ABSTRACT

MENEELY, ANDREW PHILIP. Investigating the Relationship between Developer Collaboration and Software Security. (Under the direction of Dr. Laurie Williams).

With each new developer to a software development team comes a greater challenge to manage the communication, coordination, and knowledge transfer amongst teammates. Lack of team cohesion, miscommunications, and misguided effort can lead to all kinds of problems, including security vulnerabilities. In this dissertation research, we focus on examining the statistical relationships between development team structure and security vulnerabilities. The statistical relationships demonstrated in this research provide us with (a) predictive models for finding security vulnerabilities in software prior to its release; and (b) insight into how effective software development teams are organized.

This dissertation is comprised of three research projects surrounding what we call developer activity metrics. Mostly based on social network analysis, developer activity metrics are designed to quantify how groups of software developers are working with each other. Developer activity data come from software development artifacts that provide information such as version control change logs and issue tracking systems. The developer activity data is transformed into a developer network designed to represent the socio-technical organization of labor in a team, specifically “who is working with whom” within the scope of a given development project. The three research projects are as follows: Security Correlation Study. We applied social network analysis techniques to three open source
software products, and discovered a consistent statistical association between metrics measuring developer activity and post-release security vulnerabilities.

- **Perception Corroboration Study.** We surveyed developers from the same three open source projects and found that developers’ perceptions of collaboration and expertise corroborate evidence of collaboration and expertise in developer activity metrics.

- **Synthesis Study.** We gathered the results from the related work both inside software engineering and in the field of socio-technical research in general. We synthesized our results into a single paradigm with conjectures for socio-technical research in software engineering.

These three research project have resulted in the following findings:

- Source code files changed by many developers (in our case studies, 6 developers or more) are more likely to have at least one post-release security vulnerability.

- Vulnerability prediction models based on developer activity metrics can be used across different software development projects.

- If two developers change the same source code within the same month, they typically perceive they are collaborating with each other.

- The degree of separation between two developers in a developer network typically represents their perceived socio-technical distance.

Having a high centrality in a developer network is associated with being reputed as being a project expert.
Investigating the Relationship between Developer Collaboration and Software Security

by
Andrew Meneely

A dissertation submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the degree of Doctor of Philosophy

in Computer Science

Raleigh, North Carolina
2011

APPROVED BY:

Laurie Williams
Committee Chair

Tao Xie

Annie Antón

Jason Osborne
DEDICATION

To my wife, Kelly, for listening to me talk all this out.
BIOGRAPHY

Andy Meneely is just a guy who loves studying Computer Science, and has been doing it for over half of his life.

Andy first became interested in Computer Science at the age of 13, when he learned the BASIC programming language. By the end of high school, Andy was lost in oceans of C++ code and never looked back. He attended Calvin College and double-majored in Computer Science and Mathematics, and did some very cool independent studies with even cooler professors on compilers, ray tracers, genetic algorithms, and Computer Science education. After Calvin, Andy attended North Carolina State University for his Master’s and Doctoral work. There he found himself immersed in the wonderful world of research. He has been a researcher with companies such as Nortel, Red Hat, IBM, Applied Research Associates, and Cisco. Andy is also a member of NCSU’s Realsearch group, led by Laurie Williams. Andy will be an assistant professor in the software engineering department at the Rochester Institute of Technology, starting in the fall of 2011.
ACKNOWLEDGMENTS

Metrics can provide an abstract, impartial view of complex objects in a complex world. Sadly, no metric can possibly quantify my gratitude for the massive network of support I have received over the last five years. So, I will summarize my thanks in the following plain English bulleted list. My deepest thanks:

- To Kelly, for her unending patience, for being my compass for reality, and for being married to a poor grad student all these years;
- To Laurie, for her amazing feedback, inspiring drive, and uncanny ability to find funding;
- To Dad, for his contagious passion for knowledge and steadfast dedication to the search for God’s truth;
- To Ben, for giving me faith in collaboration and for challenging me every moment;
- To Lucas, for his patient mentoring and his skepticism;
- To Mei, for sharing in my experience at every step of this crazy journey;
- To Aaron, for his refreshing perspective on both software engineering and on life;
- To the rest of the Realsearch group, past and present (e.g. Mark, Yonghee, Sarah, Jiang, Dright, Michael, John, Andrew, Raza, Jerrod, Eric), for their continually perceptive comments;
- To Mom, Alison, my committee, and to anyone else who has given me a befuddled look when I am trying to explain something. Thank you for pushing me to re-think and to re-articulate. Your bewilderment has helped me more than you know.
To Stephen Colbert, for his devotion to truthiness and for sharing in my fear of bears.

