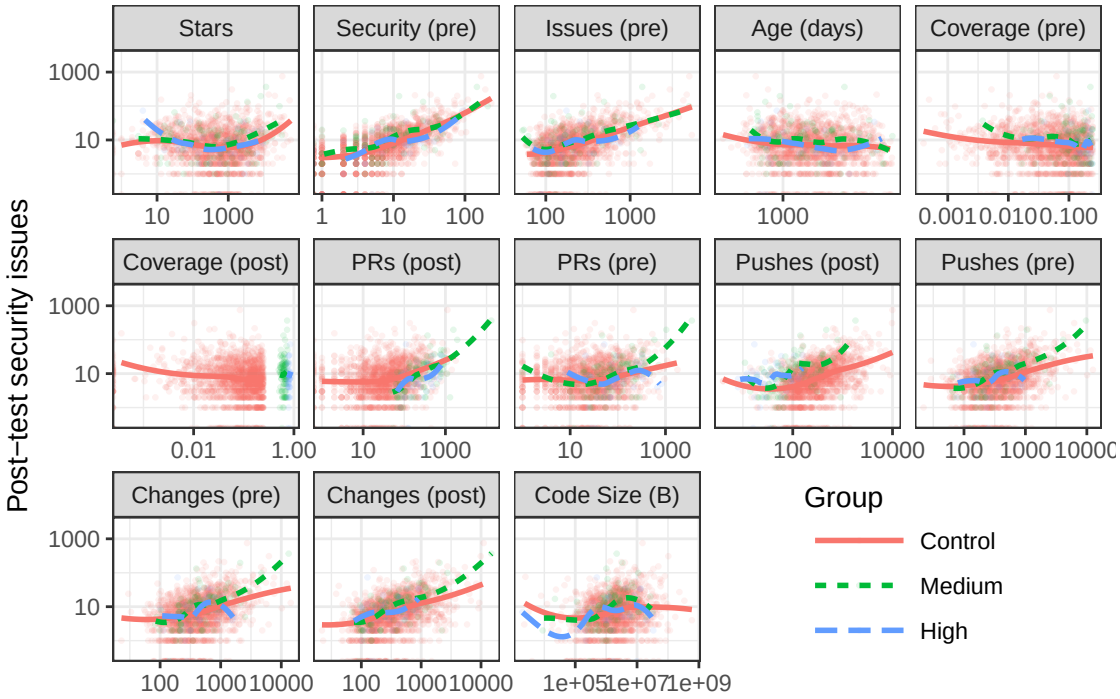

Figure 5.3: Scatterplots of explanatory variables and post-test security issues. Scales are log transformed as many variables have very large ranges. Loess local regression lines are plotted on top of the points to more clearly show the trends of each group.

any non-zero number of bot comments after. Before cleaning and filtering our selection, we had 3,803 control cases and 9 treatment cases. After filtering (using the same steps as before), we had 3,535 control cases and 8 treatment cases. Table 5.3 gives a summary of our final control and treatment groups.

We additionally measured the code review coverage before and after the implementation of automated tools (or the midpoint, for the repositories that did not implement automated review) to control for pre-existing code review practices.

5.3 Experimental Design

In a randomized controlled trial, the randomized group assignment creates *equivalent* groups—any variation between participants is evenly spread between groups, controlling for all variation in even confounding factors that had not been considered by the experimenter. As we cannot randomly assign repositories to the treatment and control groups, our treatment and control groups give us a *non-equivalent groups design* quasi-experiment. Our quasi-experimental design is heavily influenced by the work of

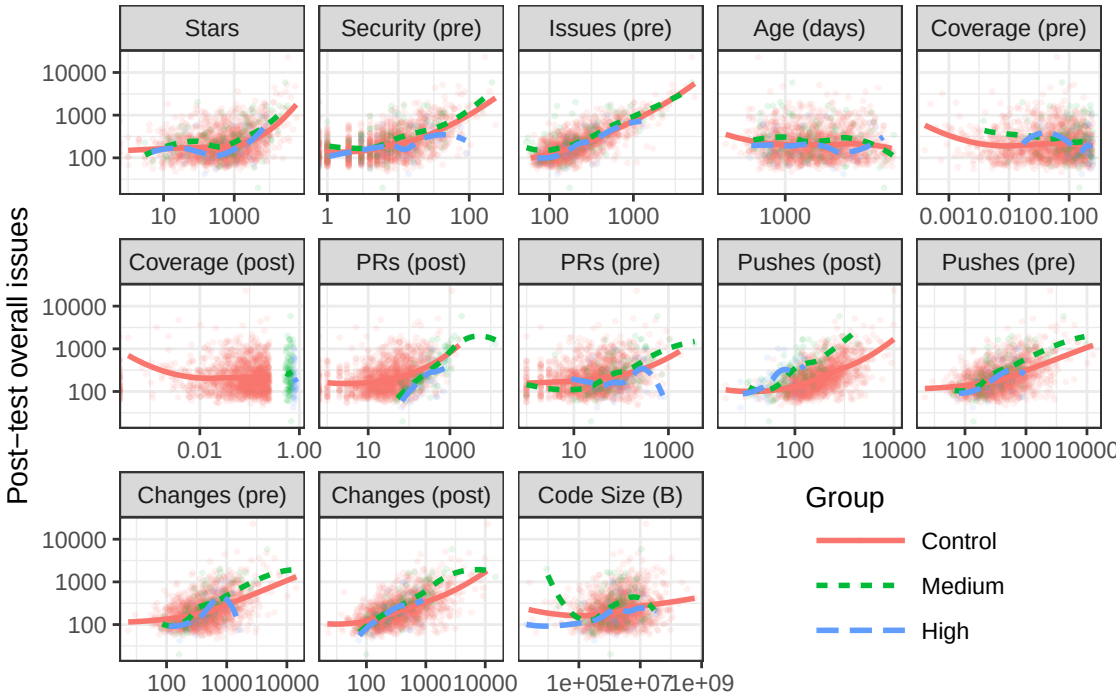

Figure 5.4: Scatterplots of explanatory variables and post-test overall issues. Scales are log transformed as many variables have very large ranges. Loess local regression lines are plotted on top of the points to more clearly show the trends of each group.

Shadish, Cook, and Campbell on experimental and quasi-experimental designs [60], and the work of Trochim on research methods [66].

We seek to answer the following four research questions:

(RQ1) *Does implementing code review improve software quality?*

(RQ2) *Does implementing code review improve software security?*

(RQ3) *Does using automated code review services improve software quality?*

(RQ4) *Does using automated code review services improve software security?*

Our hypothesis was that, up to some point, higher dosage (more code review coverage) decreases the number of both security and general software bugs reported to a project. For our study, we looked at two different dosage levels (our high and medium treatment groups) compared to our control group.

For each period, we measure the aggregate code review coverage (the fraction of changes that had at least some review), the total number of issues filed during that

Table 5.1: Summary of our sample of GitHub repositories, by group.

	Control (n=1576)		**Medium** (n=102)		**High** (n=42)	
	mean	(sd)	mean	(sd)	mean	(sd)
Stars	1888.07	(4479.99)	2539.17	(5529.44)	1703.21	(2446.84)
Age (days)	1491.53	(573.71)	1385.38	(593.05)	1452.20	(660.14)
Code Size (MB)	5.86	(21.53)	2.99	(4.96)	4.48	(7.64)
Pull Requests						
pre-test	61.88	(105.76)	168.07	(378.00)	141.76	(166.69)
post-test	70.53	(118.63)	495.75	(1305.18)	287.21	(206.94)
Pushes						
pre-test	596.75	(838.65)	597.70	(1086.03)	367.67	(239.14)
post-test	500.71	(780.30)	156.72	(229.71)	50.21	(40.90)
Overall Changes						
pre-test	658.63	(901.54)	765.76	(1423.63)	509.43	(363.04)
post-test	571.24	(843.16)	652.47	(1505.39)	337.43	(242.18)
Review Coverage						
pre-test	0.07	(0.06)	0.13	(0.07)	0.17	(0.06)
post-test	0.10	(0.07)	0.63	(0.06)	0.80	(0.03)
Security Issues						
pre-test	12.51	(17.85)	14.31	(26.28)	13.67	(18.29)
post-test	12.48	(26.73)	19.94	(43.16)	10.86	(13.08)
Overall Issues						
pre-test	303.70	(401.51)	351.43	(567.97)	271.98	(260.49)
post-test	305.30	(681.99)	444.92	(678.41)	258.52	(237.69)

period, and the total number of issues filed during that period that were security issues (as estimated by our security issue quantifier, as described in Chapter 3).