Thank you so much to all of you.
# TABLE OF CONTENTS

LIST OF TABLES .............................................................................................................. x

LIST OF FIGURES .......................................................................................................... xi

1 Introduction .................................................................................................................. 1

2 Background .................................................................................................................. 7

2.1 Network Analysis ..................................................................................................... 7

2.2 Software Development Terminology ....................................................................... 8

2.3 Security Terminology ............................................................................................... 8

2.4 Statistical Techniques ............................................................................................... 9

3 Related Work ............................................................................................................... 12

3.1 Prior Work ................................................................................................................ 12

3.2 Quantifying Developer Collaboration ..................................................................... 14

3.3 Empirical Studies on Reliability and Security ......................................................... 19

4 Security Correlation Study ......................................................................................... 22

4.1 Developer Activity Metrics ...................................................................................... 23

4.1.1 Diversity in Perspectives .................................................................................. 24

4.1.2 Unfocused Contributions .................................................................................. 27
4.2 Case Studies ........................................................................................................................................... 29

4.2.1 RHEL4 Linux Kernel ......................................................................................................................... 31

4.2.2 PHP Programming Language ......................................................................................................... 32

4.2.3 Wireshark Network Protocol Analyzer ............................................................................................. 32

4.3 Empirical Analysis ............................................................................................................................... 33

4.3.1 Association: Are The Metrics Correlated With Vulnerable Files? .............................................. 33

4.3.2 Discriminative Power: Are Some Metric Values Better Than Others? ................................. 35

4.3.3 Predictability: How Many Vulnerable Files Can Be Predicted? .................................................. 38

4.3.4 Cross-Project Analysis: Can Predictive Models be Transferred? ................................................ 40

4.4 Limitations of Security Correlation Study ......................................................................................... 41

4.5 Contributions of Security Correlation Study .................................................................................... 42

5 Perception Corroboration Study ........................................................................................................... 44

5.1 Developer Network Used in Perception Corroboration Study ......................................................... 46

5.2 Research Methodology ....................................................................................................................... 47

5.2.1 Online Survey .................................................................................................................................. 47

5.2.2 Conducting the Study ...................................................................................................................... 48

5.3 Case Studies ......................................................................................................................................... 50

5.3.1 Linux Kernel .................................................................................................................................... 51
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3.2</td>
<td>PHP Programming language</td>
<td>53</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Wireshark network protocol analyzer</td>
<td>54</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Criterion for Developer Network Edges</td>
<td>55</td>
</tr>
<tr>
<td>5.4</td>
<td>Perception Corroboration Analysis</td>
<td>58</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Collaborators and Edges</td>
<td>58</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Distance between Two Developers</td>
<td>61</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Centrality and Reputed Expertise</td>
<td>66</td>
</tr>
<tr>
<td>5.5</td>
<td>Limitations of the Perception Corroboration Study</td>
<td>69</td>
</tr>
<tr>
<td>5.6</td>
<td>Contributions of the Perception Study</td>
<td>70</td>
</tr>
<tr>
<td>6</td>
<td>Synthesis Study</td>
<td>72</td>
</tr>
<tr>
<td>6.1</td>
<td>Methodology</td>
<td>73</td>
</tr>
<tr>
<td>6.2</td>
<td>Recent Socio-Technical Research</td>
<td>74</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Literature Summary</td>
<td>74</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Socio-Technical Factors</td>
<td>77</td>
</tr>
<tr>
<td>6.3</td>
<td>The Socio-Technical Software Developer Network</td>
<td>79</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Connecting Developers</td>
<td>79</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Incorporating Time in Developer Networks</td>
<td>82</td>
</tr>
<tr>
<td>6.3.3</td>
<td>An Example Developer Network</td>
<td>83</td>
</tr>
</tbody>
</table>
6.3.4 Variations on the Developer Network .......................................................... 86

6.4 Trends and Conjectures ....................................................................................... 87

6.4.1 Number of Developers Changing a File ......................................................... 87

6.4.2 The Importance of Central Developers ......................................................... 89

6.4.3 Network Robustness and Developer Turnover .............................................. 91

6.4.4 Open Source Development and Proprietary Development ....................... 92

6.5 Summary of the Synthesis Study ....................................................................... 94

7 Contributions .......................................................................................................... 96

8 Future Work ............................................................................................................. 98

REFERENCES ............................................................................................................ 99

APPENDICES ............................................................................................................ 110

8.1 APPENDIX A. Developer Perception Survey ..................................................... 111
# LIST OF TABLES

Table 1. Developer activity metrics ............................................................................................................. 24
Table 2. Summary of Case Study Projects ........................................................................................................... 29
Table 3. Mann-Whitney-Wilcoxon test for association .................................................................................. 34
Table 4. Discriminative Power results with example critical values ............................................................... 37
Table 5. Discriminative Power results for critical values based on a fixed IR= 10% .......................... 38
Table 6. Bayesian Network prediction, 10x10 cross-validation ........................................................................ 39
Table 7. Cross-project Bayesian network prediction ......................................................................................... 41
Table 8. Case study metadata .......................................................................................................................... 51
Table 9. Unweighted distances between respondents and their named collaborators .................... 60
Table 10. Mean of squared Spearman rank correlation coefficients between unweighted and perceived distances ........................................................................................................................................... 64
Table 11. Mean of squared Spearman rank correlation coefficients between weighted and perceived distances ........................................................................................................................................... 65
Table 12. Summary of Votes for Experts ............................................................................................................... 68
Table 13. Association between centrality and reputed expertise ................................................................. 69
Table 14. Recent literature using a socio-technical perspective on development teams .................. 76
Table 15. Example log representing version control logs (VCLog) ............................................................. 84
LIST OF FIGURES

Figure 1. Example contribution network. ................................................................. 28
Figure 2. Screenshot of the respondent verifying his or her identity in the survey. .......... 48
Figure 3. Visualization of the Linux kernel developer network .................................. 52
Figure 4. Visualization of the PHP developer network. .............................................. 54
Figure 5. Visualization of the Wireshark developer network ...................................... 55
Figure 6. Width of the edge window and number of edges ........................................ 57
Figure 7. Question regarding collaborators and edges................................................ 59
Figure 8. Screen capture of the question regarding collaborators and edges................ 59
Figure 9. Question socio-technical distance between two developers.......................... 62
Figure 10. Example screen capture from custom survey system.................................... 63
Figure 11. Question regarding expertise...................................................................... 67
Figure 10. Screen capture of the question regarding project expertise.......................... 67
Figure 13. Resulting developer network from example................................................ 85
Figure 14. Pseudo-SQL code for computing edges in the example developer network ....... 85
1 Introduction

With each new developer to a software development team comes a greater challenge to manage the communication, coordination, and knowledge transfer amongst teammates. In large software development projects, no single person can possibly know every aspect of the system, so the team members must be organized into various structures of communication and coordination. Lack of team cohesion, miscommunications, and misguided effort can lead to all kinds of problems in the software product. An understanding of developer collaboration from the perspective of the entire team could help structure development efforts.

With respect to structuring development teams, conventional wisdom leads us in many different directions. One perspective originates from Eric Raymond in *The Cathedral and the Bazaar* [49], in the form of what he calls Linus’ Law:

> Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone. Or, less formally, given enough eyeballs, all bugs are shallow.

According to Raymond, collaboration among many developers is a valuable practice that ought to be intentionally pursued. With more people comes a more diverse set of perspectives that ultimately serve to benefit the product. As a result, a culture of software development that encourages collaboration is one that more closely represents the organic expansion of a bazaar than the formally organized building of a cathedral. To Raymond, the
ideal development team ought to include not only many developers, but a large amount of collaboration among those developers.

Fred Brooks presents a competing perspective in his widely-known book, The Mythical Man-Month [10]:

...training cannot be partitioned, so this part of the added effort varies linearly with the number of workers. [...] The added effort of communicating may fully counteract the division of the original [development] task.

 [...] Adding manpower to a late project makes it later.

The latter sentence is known as “Brooks’ Law.” While Brooks was specifically discussing effort estimation and productivity, one could apply a similar argument to many negative outcomes. Too many new developers in a short period of time can potentially lead to inconsistent implementation, poor system integration, or even security vulnerabilities being injected into the system. So, according to Brooks’ reasoning, having too many developers working together brings about a “too many cooks in the kitchen” effect that could negatively affect the product.

Furthermore, how the development team is structured is a concern that both Brooks and Raymond share. Brooks describes software development as “an exercise in complex interrelationships” [10], and describes the development team as a network of communication and coordination (a structure since more formally defined as a developer network [6], [32], [35], [36], [53], [70]). To Raymond, code should never be developed in relative isolation, rather the product ought to be stabilized as often as possible so that developers can review
and test one another’s work. When the team organization becomes too complex, collaboration suffers, and so could the product itself.