To test for treatment effects, we consider two ways in which code review may affect security and quality outcomes. First, we can look for an *average* treatment effect, where there is a separation in post-test results between the average of the treatment group and the average of the control group, after conditioning on covariates. Second, we can look for *interaction* effects between the treatment assignment and our pre-test score, where the effect of treatment is greater for units with higher pre-test scores. In the context of software security and quality this may make more sense: a project that is already more defect prone may benefit more from code review as there are more defects to catch early on in the development process. We test for both by including interaction terms between our group assignment and our pre-test covariate in all of our models.

Table 5.2: Top primary languages in our repository sample.

Language	# Repositories		Median Primary Size (KB)	
		(Control, Medium, High)		
JavaScript	338	*(303, 24, 11)*	933	*(932, 980, 500)*
Java	290	*(274, 13, 3)*	2148	*(2166, 1842, 1626)*
Python	202	*(178, 15, 9)*	1246	*(1267, 598, 1311)*
C++	183	*(173, 6, 4)*	3106	*(3136, 2788, 3142)*
PHP	153	*(144, 8, 1)*	1606	*(1532, 1688, 7824)*
C	106	*(101, 3, 2)*	2222	*(2211, 1739, 9713)*
C#	97	*(86, 10, 1)*	2185	*(2169, 2332, 17561)*
Ruby	63	*(57, 5, 1)*	763	*(763, 2141, 208)*
Go	28	*(21, 4, 3)*	602	*(492, 929, 967)*
Shell	24	*(22, 2, 0)*	481	*(604, 14, –)*

Building on the regression analysis techniques we used in Chapter 4, due to our heavily skewed (and constrained domain) response variables, we apply count regression models for software quality and security, where our response is a count of overall issues or security issues. To account for our non-equivalent groups, we include a variety of potentially confounding covariates in the model. We only include covariates measured *before* the point of intervention (the changepoint for treatment cases or the midpoint for control cases).

We also note that simpler, log-normalized linear models using ordinary least squares regression performed particularly poorly in diagnostic tests for our data. While log-transforming the response variable can sometimes approximate normal residuals, our data is too heavily skewed (and truncated at zero), particularly for security issues.

Then, we repeat both models, but with the two treatment groups combined together. This gives us a larger sample size for our treatment group, giving us more statistical power, and allowing us to consider the effects for any repository which implements at least moderate code review practices.

Finally, we modify the models for both overall issues and security issues to look for treatment effects for the use of automated code review services. We include the same covariates as before, including the pre-test review coverage in order to control for pre-existing code review practices. We include interaction terms between the group assignment and both the pre-test value (issues or security issues) and the pre-test review coverage.

Table 5.3: Summary of our sample of GitHub repositories for automated code review usage, by group.

	Control (n=3535)		**Treatment** (n=8)	
	mean	(sd)	mean	(sd)
Stars	2533.40	(7127.83)	822.50	(974.29)
Age (days)	1513.00	(590.14)	1392.85	(432.75)
Code Size (MB)	5.52	(19.11)	1.95	(3.85)
Pull Requests				
pre-test	194.37	(454.14)	310.12	(220.67)
post-test	239.27	(608.88)	186.38	(129.43)
Pushes				
pre-test	564.69	(825.74)	552.50	(390.90)
post-test	351.81	(671.89)	105.38	(89.74)
Overall Changes				
pre-test	759.06	(1169.49)	862.62	(539.41)
post-test	591.08	(1011.20)	291.75	(183.36)
Review Coverage				
pre-test	0.18	(0.15)	0.29	(0.15)
post-test	0.32	(0.26)	0.51	(0.23)
Security Issues				
pre-test	14.92	(24.81)	16.00	(19.91)
post-test	14.89	(33.69)	7.88	(9.61)
Overall Issues				
pre-test	360.01	(525.07)	359.00	(267.45)
post-test	366.07	(690.51)	189.62	(160.69)

5.4 Results

RQ1: Does code review improve software quality?

To investigate the causal effect of code review coverage on software quality, we build a negative binomial regression model using the number of overall post-test issues in a repository as the response variable, and the group assignment (under treatment coding compared to the control group) as our main explanatory variables. Our pre-test covariates include the number of pushes and merged pull requests, the size of the repository, the number of stars, the age of the repository, the pre-test code review coverage, and the duration of the pre-test period. We include an interaction term between the pre-test number of issues and the group assignment. The model is presented in

Table 5.4.

We find that our medium treatment group has a significant negative interaction with the number of pre-test issues, while our high treatment group has a significant positive interaction with the number of pre-test issues. For repositories with higher numbers of pre-test issues, we find that our medium treatment group had *fewer* post-test issues than our control group, while our high treatment group had *more* post-test issues. For example, for repositories with 1500 pre-test issues (holding other variables constant), we would expect that those implementing moderate code review would have 248 fewer post-test issues than our control group, and those implementing stringent code review would have 2,097 more issues than our control group. This interaction is shown in Figure 5.5.

It is possible that our group selection is connected to some unmeasured confounds for the highest rate of code review. One hypothesis could be that despite requiring pull requests, projects in the high treatment group may be more likely to have quick or "drive-by" comments rather than substantive review. A project that reviews *most* changes (like in our medium treatment group) may actually encourage more thorough and useful review.

Table 5.4: Negative binomial models of effects of our medium and high treatment groups on software quality and security. Listed are the risk ratios for our treatment groups and their interaction effect with the pre-test numbers of issues and security issues, when significant. Each model also includes the code size, age, number of stars, memory safety, pre-test issues (and pre-test security issues for the security model), pre-test pushes, pre-test pull requests, pre-test review coverage, and the number of days in the pre-test period for each repository. The full models, including all covariates, are listed in Appendix A.

	Dependent variable (post-test)	
	Issues	Security Issues
	(1)	(2)
medium treatment	1.5780***	1.9289***
medium:pre-test issues	0.9995***	
medium:pre-test security issues		0.9778***
high treatment	⋄	⋄
high:pre-test issues	1.0008*	
high:pre-test security issues		⋄

⋄ $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

Table 5.5: Marginal effect of treatment on overall issues (with 95% confidence intervals). The predicted number of overall post-test issues at various pre-test issue counts while holding the other variables constant at their median.

	Pre-test overall issues			
Group	100	500	1000	2500
Control	182.25	310.16	602.88	4427.70
	(176.44–188.26)	(304.20–321.84)	(564.30–644.10)	(3651.74–5368.53)
Medium	274.58	388.33	598.89	2196.89
	(243.93–309.09)	(346.60–435.07)	(504.82–710.50)	(1411.48–3419.33)
High	177.88	408.63	1155.68	26 144.01
	(144.59–218.82)	(325.90–512.35)	(703.65–1898.11)	(6218.15–109921.69)

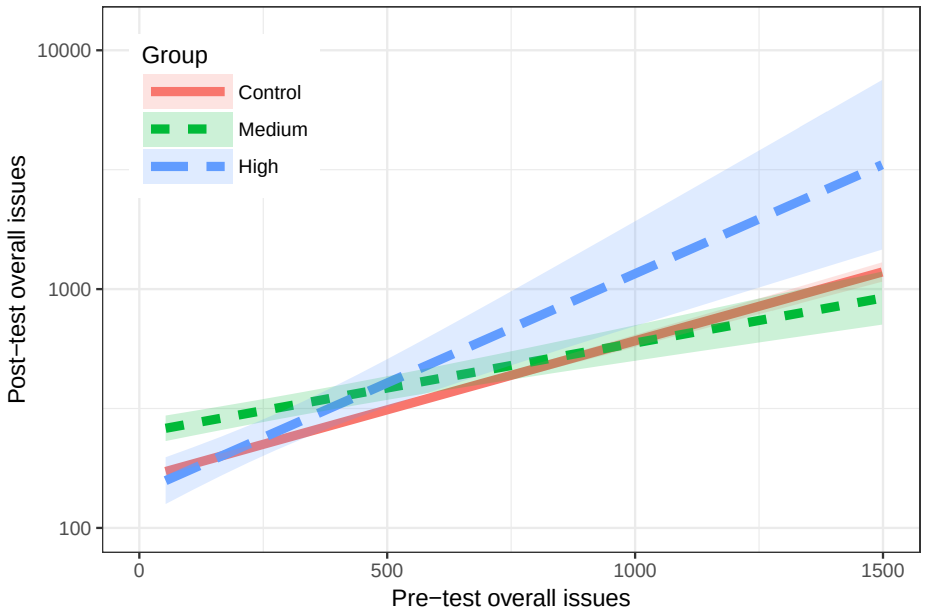

Figure 5.5: The marginal effect between the treatment groups and control groups over a range of pre-test issues. We see that as the number pre-test issues increases, the medium treatment group tends to have fewer issues, while the high treatment group tends to have more issues.

Table 5.6: Marginal effect of treatment on security issues (with 95% confidence intervals). The predicted number of post-test security issues at various pre-test security issue counts while holding the other variables constant at their median.

	Pre-test security issues			
Group	0	10	50	100
Control	6.04	8.52	33.68	187.66
	(5.71–6.39)	(8.17–8.89)	(29.96–37.86)	(144.67–243.42)
Medium	11.66	13.13	21.09	38.15
	(9.70–14.00)	(11.15–15.45)	(15.75–28.25)	(21.02–69.28)
High	6.77	8.96	27.42	111.04
	(4.87–9.41)	(6.81–11.77)	(16.05–46.83)	(35.03–352.00)

RQ2: Does code review improve software security?

To investigate the causal effect of code review coverage on software security, we build a negative binomial regression model using the post-test number of security issues as the response variable, and the group assignment (under treatment coding compared to the control group) as our main explanatory variable. We add the number of overall issues in the pre-test to our set of covariates. We include an interaction term between the pre-test number of security issues and the group assignment. The model is presented in Table 5.4.

We found that the medium treatment group had a significantly higher baseline post-test number of security issues, but it also had a significant negative interaction effect with the pre-test number of security issues. At higher pre-test security issues, the medium treatment group had *lower* post-test security issues than the control group.

One way of interpreting this is that implementing code review is more effective for repositories that were already more prone to security issues. However, with the high treatment group, we did not find any significant treatment or interaction effect. The effect of the medium treatment group for higher pre-test security issues is shown more clearly visually in Figure 5.6.

For example, if a project with 50 security issues implemented moderate code review (at least 50% coverage), holding other variables constant, we would expect that they would have 12.6 fewer security issues after, compared to projects that did not implement code review.

Two-group Design

Due to the high error in the fit for our high treatment group in our three group analysis, we also performed an analysis with our two treatment groups combined into a

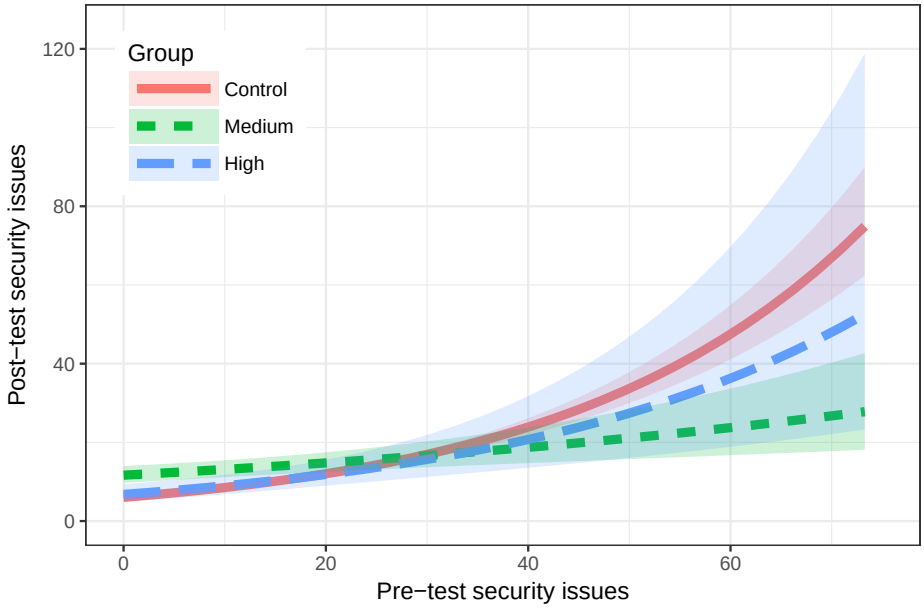

Figure 5.6: The marginal effect between the treatment groups and control groups over a range of pre-test rates. We see that as the pre-test number of security issues increases, the medium treatment group tends to have significantly lower post-test proportions of security issues. However, the high treatment group has too much variance (as shown here by its confidence interval band) for us to find statistically significant effects.

Table 5.7: Negative binomial models for the effect of a single combined treatment group on software quality and security. Listed are the risk ratios for our treatment group and its interaction effect with the pre-test numbers of issues and security issues, when significant. Each model also includes the code size, age, number of stars, memory safety, pre-test issues (and pre-test security issues for the security model), pre-test pushes, pre-test pull requests, pre-test review coverage, and the number of days in the pre-test period for each repository. The full models, including all covariates, are listed in Appendix A.

	Dependent variable (post-test)	
	Issues	Security Issues
	(1)	(2)
treatment	1.3867***	1.6559***
treatment:pre-test issues	0.9998*	
treatment:pre-test security issues		0.9840***

$\diamond$ $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

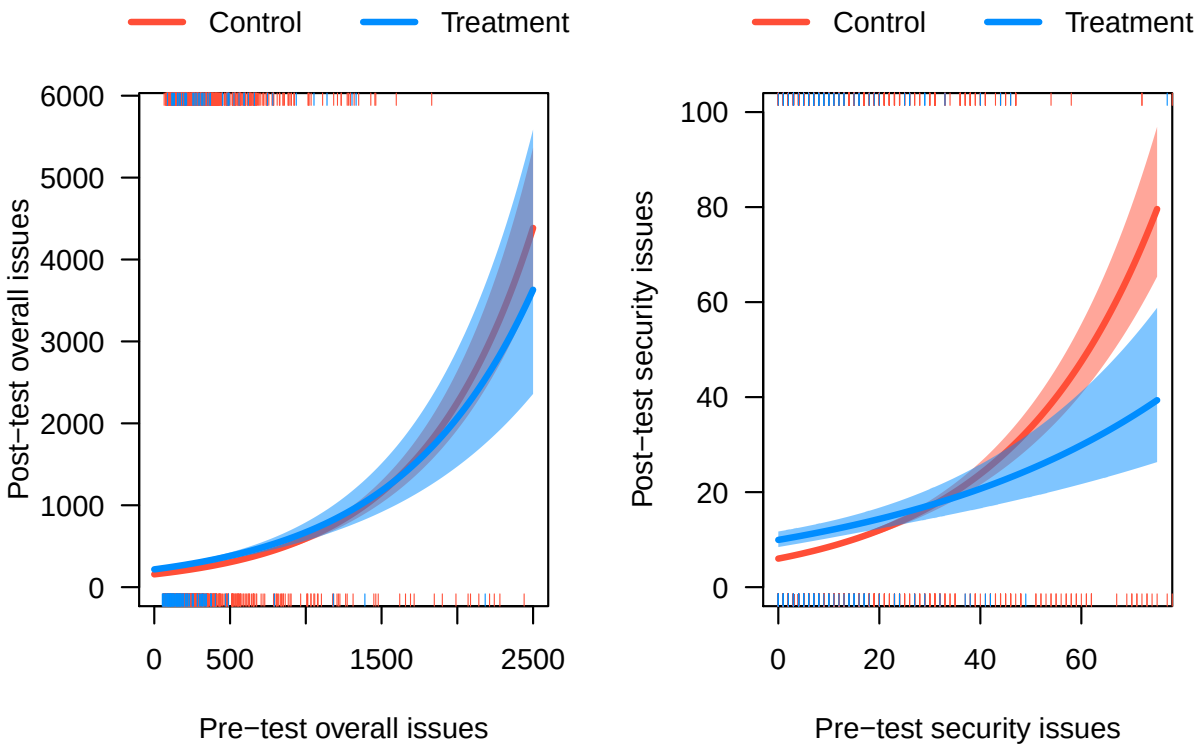

Figure 5.7: The marginal effect between the combined treatment group and the control group over a range of pre-test values, for both overall issues (left) and security issues (right). We see that the interaction effect is much stronger for security issues than for overall issues.

single group. This gives us greater statistical power for asking a slightly simpler question: does implementing at least some code review affect the quality and security of a project? We repeat the same analyses as in Sections 5.4 and 5.4, but combine the data for our medium and high treatment groups into a single treatment group.