When team members are not properly coordinating effort, the product’s security is of particular concern. Vulnerabilities are rare, yet expensive. Our own case studies have shown that security vulnerabilities are rare occurrences: between 3% and 6% of source code files have a post-release vulnerability in the three case studies we examined [32], [33], [56]. As for the financial risk, vendors can lose 0.6% of their stock market value following the announcement of a security vulnerability in their software [62]. Furthermore, finding security vulnerabilities in software requires the developer to “think like an attacker”, which is a skill that requires both expertise and objectivity [28], [71]. With these expensive, hard-to-find security vulnerabilities lurking in huge code bases that require objective experts to find and fix, teams need to organize their development for better fortification.

While Raymond’s and Brooks’ views are well-reasoned, software development teams need empirical analysis to determine the underlying grains of truth of Brooks’ Law and Linus’ Law. In this dissertation, we examine developer activity metrics as a suite of metrics that are designed to quantify various properties of software development teams. The goal of this research is to provide actionable insight into structural nature of developer collaboration and software security by empirically analyzing developer activity metrics. The way developers work as a team can be influenced by all kinds of factors, both of the social nature and the technical nature. Sociologists refer to the way that laborers self-organize in a team as “socio-technical” [64]. Socio-technical factors can include shared knowledge,
communication, and coordination. In software development, for example, two developers working on the same code represents a potential socio-technical connection (e.g. shared knowledge) between the two developers. In this research, we derive our developer activity metrics from a combination of analyzing network structures (i.e. graphs) where connections are defined by socio-technical factors of how software developers work together.

We approach our study of developer activity metrics from two directions: whether or not developer activity metrics are related to security, and whether or not developer activity metrics actually represent the structure of the development team. The latter is a form of internal validity, and the former is external validity of software metrics [15], [16], [27]. Additionally, we synthesize our results with the results of other case studies to provide conjectures about developer activity metrics and socio-technical research in general. Specifically, the research questions we examine in this dissertation are:

- **Security Correlation Study** (Chapter 4). Are developer activity metrics related to security vulnerabilities?
  - Are developer activity metrics statistically associated with vulnerable files?
  - Can a “critical point” be found in each metric’s range that is linked to an increase in the likelihood of having a vulnerable file?
  - How many of the vulnerable files can be predicted by developer activity metrics?
  - Can security data from one project be used to predict vulnerabilities on another project?
• **Perception Corroboration Study (Chapter 5).** Does the developer network represent the socio-technical structure of the development team?
  - If two developers work on the same file in the same month, then do they perceive they are collaborating?
  - Does the distance between two developers in the developer network represent the perceived degree of separation between those two developers?
  - Does having a high developer centrality in the developer network indicate being a project expert?

• **Synthesis Study (Chapter 6).** What is the status of socio-technical research of software development teams?
  - How does the number of developers working on source code affect its quality?
  - How does developer network centrality relate to roles and responsibilities of individual developers in the development team?
  - How does developer network robustness relate to risk in software development?
  - How does the developer network differ between open source and closed source case studies?

The rest of this dissertation is organized as follows. Chapter 2 provides background with respect to network analysis, software development terminology, and statistical techniques. Chapter 3 provides related work. Chapter 4 presents our security correlation study between developer activity metrics and security vulnerabilities. Chapter 5 presents our study on corroborating the meaning of developer activity metrics with developer perceptions. Chapter
6 covers a synthesis study of the current academic work in developer activity metrics. Chapters 7 and 8 summarize our contributions and future work.
2 Background

In this section, we provide definitions for terms used and techniques used throughout this document.

2.1 Network Analysis

Network analysis is the study of characterizing and quantifying network structures, represented by graphs [8]. In network analysis, vertices of a graph are called nodes, and are connected via edges. A sequence of non-repeating, adjacent nodes is a path, and a shortest path between two nodes is called a geodesic path. Note that a geodesic path is not necessarily unique, that is, two or more paths may be tied for the shortest path between two nodes. In the case of weighted edges, the geodesic path is the path of minimum weight. The diameter of a network is equal to the length of the longest unweighted geodesic path in the network. Informally, the unweighted geodesic path represents the “degree of separation” between two nodes.