These models are shown in Table 5.7. We see in both models that for the combined treatment group there is a significant negative interaction effect between the pre-test value and the treatment, although the effect is much larger for security issues than for overall issues. For a wide range of pre-test values, the treatment appears to have little or no effect on overall issues, but for higher security pre-test values there is a clear treatment effect. These interaction effects are visualized in Figure 5.7.

Table 5.8: Negative binomial models of effects of automated code review usage on software quality and security. Listed are the risk ratios for our treatment group and its interaction effect with the pre-test numbers of issues and security issues and pre-test review coverage, when significant. Each model also includes the code size, age, number of stars, memory safety, pre-test issues (and pre-test security issues for the security model), pre-test pushes, pre-test pull requests, pre-test review coverage, and the number of days in the pre-test period for each repository. The full models, including all covariates, are listed in Appendix A.

	Dependent variable (post-test)	
	Issues	Security Issues
	(1)	(2)
treatment	0.1984***	⋄
treatment:pre-test issues	⋄	
treatment:pre-test security issues		⋄
treatment:pre-test review coverage	⋄	⋄

⋄ $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

RQ3-4: Does automatic code review improve software quality and security?

Following the same methodology as before, we construct negative binomial regression models to analyze the effects of automated code review on overall issues and security issues. We did not find any significant effect of automated code review on post-intervention security issues, but we found a strong effect on the overall number of post-intervention issues. These models are shown in Table 5.8. We included interactions with both the number of issues and the review coverage for each repository, but we did not find significant interactions effects. Overall, controlling for a number of covariates, our treatment group had 80% fewer overall issues than our control group after the intervention (95% CI: 38.3%–93.6%).

5.5 Threats to Validity

Construct validity

The concerns raised in Chapter 4 apply to our analysis of treatment effects, as we are measuring most of the same metrics here as we did during our regression analysis. We are ultimately measuring treatment effects on proxies for the underlying software quality and security of projects.

Internal validity

The lack of randomized assignment in our quasi-experiment means that it is harder to rule out unobserved confounding variables. Since we cannot guarantee that our treatment and control groups are identical across all observed and unobserved covariates, we may inadvertently select cases such that they have differing baseline maturation rates of outcomes (overall issues and security issues), or our treatment cases may have started with much higher (or lower) outcomes than our control and we could see a regression toward the mean.

First, it is possible that a project will become more popular and actively used over time, and thus receive more issues filed against it (both security issues and general defects). This may be particularly true for security issues, as a popular project will be more likely to provide the economic incentives for security auditing and outside exploitation. Similarly, as a project matures, the quality and architecture of its code may improve, as well as the quality of its other development processes (e.g., use of testing), which could cause a project to naturally have fewer defects. These two directions (popularity and maturity) could yield opposite effects.

We examined potential differing rates of maturation by dividing the pre-test and post-test periods in half for each repository (the size of these periods is shown in Table 5.9), and counting the number of overall issues and security issues in each sub-period. We plot the average issues and security issues in each sub-period for each group in Figure 5.8, along with the average number of pull requests and total changes. We see that all three groups had similar maturation patterns over the pre-test periods, but the number of issues and security issues in the control group stays constant during the post-test period. This could indicate a selection bias that might artificially make the treatment outcomes appear worse—our control group may contain repositories that became increasingly less active, masking an actual positive effect of code review on software quality. Additionally, if this is the case, we may be underestimating the magnitude of the effect of code review on software security. Finally, looking at the maturation of changes and pull requests over time, we see consistent changes across all three groups, but, as might be expected, implementing code review involves a shift from direct pushes to using pull requests, so we see a sharp growth in the rate of pull requests for our treatment groups.

Quantifier reliability

We also use the divided pre-test and post-test measurements of security issues to measure the reliability of our quantifier. We estimate the number of security issues in each sub-period, and then compare their sum to the estimate over the entire period. This split-half reliability gives us an estimate of the reliability of our quantification estimates over the various changepoint splits used in our analysis. If our quantifier is highly reliable, we would expect highly consistent estimates when the changepoint is

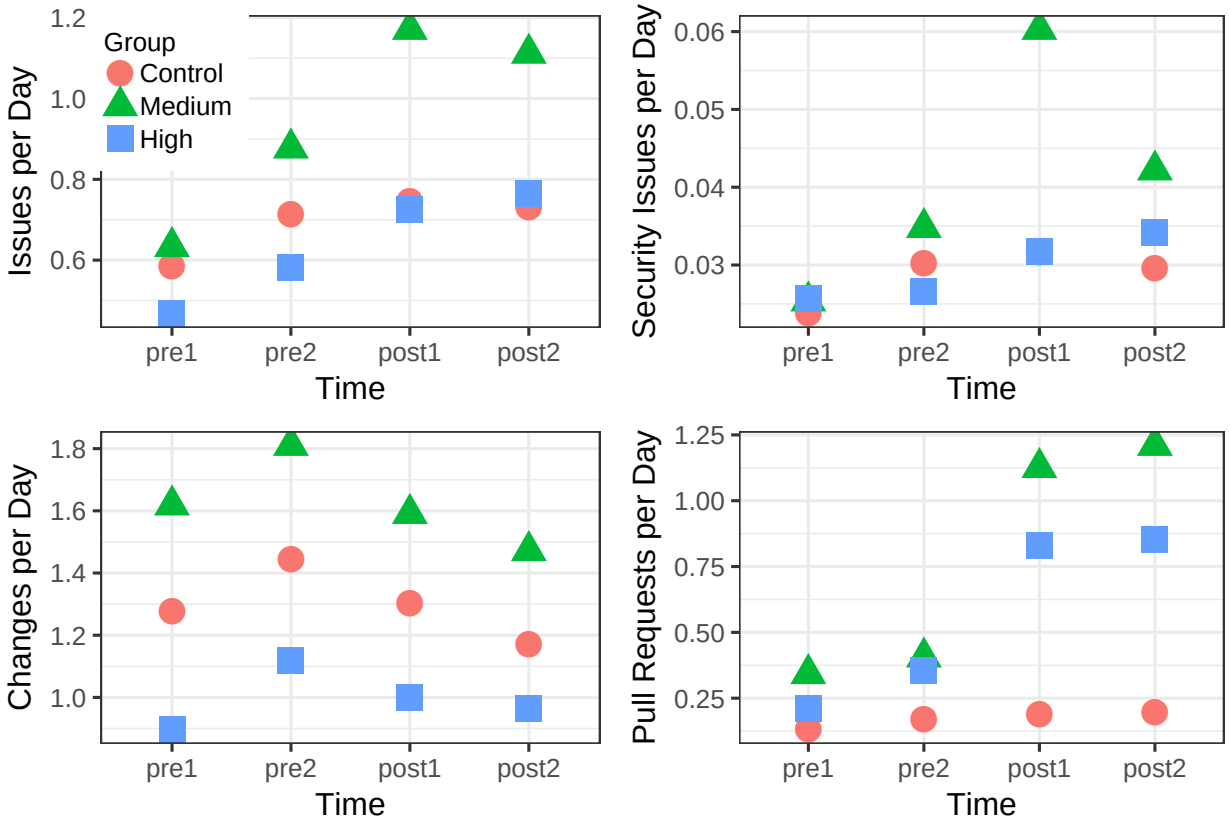

Figure 5.8: Covariates by group, over four sub-periods (two halves of the pre-test period and two halves of the post-test period). Each covariate is normalized by the number of days in the sub-period. We see similar development maturation in the pre-test across groups, although the medium treatment group is consistently the most active, with the most overall issues and security issues per day.