In network analysis, centrality metrics are used to quantify the location of a node or edge relative to the rest of the network. In this document, we use two centrality metrics: degree and betweenness. The degree of a node is equal to the number of neighbors a node has (not to be confused with “degree of separation”). The betweenness [8] of node \( n \) is defined as the number of geodesic paths that include \( n \). Similarly, the edge betweenness of edge \( e \) is defined as the number of geodesic paths which pass through \( e \). A high betweenness means a high centrality.
Additionally, we analyze groups of nodes with clusters. A cluster of nodes is a set of nodes such that the number of intra-set connections greatly outnumbers the number of inter-set connections [8]. For analyzing clusters in a graph, we used the edge betweenness clustering technique [8]. The edge betweenness clustering technique is a standard network analysis algorithm based on the principle that edges of highest betweenness are also edges that connect the largest clusters in the network. In our studies, we applied this principle to identify edges that lie between multiple clusters of nodes.

2.2 Software Development Terminology

When we refer to a developer, we are referring to any person who has been making changes to the source code of a given project. By extension, we use the term development team to refer to the entire group of developers who are changing code on a given software project. Depending on the team culture, testers and managers may or may not be changing the source code during the development process.

2.3 Security Terminology

A vulnerability is “a flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's [implicit or explicit] security policy” [51]. When we refer to a source code file as being vulnerable, we are implying that the source code file was found to have a vulnerability at the time of the software’s release. We do not count vulnerabilities from regressions. That is, if a vulnerability was introduced by a post-release fix and not by development, it would not have been present at the time of
release. If a vulnerability has never been found for a given source code file, then we label that file as being “neutral”.

2.4 Statistical Techniques

Throughout this research, we use a variety of statistical techniques to analyze our observational data. In many situations, we use binary classification, where each of our observations is a source code file that is classified as either vulnerable or neutral. To test if a numerical metric is associated with being vulnerable at a statistically significant (p<0.05) level, we use the Mann-Whitney-Wilcoxon (MWW) rank sum test. We use the MWW test because it does not assume an underlying normal distribution for our metrics [48]. The MWW test computes a p-value, which we compare to 0.05 to reach our conclusions.

Beyond association, we also form models to predict whether a file will be vulnerable or neutral. The primary type of model we used for prediction was Bayesian networks. Bayesian networks rely on Bayesian inference on a network of metrics, taking into account conditional dependencies between the metrics [69]. The output of a Bayesian network is a prediction of whether a file is neutral or vulnerable, which we can compare to the actual values. A binary classifier can make two possible types of errors: false positives (FP) and false negatives (FN). In these studies, a FP is the classification of a neutral file as vulnerable, and a FN is the classification of a vulnerable file as neutral. Likewise, a correctly classified vulnerable file is a true positive (TP), and a correctly classified neutral file is a true negative (TN). We evaluate predictive performance using precision, recall, inspection rate, and F-score. Precision (P) is defined as the proportion of correctly predicted vulnerable files:
\[ P = \frac{TP}{TP+FP} \]  \hspace{1cm} (1)

**Recall (R)** is defined as the proportion of total vulnerabilities found:

\[ R = \frac{TP}{TP+FN} \]  \hspace{1cm} (2)

**Inspection Rate (IR)** is the proportion of total files that were classified as vulnerable. We introduced the term in our own publications as a way to estimate the amount of effort that a software development team would need to plan for if they used our model [56].

\[ IR = \frac{(TP+FP)}{(TP+TN+FP+FN)} \]  \hspace{1cm} (3)

Optimally, inspection rate is minimized while precision and recall are maximized. For example, an IR=10% and R=50% means that the classifier found 50% of the known vulnerabilities in just 10% of the files. A classifier with P=25% means that, of the files classified as vulnerable, 25% were actually vulnerable.

Additionally, we use the F-measure as a combination of precision and recall. We use the F-measure primarily to compare the overall predictive performance of two models. The F-measure contains a parameter \( \beta \) that allows one to assign a higher weight to either precision or recall. The F-measure is defined in Equation 1 as the harmonic mean of precision and recall, where the parameter \( \beta \) represents the weight to which one assigns recall over precision [23].

\[ F_\beta = \frac{1 + \beta^2 \cdot \text{precision} \cdot \text{recall}}{\beta^2 \cdot \text{precision} + \text{recall}} \]  \hspace{1cm} (4)

For situations where we needed to examine the amount of agreement between the ranks of two ordinal variables, we used the **Spearman rank correlation coefficient**. The Spearman
rank correlation coefficient is a non-parametric measure of the linear relationship between two random variables. In some situations, we aggregated many Spearman coefficients by computing their mean. We test the significance of this mean by computing a 95% confidence interval for the following transformation of Spearman’s $r$:

$$\frac{\sqrt{n-3}}{2} \log\left(\frac{1 + r}{1 - r}\right)$$

We used the Student’s $t$-test confidence interval for the mean of Spearman coefficients [26], [46], which assumes that the Spearman coefficient itself is normally distributed.
3 Related Work

Our own prior work is covered in Section 3.1. In Sections 3.2 and 3.3, we cover recent related work in the area of measuring developer collaboration, metrics vulnerability prediction, and software metrics studies, respectively.