Table 5.9: Distribution of days in the pre-test and post-test periods, by group.

Days in period	Mean	Min	25%	Median	75%	Max
Pre-test	472.8	209	471	471	471	813
Control	471	471	471	471	471	471
Medium	485.7	209	433	458.5	547	813
High	527.6	346	447	508	611.8	764
Post-test	438.2	98	440	440	440	702
Control	440	440	440	440	440	440
Medium	425.3	98	364	452.5	478	702
High	383.4	147	299.2	403	464	565

only shifted slightly in time. Comparing the split-half-and-combine measurements to our full period estimates, we find a mean absolute error of 0.44 issues for the pre-test period and 1.17 issues for the post-test period. The distributions of these errors are summarized in Table 5.10.

Table 5.10: Distribution of the error of our split-half-and-combine measurements compared to the full period estimates.

	Mean	Min	25%	Median	75%	Max
Absolute Error						
Pre-Test	0.4404	0	0	0	1	11
Post-Test	1.171	0	0	1	1	173
Relative Error						
Pre-Test	−0.0971	−11	0	0	0	10
Post-Test	−0.9199	−173	−1	0	0	19

External validity

We draw our treatment and control groups from a wide population of GitHub repositories, as in our regression models in Chapter 4 (see Section 4.4 for more about possible validity concerns of using GitHub repositories as our population). It is possible that the effects we found only hold at the specific variation of treatment found in our sample. The effects might not hold for other levels or manners of code review. It is also possible that the effect only holds for issues reported to a GitHub project. Other defects, bugs, or security vulnerabilities, in other software or platforms, may not have the same causal relationship with code review as measured in this work.

Treatment effects

Our treatment group repositories often have at most partial treatment application (that is, they do not have perfect code review coverage). Additionally, our control group is not necessarily devoid of any code review. However, our experimental design can still detect the effects of a *difference* in treatment levels. We hypothesize that code review becomes most useful when applied consistently within a project, but our results do not allow us to determine what level of consistency is required to see an effect. At best, we can determine what level is *sufficient* for an effect to be visible.

5.6 Discussion

While our treatment analysis generally aligns with the associations we found in Chapter 4, our results paint a more complex picture of the relationship between code review and software quality and security. Our results seem to indicate that code review may (confoundingly) cause an increase in the number of issues reported to a project. This could happen for a variety of reasons—one plausible explanation is that code review is linked to increased engagement and participation in general, which leads to more users and more people reporting more bugs (rather than this being a fundamental increase in defects). Additionally, code review itself could be correlated with increased use of the issue tracker for project management purposes, which we would be unable to detect.

However, despite this potential increased use and reporting, we found a *negative* treatment effect on security issues. We might expect that the real effect on the security of the software to be even greater, as the expansion of reporting may suppress the effect in our analysis. In general, our results provide evidence that implementing code review can reduce both overall defects and security bugs for some projects, but these benefits may be limited to particularly defect-prone projects.

Additionally, the variance of outcomes for our treatment groups were very high, indicating that projects should implement code review *intentionally* and *thoughtfully*. Rather than treating code review as a panacea that will eliminate bugs, it may be better to think of it (as any other process to be implemented) as something to be tailored to the particular project and developers, and then measured and tested as it is used. This applies to both the benefits and costs of adding additional process. Each project will have different costs for performing code review (although these costs may go down over time as developers become more comfortable with it) and costs for having bugs and security vulnerabilities. It can also be useful to consider modern code review as only a part of a broad secure software development life cycle (which would include other processes such as design review, testing regimes, and response processes).

There are existing tools and research that attempt to reduce the costs of modern code review. Our analysis of several existing automated code review services found that adding these tools can greatly decrease overall issues compared to projects that did not use them. While we did not find any effect on security issues, the value proposition for automated code review tools is particularly great. The costs that concerned Czerwonka et al. [15] (particularly in terms of time spent to perform reviews and turnaround time for a developer to be blocked waiting to receive reviews) could be dramatically improved with the use of such automated tools. Our results provide some evidence for the potential effectiveness of improving code review more broadly through the development of supporting tools. However, our treatment group only included 8 cases, limiting our ability to generalize this result.

More broadly, we feel that our changepoint detection technique could be applicable

to a variety of observational studies. The main constraint is that there need to be a sufficient number of needles in the proverbial haystack to be able to have the statistical power necessary to detect potentially small effects (or effects with high variance).

Conclusions

Our work in this dissertation takes a first step to quantify, on a generalizable sample of open source software projects, the effects of code review practices on software quality and security. Focusing on code review coverage and participation, we found broadly generalizable correlations between code review practices and both software quality and security. We then took our association results a step further by using timeseries archival data to test for causal treatment effects of implementing modern code review in software projects on their quality and security, finding that code review does appear to improve software security, but it may actually increase the number of overall issues reported to a project. Further work is needed to determine the "dosage" levels for code review practices in order to see effects on software quality and security. Our research supports the practices of mandatory and thorough code review for defect-prone or security-sensitive components, but we caution that code review should not be taken as a kind of panacea for bugs. Individual projects should implement code review thoughtfully, measuring its impact and benefits, and its costs on developer productivity. Our hope is that our estimation of the treatment effects of code review process as an intervention for both software quality and security will help development teams make more informed and economically motivated decisions about their development processes.

We were able to perform our large-scale studies thanks to our quantification techniques, without which we would not have been able to label the large datasets we worked with. Our novel techniques for implementing and optimizing quantifiers using neural networks could be useful for any quantitative study where manual labeling of quantities of interest is prohibitively expensive at scale.

Finally, our research supports the development of tools to support code review. As we argued in the beginning of this dissertation, both the costs and effectiveness of code review matter for its adoption by development teams. Adding new processes involves investment in both time and cost. We believe that our analysis of automated code review services provides evidence that automation, even if it only detects "low-hanging fruit" defects, can improve software quality. While we only analyzed the use of a small number of relatively simple automated tools, more complex tools and services, particularly ones focused on software security, could have a great impact on software security while maintaining low investment costs.

Bibliography

[1] M. Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015 (cit. on p. 13).

[2] A. Aurum, H. Petersson, and C. Wohlin. "State-of-the-art: software inspections after 25 years". In: *Software Testing, Verification and Reliability* 12.3 (Sept. 2002), pp. 133–154. DOI: 10.1002/stvr.243 (cit. on pp. 1, 4).

[3] A. Bacchelli and C. Bird. "Expectations, outcomes, and challenges of modern code review". In: *Proceedings of the 35th International Conference on Software Engineering*. ICSE '13. May 2013, pp. 712–721. DOI: 10.1109/ICSE.2013.6606617 (cit. on pp. 1, 4).

[4] J. Barranquero, J. Díez, and J. J. del Coz. "Quantification-oriented learning based on reliable classifiers". In: *Pattern Recognition* 48.2 (2015), pp. 591–604. DOI: 10.1016/j.patcog.2014.07.032 (cit. on p. 12).

[5] M. Basseville, I. V. Nikiforov, et al. *Detection of abrupt changes: theory and application*. Vol. 104. Prentice Hall Englewood Cliffs, 1993 (cit. on pp. 36, 37).

[6] A. Bella, C. Ferri, J. Hernandez-Orallo, and M. J. Ramirez-Quintana. "Quantification via Probability Estimators". In: *Proceedings of the 2010 IEEE International Conference on Data Mining*. ICDM '10. Dec. 2010, pp. 737–742. DOI: 10.1109/ICDM.2010.75 (cit. on p. 12).

[7] M. Bernhart and T. Grechenig. "On the understanding of programs with continuous code reviews". In: *Proceedings of the 21st International Conference on Program Comprehension*. ICPC '13. IEEE, May 2013, pp. 192–198. DOI: 10.1109/ICPC.2013.6613847 (cit. on p. 4).

[8] A. Bosu, M. Greiler, and C. Bird. "Characteristics of Useful Code Reviews: An Empirical Study at Microsoft." In: *Proceedings of the 12th ACM/IEEE Working Conference on Mining Software Repositories*. MSR '15. May 2015, pp. 146–156. DOI: 10.1109/MSR.2015.21 (cit. on p. 7).

[9] D. Bowes, T. Hall, and J. Petrić. "Software defect prediction: do different classifiers find the same defects?" In: *Software Quality Journal* (2017), pp. 1–28. DOI: 10.1007/s11219-016-9353-3 (cit. on p. 34).

[10] F. Camilo, A. Meneely, and M. Nagappan. “Do Bugs Foreshadow Vulnerabilities? A Study of the Chromium Project”. In: *Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories*. MSR ’15. IEEE, 2015, pp. 269–279. DOI: 10.1109/MSR.2015.32 (cit. on pp. 1, 6).

[11] J. C. Cappelleri and W. M. Trochim. “Ethical and scientific features of cutoff-based designs of clinical trials: a simulation study”. In: *Medical Decision Making* 15.4 (1995), pp. 387–394 (cit. on p. 36).

[12] T. Chen and C. Guestrin. “XGBoost: A Scalable Tree Boosting System”. In: *CoRR* abs/1603.02754 (2016). URL: http://arxiv.org/abs/1603.02754 (cit. on p. 18).

[13] Chromium browser project. *Code Reviews*. URL: https://chromium.googlesource.com/chromium/src/+/master/docs/code_reviews.md (cit. on p. 4).

[14] T. D. Cook. “Waiting for Life to Arrive: A history of the regression-discontinuity design in Psychology, Statistics and Economics”. In: *Journal of Econometrics* 142.2 (2008). The regression discontinuity design: Theory and applications, pp. 636–654. DOI: 10.1016/j.jeconom.2007.05.002 (cit. on p. 36).

[15] J. Czerwonka, M. Greiler, and J. Tilford. “Code Reviews Do Not Find Bugs: How the Current Code Review Best Practice Slows Us Down”. In: *Proceedings of the 37th International Conference on Software Engineering*. ICSE ’15. Florence, Italy: IEEE, 2015, pp. 27–28. DOI: 10.1109/ICSE.2015.131 (cit. on pp. 1, 55).

[16] A. Edmundson, B. Holtkamp, E. Rivera, M. Finifter, A. Mettler, and D. Wagner. “An Empirical Study on the Effectiveness of Security Code Review”. In: *Proceedings of the 5th International Symposium on Engineering Secure Software and Systems*. ESSoS ’13. 2013, pp. 197–212. DOI: 10.1007/978-3-642-36563-8_14 (cit. on p. 6).

[17] A. Esuli and F. Sebastiani. “Optimizing Text Quantifiers for Multivariate Loss Functions”. In: *ACM Transactions on Knowledge Discovery from Data* 9.4 (June 2015), 27:1–27:27. DOI: 10.1145/2700406 (cit. on pp. 12, 13).

[18] M. O. Finkelstein, B. Levin, and H. Robbins. “Clinical and prophylactic trials with assured new treatment for those at greater risk: I. A design proposal.” In: *American Journal of Public Health* 86.5 (1996), pp. 691–695 (cit. on p. 36).

[19] G. Forman. “Counting positives accurately despite inaccurate classification”. In: *Proceedings of the 16th European Conference on Machine Learning*. ECML ’05. Springer Berlin Heidelberg, 2005, pp. 564–575. DOI: 10.1007/11564096_55 (cit. on pp. 10, 12).

[20] W. Gao and F. Sebastiani. “Tweet sentiment: from classification to quantification”. In: *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. ASONAM ’15. 2015, pp. 97–104 (cit. on pp. 10, 13).

[21] M. Gegick, P. Rotella, and L. Williams. “Predicting Attack-prone Components”. In: *Proceedings of the 6th International Conference on Software Testing, Verification and Validation*. ICST ’09. IEEE, Apr. 2009, pp. 181–190. DOI: 10.1109/ICST.2009.36 (cit. on p. 11).

[22] M. Gegick, P. Rotella, and T. Xie. “Identifying security bug reports via text mining: An industrial case study”. In: *Proceedings of the 7th International Working Conference on Mining Software Repositories*. MSR ’10. May 2010, pp. 11–20. DOI: 10.1109/MSR.2010.5463340 (cit. on p. 11).

[23] A. Gelman and Z. Huang. “Estimating Incumbency Advantage and Its Variation, as an Example of a Before–After Study”. In: *Journal of the American Statistical Association* 103.482 (2008), pp. 437–446. DOI: 10.1198/016214507000000626 (cit. on p. 36).

[24] B. Ghotra, S. McIntosh, and A. E. Hassan. “Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models”. In: *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering*. ICSE ’15. IEEE, 2015, pp. 789–800. DOI: 10.1109/ICSE.2015.91 (cit. on p. 34).

[25] GitHub. *GitHub API*. URL: https://developer.github.com/v3/ (cit. on p. 23).

[26] GitHub. *GitHub Linguist*. URL: https://github.com/github/linguist (cit. on p. 23).

[27] Google. *BigQuery*. URL: https://cloud.google.com/bigquery/ (cit. on p. 23).

[28] I. Grigorik. *GitHub Archive*. URL: https://www.githubarchive.org/ (cit. on pp. 23, 37).

[29] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. “A Systematic Literature Review on Fault Prediction Performance in Software Engineering”. In: *IEEE Transactions on Software Engineering* 38.6 (Nov. 2012), pp. 1276–1304. DOI: 10.1109/TSE.2011.103 (cit. on p. 11).

[30] C. Heitzenrater and A. Simpson. “A Case for the Economics of Secure Software Development”. In: *Proceedings of the 2016 New Security Paradigms Workshop*. NSPW ’16. Granby, Colorado, USA: ACM, 2016, pp. 92–105. DOI: 10.1145/3011883.3011884 (cit. on p. 1).

[31] D. J. Hopkins and G. King. “A method of automated nonparametric content analysis for social science”. In: *American Journal of Political Science* 54.1 (2010), pp. 229–247. DOI: 10.1111/j.1540-5907.2009.00428.x (cit. on p. 10).

[32] G. W. Imbens and J. M. Wooldridge. “Recent Developments in the Econometrics of Program Evaluation”. In: *Journal of Economic Literature* 47.1 (Mar. 2009), pp. 5–86. DOI: 10.1257/jel.47.1.5 (cit. on p. 36).

[33] B. A. Jacob and L. Lefgren. “Remedial education and student achievement: A regression-discontinuity analysis”. In: *The review of Economics and Statistics* 86.1 (2004), pp. 226–244 (cit. on p. 36).

[34] T. Joachims, T. Finley, and C.-N. J. Yu. “Cutting-plane training of structural SVMs”. In: *Machine Learning* 77.1 (2009), pp. 27–59. DOI: 10.1007/s10994-009-5108-8 (cit. on p. 13).

[35] T. Joachims, T. Hofmann, Y. Yue, and C.-N. Yu. “Predicting structured objects with support vector machines”. In: *Communications of the ACM* 52.11 (Nov. 2009), pp. 97–104. DOI: 10.1145/1592761.1592783 (cit. on p. 13).

[36] C. F. Kemerer and M. C. Paulk. “The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data”. In: *IEEE Transactions on Software Engineering* 35.4 (July 2009), pp. 534–550. DOI: 10.1109/TSE.2009.27 (cit. on p. 7).

[37] R. Killick, K. Haynes, and I. A. Eckley. *changepoint: An R package for changepoint analysis*. R package version 2.2.2. 2016. URL: https://CRAN.R-project.org/package=changepoint (cit. on pp. 37, 38).

[38] R. Kohavi. “A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection”. In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence*. IJCAI ’95. Montreal, Quebec, Canada: Morgan Kaufmann Publishers Inc., 1995, pp. 1137–1143 (cit. on p. 12).

[39] G. Kudrjavets, N. Nagappan, and L. Williams. “On the Effectiveness of Unit Test Automation at Microsoft”. In: *Proceedings of the 20th International Symposium on Software Reliability Engineering*. ISSRE ’09. IEEE, 2009, pp. 81–89. DOI: 10.1109/ISSRE.2009.32 (cit. on p. 36).