3.1 Prior Work

In a study at Nortel [37], we examined the relationship between developer activity metrics and reliability. The empirical case study examined three releases of a large, proprietary networking product. The authors used developer centrality metrics from the developer network to examine whether files are more likely to have failures if they were changed by developers who are peripheral to the network. The authors formed a model that included metrics of developer centrality, code churn (the degree to which a file was changed recently), and lines of code to predict failures from one release to the next. Their model’s prioritization found 58% of the system’s failures in 20% of the files, where a perfect prioritization would have found 61%. The study did not include work on developer clusters, unfocused contributions, or security.

We also performed an empirical case study [29] [34] of the OpenMRS\(^1\) healthcare system to examine how developers collaborate on issue tickets. We proposed two annotations to be used in issue tracking systems: solution originator and solution approver. We annotated which developers were originators or approvers of the solution to 602 issues from the OpenMRS healthcare system. We used these annotations to augment the version control logs

\(^1\) http://www.openmrs.org
and found 47 more contributors to the OpenMRS project than the original 40 found in the version control logs. Using social network analysis, we found that approvers are likely to score high in centrality and hierarchical clustering. Our results indicate that issue tracking annotations are an improvement to developer activity metrics that strengthen the connection between what we can measure in the project development artifacts and the team's collaborative problem-solving process.

In collaboration with Cisco [30], we performed a longitudinal case study of a development project examining team expansion. Using team-level metrics, we quantified characteristics of team expansion, including team size, expansion rate, expansion acceleration, and modularity with respect to department designations. We examined statistical correlations between our monthly team-level metrics and monthly product-level metrics. Our results indicate that increased team size and linear growth are correlated with later periods of better product quality. However, periods of accelerated team expansion are correlated with later periods of reduced software quality. Furthermore, our linear regression prediction model based on team metrics was able to predict the product’s post-release failure rate within a 95% prediction interval for 38 out of 40 months. Our analysis provided insight for project managers into how the expansion of development teams can impact product quality.

Beyond performing empirical case studies [31], we performed a systematic literature review of software metric validation criteria. We conducted a systematic literature review that began with 2,288 papers and ultimately focused on 20 papers. After extracting 47 unique validation criteria from these 20 papers, we performed a comparative analysis to explore the
relationships amongst the criteria. We present an analysis of the criteria’s categorization, conflicts, common themes, specific benefits, and philosophical motivations behind the validation criteria. Researchers proposing new metrics can consider applying these validation criteria using our analysis.

### 3.2 Quantifying Developer Collaboration

The topics of developers and collaboration have been examined in several recent empirical studies. All of the studies, however, either examine the meaning of developer activity metrics or relate them to reliability. None of the studies relate developer activity metrics to security.

Wolf, et al. [70] applied social network analysis techniques to do an historical case study of the IBM Jazz development platform. Their socio-technical network was designed to examine the “relationship between successful coordination outcome and communication structures” in software teams. That is, the researchers were examining if the outcome of coordinating with each other resulted in a failed build. Forming a developer network based on recorded communication in the team, the researchers applied social network analysis measures to the network. No individual measure was able to successfully predict the outcome of the build, but they found that a combination of their network analysis measure was able to predict successful build outcomes with a 55% to 75% recall and 50% to 76% precision.

Bird et al. [1] examined social structures in open source projects using social network analysis. Also discussing connections and contradictions between some of Brooks’s ideas [4] and the bazaar-like development of open source projects, the authors empirically examine
how open source developers self-organize. The authors use similar network structures as our
developer network to find the presence of sub-communities within open source projects. In
addition to examining version control change logs, the authors mined email logs and other
artifacts of several open source projects to find a community structure. The authors conclude
that sub-communities do exist in open source projects, as evidenced by the project artifacts
exhibiting a social network structure that resembles collaboration networks in other
disciplines. In our study, we leverage network analysis metrics as an estimation of
collaboration and examine their relationship to vulnerabilities in the project.

Pinzger et al. [17] were the first to propose the contribution network. The contribution
network is designed to use version control data to quantify the direct and indirect
contribution of developers on specific resources of the project. The researchers used metrics
of centrality in their study of Microsoft Windows Vista and found that closeness was the
most significant metric for predicting reliability failures. Files that were contributed to by
many developers, especially by developers who were making many different contributions
themselves, were found to be more failure-prone than files developed in relative isolation.
The finding is that files which are being focused on by a few developers are less problematic
than files developed by many developers. In our study, we use centrality metrics on
contribution networks to predict vulnerabilities in files.