[40] E. Loper and S. Bird. “NLTK: The Natural Language Toolkit”. In: *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics*. ETMTNLP ’02. Software available from nltk.org. July 2002, pp. 63–70. DOI: 10.3115/1118108.1118117 (cit. on p. 15).

[41] P. McCullagh and J. Nelder. *Generalized Linear Models*. 2nd. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. Taylor & Francis, 1989 (cit. on p. 28).

[42] 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: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR ’14. May 2014, pp. 192–201. DOI: 10.1145/2597073.2597076 (cit. on pp. 1, 2, 6, 27, 29, 34).

[43] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates. "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits". In: *Proceedings of the 7th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '13. IEEE, Oct. 2013, pp. 65–74. DOI: 10.1109/ESEM.2013.19 (cit. on pp. 1, 6).

[44] A. Meneely, A. C. R. Tejeda, B. Spates, S. Trudeau, D. Neuberger, K. Whitlock, C. Ketant, and K. Davis. "An empirical investigation of socio-technical code review metrics and security vulnerabilities". In: *Proceedings of the 6th International Workshop on Social Software Engineering*. SSE '14. ACM, Nov. 2014, pp. 37–44. DOI: 10.1145/2661685.2661687 (cit. on pp. 2, 6, 7, 27, 31, 34).

[45] L. S. Meyers, G. Gamst, and A. J. Guarino. *Applied Multivariate Research: Design and Interpretation*. 1st. Sage, 2006 (cit. on p. 28).

[46] L. Milli, A. Monreale, G. Rossetti, F. Giannotti, D. Pedreschi, and F. Sebastiani. "Quantification Trees". In: *Proceedings of the 13th IEEE International Conference on Data Mining*. ICDM '13. IEEE, Dec. 2013, pp. 528–536. DOI: 10.1109/ICDM.2013.122 (cit. on p. 12).

[47] J. C. Munson and S. G. Elbaum. "Code churn: a measure for estimating the impact of code change". In: *Proceedings of the 1998 International Conference on Software Maintenance*. ICSM '98. 1998, pp. 24–31. DOI: 10.1109/ICSM.1998.738486 (cit. on p. 26).

[48] N. Nagappan and T. Ball. "Use of relative code churn measures to predict system defect density". In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE '05. IEEE, May 2005, pp. 284–292. DOI: 10.1109/ICSE.2005.1553571 (cit. on p. 26).

[49] N. Nagappan and T. Ball. "Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study". In: *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*. ESEM '07. Sept. 2007, pp. 364–373. DOI: 10.1109/ESEM.2007.13 (cit. on p. 26).

[50] P. Nakov, A. Ritter, S. Rosenthal, V. Stoyanov, and F. Sebastiani. "SemEval-2016 Task 4: Sentiment Analysis in Twitter". In: *Proceedings of the 10th International Workshop on Semantic Evaluation*. 2016 (cit. on p. 13).

[51] A. Ozment and S. E. Schechter. "Milk or Wine: Does software security improve with age?" In: *Proceedings of the 15th USENIX Security Symposium*. Security '15. USENIX Association, 2006. URL: https://www.usenix.org/conference/15th-usenix-security-symposium/milk-or-wine-does-software-security-improve-age (cit. on pp. 6, 26).

[52] E. Page. "On problems in which a change in a parameter occurs at an unknown point". In: *Biometrika* 44.1/2 (1957), pp. 248–252 (cit. on p. 37).

[53] Princeton University. *About WordNet*. 2010. URL: http://wordnet.princeton.edu (cit. on p. 15).

[54] M. M. Rahman, C. K. Roy, and J. A. Collins. "CoRReCT: Code Reviewer Recommendation in GitHub Based on Cross-project and Technology Experience". In: *Proceedings of the 38th International Conference on Software Engineering Companion*. ICSE '16. ACM, May 2016, pp. 222–231. DOI: 10.1145/2889160.2889244 (cit. on p. 7).

[55] B. Ray, D. Posnett, V. Filkov, and P. Devanbu. "A large scale study of programming languages and code quality in GitHub". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE '14. Nov. 2014, pp. 155–165. DOI: 10.1145/2635868.2635922 (cit. on pp. 8, 25, 28).

[56] P. Rigby, B. Cleary, F. Painchaud, M.-A. Storey, and D. German. "Contemporary peer review in action: Lessons from open source development". In: *IEEE Software* 29.6 (Nov. 2012), pp. 56–61. DOI: 10.1109/MS.2012.24 (cit. on p. 4).

[57] G. Rodríguez. *Lecture Notes on Generalized Linear Models*. 2007. URL: http://data.princeton.edu/wws509/notes/ (cit. on p. 28).

[58] A. Ruef, M. Hicks, J. Parker, D. Levin, M. L. Mazurek, and P. Mardziel. "Build It, Break It, Fix It: Contesting Secure Development". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. ACM, 2016, pp. 690–703. DOI: 10.1145/2976749.2978382 (cit. on pp. 9, 35).

[59] G. Salton and C. Buckley. "Term-weighting approaches in automatic text retrieval". In: *Information Processing & Management* 24.5 (1988), pp. 513–523. DOI: 10.1016/0306-4573(88)90021-0 (cit. on p. 15).

[60] W. R. Shadish, T. D. Cook, and D. T. Campbell. *Experimental and Quasi-experimental Designs for Generalized Causal Inference*. 2nd ed. Houghton Mifflin, 2002 (cit. on p. 42).

[61] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities". In: *IEEE Transactions on Software Engineering* 37.6 (Nov. 2011), pp. 772–787. DOI: 10.1109/TSE.2010.81 (cit. on p. 26).

[62] L. Szekeres, M. Payer, T. Wei, and D. Song. "SoK: Eternal War in Memory". In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. S&P '13. IEEE, 2013, pp. 48–62. DOI: 10.1109/SP.2013.13 (cit. on pp. 23, 26).

[63] C. Thompson. *Dataset for Large-Scale Analysis of Modern Code Review Practices and Software Security in Open Source Software*. 2017. DOI: 10.6078/D13M3J (cit. on pp. 3, 17).

[64] C. Thompson and D. Wagner. “A Large-Scale Study of Modern Code Review and Security in Open Source Projects”. In: *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. PROMISE ’17. Toronto, Canada: ACM, 2017. DOI: 10.1145/3127005.3127014 (cit. on pp. 10, 22).

[65] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. “Investigating code review practices in defective files: an empirical study of the Qt system”. In: *Proceedings of the 12th ACM/IEEE Working Conference on Mining Software Repositories*. MSR ’15. May 2015, pp. 168–179. DOI: 10.1109/MSR.2015.23 (cit. on pp. 1, 2, 6).

[66] W. M. Trochim. *The Research Methods Knowledge Base, 2nd Edition*. Oct. 2006. URL: http://www.socialresearchmethods.net/kb/ (cit. on p. 42).

[67] B. Vasilescu, S. van Schuylenburg, J. Wulms, A. Serebrenik, and M. G. J. van den Brand. “Continuous integration in a social-coding world: Empirical evidence from GitHub. **Updated version with corrections*”. In: *CoRR* abs/1512.01862 (2015). URL: http://arxiv.org/abs/1512.01862 (cit. on p. 7).

[68] S. Zaman, B. Adams, and A. E. Hassan. “Security Versus Performance Bugs: A Case Study on Firefox”. In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. MSR ’11. ACM, May 2011, pp. 93–102. DOI: 10.1145/1985441.1985457 (cit. on pp. 1, 6).

Appendix A

Full regression models

The tables in this appendix show our full negative binomial regression models with untransformed coefficients and standard errors.

- Table A.1 shows the full models from our association analysis in Chapter 4.
- Table A.2 shows the full models with untransformed coefficients and standard errors from our main treatment analysis in Chapter 5.
- Table A.3 shows the full models with untransformed coefficients and standard errors from our treatment analysis with a single combined treatment group in Chapter 5.
- Table A.4 shows the full models for our automated code review treatment analysis in Chapter 5.

Table A.1: Full association analysis models. We report here the full coefficients and model parameters for each of our four negative binomial regression models from Chapter 4. The coefficients here are the untransformed parameters, along with their standard errors.