Nagappan et al. [14] created a logistic regression model for failures in the Windows Vista
operating system. The model was based on what they called “Overall Organizational
Ownership” (OOW). The metrics for OOW included concepts like organizational
cohesiveness and diverse contributions. Among the findings is that more edits made by many, non-cohesive developers leads to more problems post-release. The OOW model was able to predict with 87% average precision and 84% average recall. The OOW model bears a resemblance to the contribution network in that both models attempt to differentiate healthy changes in software from the problematic changes.

Weyuker et al. [25] examined various releases of a large industrial software system to predict which files are most likely to contain the largest number of faults. Inspection guidance and automated testing efforts are among the applications intended for their fault prediction model. Their model is based on the negative binomial distribution and their model’s variables, based on developer information, attempt to capture information about the amount and the type of developers who have worked on any given file. Validation for their model included a comparison with a working model based on static code metrics and churn information. Weyuker et al. reported finding 84.9% of the faults in 20% of the files with the developer information, where without the developer information, 83.9% of the faults were found. The amount of failures found using the optimal prioritization was not mentioned. Our models use some similar developer counts in combination with network metrics to predict failures.

Zimmerman and Nagappan [27] applied network analysis to dependency graphs for predicting failures in files. By applying metrics of centrality and network motifs to the directed dependency graphs of source code, the researchers found that central components were more failure-prone. Furthermore, network metrics proved to identify 60% of the
critical, failure-prone binaries, which was better than object-oriented complexity metrics that only identified 30%.

Mockus and Weiss [38] used metrics based on developer information for failure prediction to assess risk in a large industrial software system. Developer metrics included counts of distinct developers and a quantitative measurement of developer experience in terms of recent changes of the current project, experience in the subsystem, and in the product overall. They used step-wise variable selection to construct a logistic regression model for estimating post-release failures.

Hudepohl et al. [16] used developer information in combination with various other metrics to create a risk assessment tool at Nortel called EMERALD. The developer information was a measurement of experience similar to the variables used by Mockus and Weiss. EMERALD’s developer variables, however, incorporated developer experience in terms of Nortel career, as opposed to specific projects. For example, one of the experience measurements was the count of the number of developers who were within their first ten code updates while working at Nortel as a way to identify inexperienced developers. EMERALD’s other variables included complexity metrics, customer usage metrics, churn information, and past failure counts from both testing and post-release phases. Hudepohl et al. reported that over half of the field failure patches were correctly identified as “red” (highest risk) in 20% of the files.

Arisholm and Briand [3] identified developer experience and skill level as fundamental factors affecting fault-proneness in an object-oriented system. Since they had no data on
skills and experience of developers, they did not consider developer information in their model. Nonetheless, they used a stepwise logistic regression model and a cross-validation classification analysis to validate their results. Most of the variables in their model could be classified in the categories of object-oriented metrics and code churn information. Their results from cross-validation analysis showed less than 20% false positives and false negatives, with an estimated verification effort savings of 29%. We used developer information in our model, however, not based on skill but on the structure of developer connections within the developer network.

Bird et al. [5] performed an empirical case study of the Microsoft Windows Vista operating system to examine if software quality suffers under distributed development. They compared the number of post-release failures for binaries that were developed by collocated teams. They examined several levels of collocation, including sharing a building, cafeteria, campus, locality (i.e. region), and continent. In general, the binaries showed no statistically significant difference in number of post-release failures between the distributed and collocated teams.

Also, Sarma et al. [15] and Begel et al. [3] have developed tools that visualize and utilize many different aspects of development artifacts, including the developer network. Nia, et al. [12] discuss the validity of information flow in situations where not every edge of the developer network may be known. Their findings indicate that some centrality measures are sensitive to such a missing edge, and provide guidance on which measures ought to be used with developer networks. We do not apply the measures that Nia, et al. consider to be invalid.
3.3 Empirical Studies on Reliability and Security

Nagappan and Ball [21] used metrics based on code churn data to predict defect density in Windows Server 2003. Their hypothesis was on comparing the predictive power of relative code churn metrics to absolute code churn metrics. A relative code churn metric, as defined by Nagappan and Ball, is one that is normalized by parameters such as lines of code, files counts, etc. Multiple linear regression, Principle Component Analysis, and step-wise variable selection were all used to make predictions about defect density. Data splitting was used to validate the predictive power of the chosen model and to show that relative code churn metrics are more powerful than absolute code churn metrics. Along with the use of code churn metrics, similar statistical techniques to ours were used, such as multiple linear regression, logistic regression, and step-wise variable selection.