	Dependent variable			
	Issues		Security Issues	
	(1)	(2)	(3)	(4)
(Intercept)	3.635**	3.678***	6.822e−1***	7.247e−1***
	(6.185e−2)	(6.301e−3)	(6.980e−2)	(7.103e−2)
code size (KB)	1.368e−6	1.490e−6	−2.040e−7	−3.215e−7
	(9.761e−7)	(9.796e−7)	(1.136e−6)	(1.145e−6)
churn (KLOC)	8.242e−6	5.715e−5**	7.898e−6	3.254e−5
	(2.276e−5)	(2.183e−5)	(2.305e−5)	(2.203e−5)
age (days)	4.167e−4***	4.070e−4***	6.958e−5	7.039e−5
	(3.482e−5)	(3.543e−5)	(3.899e−5)	(3.962e−5)
contributors	2.995e−3***	2.631e−3***	−3.213e−4	−5.048e−4
	(6.400e−4)	(6.456e−4)	(7.073e−4)	(7.153e−4)
stars	2.359e−4***	2.271e−4***	−8.253e−5***	−8.349e−5***
	(1.597e−5)	(1.602e−5)	(1.837e−5)	(1.841e−5)
pull requests	1.681e−3***	2.050e−3***	−1.095e−4	6.317e−5
	(7.188e−5)	(6.517e−5)	(8.842e−5)	(7.693e−5)
memory safety	1.391e−1**	1.521e−1**	5.887e−2	6.258e−2
	(5.165e−2)	(5.176e−2)	(5.817e−2)	(5.825e−2)
unreviewed pull requests	2.971e−3***		1.686e−3***	
	(4.099e−4)		(4.271e−4)	
commenters per pr		−3.650e−2		−5.263e−2
		(2.839e−2)		(3.191e−2)
review comments per pr		−9.571e−2*		−5.146e−2
		(4.496e−2)		(5.860e−2)
n		2881		
θ	0.909	0.906	0.927	0.924
	(0.021)	(0.021)	(0.032)	(0.031)
Null deviance	5561.800	5542.200	5898.900	5586.100
Null d.f.		2880		
Residual deviance	3326.900	3328.500	3086.200	3085.800
Residual d.f.	2872	2871	2871	2870
AIC	32 610	32 626	14 517	14 524
2*log-likelihood	−32 590.387	−32 603.705	−14 494.788	−14 500.216

* $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

Table A.2: Full treatment analysis models, for three groups. We report here the full coefficients and model parameters for both negative binomial regression models for three groups from Chapter 5. The coefficients here are the untransformed parameters, along with their standard errors.

	Dependent variable	
	Issues	Security Issues
	(1)	(2)
(Intercept)	6.359***	2.363***
	(1.916e−1)	(2.889e−1)
code size (KB)	2.371e−6***	1.503e−6
	(6.504e−7)	(9.699e−7)
age (days)	−1.233e+1***	−1.450e+1***
	(2.016)	(3.101)
stars	1.526e−5***	5.333e−6
	(3.601e−6)	(5.481e−6)
memory safety	9.895e−2**	1.480e−1**
	(3.476e−2)	(5.363e−2)
pre-test		
issues	1.329e−3***	1.998e−4*
	(4.433e−5)	(7.913e−5)
security issues		3.436e−2***
		(1.504e−3)
pushes	1.171e−4***	3.808e−5
	(2.193e−5)	(3.253e−5)
pull requests	1.127e−4	7.870e−4***
	(1.285e−4)	(1.974e−4)
review coverage	−2.474e−1	−1.455***
	(2.405e−1)	(3.698e−1)
days	−2.653e−3***	−1.004e−3
	(3.914e−4)	(5.892e−4)
medium treatment	4.562e−1***	6.569e−1***
	(6.690e−2)	(9.594e−2)
medium:pre-test	−4.628e−4***	−2.250e−2***
	(1.016e−4)	(3.333e−3)
high treatment	−9.929e−2	1.136e−1
	(1.277e−1)	(1.701e−1)
high:pre-test	7.500e−4*	−6.382e−3
	(3.280e−4)	(6.798e−3)
n	1720	
θ	3.438	1.759
	(0.114)	(0.070)
Null deviance	4675.300	3498.200
Null d.f.	1719	1719
Residual deviance	1794.200	1858.800
Residual d.f.	1706	1705
AIC	21 364	11 217
2*log-likelihood	−21 334.379	−11 184.519

* $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

Table A.3: Full treatment analysis models, for two groups. We report here the full coefficients and model parameters for both negative binomial regression models for two groups from Chapter 5. The coefficients here are the untransformed parameters, along with their standard errors.

	Dependent variable	
	Issues	Security Issues
	(1)	(2)
(Intercept)	6.460***	2.527***
	(1.895e−1)	(2.861e−1)
code size (KB)	2.338e−6***	1.466e−6
	(6.525e−7)	(9.729e−7)
age (days)	−1.227e+1***	−1.450e+1***
	(2.020)	(3.106)
stars	1.514e−5***	5.501e−6
	(3.612e−6)	(5.493e−6)
memory safety	9.612e−2**	1.450e−1**
	(3.485e−2)	(5.372e−2)
pre-test		
issues	1.333e−3***	1.957e−4*
	(4.445e−5)	(7.922e−5)
security issues		3.445e−2***
		(1.504e−3)
pushes	1.134e−4***	3.720e−5
	(2.200e−5)	(3.260e−5)
pull requests	1.285e−4	7.630e−4***
	(1.288e−4)	(1.951e−4)
review coverage	−3.293e−1	−1.516***
	(2.398e−1)	(3.683e−1)
days	−2.850e−3***	−1.333e−3*
	(3.872e−4)	(5.838e−4)
treatment	3.269e−1***	5.043e−1***
	(5.987e−2)	(8.608e−2)
treatment:pre-test issues	−2.063e−4***	
	(9.814e−5)	
treatment:pre-test security issues		−1.612e−2***
		(3.051e−3)
n	1720	
θ	3.416	1.750
	(0.113)	(0.070)
Null deviance	4645.500	3483.000
Null d.f.	1719	1719
Residual deviance	1794.700	1858.300
Residual d.f.	1708	1707
AIC	21 372	11 219
2*log-likelihood	−21 346.304	−11 191.494

* $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

Table A.4: Full treatment analysis models for automated code review usage. We report here the full coefficients and model parameters for both negative binomial regression models for automated code review usage from Chapter 5. The coefficients here are the untransformed parameters, along with their standard errors.

	Dependent variable	
	Issues	Security Issues
	(1)	(2)
(Intercept)	7.163***	2.821
	(1.829)	(3.628)
code size (KB)	2.467e−6***	3.431e−6***
	(5.330e−7)	(7.626e−7)
age (days)	−1.698e+1***	−2.098e+1***
	(1.454)	(2.192)
stars	1.310e−5***	2.417e−7
	(1.571e−6)	(2.299e−6)
memory safety	7.981e−2**	1.413e−1***
	(2.691e−2)	(4.041e−2)
pre-test		
issues	1.226e−3***	2.144e−4***
	(2.741e−5)	(4.983e−5)
security issues		2.781e−2***
		(8.851e−4)
pushes	1.589e−4***	1.370e−4***
	(1.752e−5)	(2.509e−5)
pull requests	−2.002e−4***	−2.647e−4***
	(3.564e−5)	(5.109e−5)
review coverage	5.825e−1***	4.274e−1***
	(8.076e−2)	(1.195e−1)
days	−4.163e−3	−1.750e−3
	(3.882e−3)	(7.702e−3)
treatment	−1.617**	−1.403
	(5.672e−1)	(9.003e−1)
treatment:pre-test issues	1.126e−3	
	(1.033e−3)	
treatment:pre-test security issues		1.251e−2
		(2.760e−2)
treatment:pre-test review coverage	2.423	1.679
	(2.480)	(3.702)
n	3543	
θ	2.986	1.552
	(0.068)	(0.041)
Null deviance	9718.200	7254.200
Null d.f.	3542	3542
Residual deviance	3722.500	3833.100
Residual d.f.	3530	3529
AIC	45 177	23 934
2*log-likelihood	−45 149.220	−23 903.897

* $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$