Neuhaus et al.[15] examined patterns of vulnerable software components in the history of Mozilla Firefox. The authors present a tool that could be used to automatically mine the locations of known security vulnerabilities. Known security vulnerabilities can be traced to security issues tracked by Bugzilla, which are then linked to the version control commits via commit messages in the version control system. The authors also examined components in terms of combining a C/C++ header file (.h) with its source file (.c or .cpp). The authors found that components likely to have a single vulnerability were not likely to have another vulnerability. Additionally, the authors discovered components with multiple vulnerabilities had some commonalities specific to the Mozilla project, such as importing specific libraries or calling specific functions.
Shin and Williams [57], [58] investigated whether the code level complexity metrics such as cyclomatic complexity can be used as indicators of vulnerabilities at the function level. The authors performed a case study on the Mozilla JavaScript Engine written in C/C++. Their results included statistically significant correlations (Spearman r=0.30 at best) between complexity metrics and vulnerabilities. Interestingly, the complexity measures for vulnerable functions were higher than the ones for faulty functions. Results of the vulnerability prediction using logistic regression showed very high accuracy (over 90%) and low false positive rates (less than 2%), but the false negative rate was very high (over 79%). To extend the study, Shin et al. [20], evaluated the statistical connection between vulnerabilities and metrics of complexity, code churn, and developer activity. The study denotes two case studies of large, open source projects: multiple releases of Mozilla Firefox and the Red Hat Enterprise Linux 4 kernel. Among the findings include a statistically significant correlation between metrics of all three categories and security vulnerabilities. Also, in the Mozilla project, a model containing all three types of metrics was able to find 70.8% of the known vulnerabilities by selecting only 10.9% of the project’s files.

Gegick et al. [19] examined metrics from the reliability field as predictors of security vulnerabilities. They performed an empirical case study of a Cisco product, examining components of the system that had both security vulnerabilities and non-security faults. They used a classification and regression tree to predict vulnerable components of the system. They found that 57% of the vulnerable components were in the top nine percent of the components, but with a 48% false positive rate.
Walden et al. [65] analyzed the association between the security resource indicator (SRI) and vulnerabilities on fourteen open source PHP web applications. The SRI is measured as a sum of binary values depending on the existence of the four resources in development organizations: a security URL, a security email address, a vulnerability list for their products, and secure development guidelines. The researchers found that SRI is useful to compare security levels between organizations, but does not indicate vulnerable code locations. Additionally they measured the correlation between three complexity metrics and vulnerabilities. The correlations varied, with complexity metrics performing similarly as code size and code churn metrics (Spearman r=0.31 at best). The study measured vulnerability density using a static analyzer, not with reported vulnerabilities.

Nguyen and Tran [41] did a case study of the Mozilla Firefox JavaScript engine examining correlations between various metrics security vulnerabilities. The goal of the study was to produce the strongest prediction model possible using public security vulnerability data. The researchers examined 22 attributes, including cyclomatic complexity, code size, and data flow metrics. On average, the prediction model was able to predict vulnerabilities with 60% precision and 60% recall.
4 Security Correlation Study

Open source software is often considered to be secure [23], [68]. One factor in this confidence in the security of open source software lies in leveraging large developer communities to find vulnerabilities in the code [23], [68]. Eric Raymond states Linus’ Law as “Given enough eyeballs, all bugs are shallow”. According to Raymond’s reasoning, diversity of developer perspectives ought to be embraced, not avoided. Therefore, more developers mean more vulnerabilities found and fixed, or even prevented.

But does Linus’ Law hold up ad infinitum? Can a project have too many developers, resulting in insecure software?

One opposing force to Linus’ Law might be the notion of “too many cooks in the kitchen”, or what has been called an unfocused contribution [47]. Consider having many people make a meal: without enough coordination and communication, ingredients can get skipped, added twice, or significant steps of the recipe may be left out. The meal can suffer as a result of too many people. Likewise, if source code does not receive the focus it needs because of too many people, perhaps the security of a software project can suffer as a result.

An analysis of the structure of open source developer collaboration can help the community understand how this structure impacts the prevention or the injection of security vulnerabilities. We performed an empirical analysis of Linus’ Law and unfocused contributions in the open source Red Hat Enterprise Linux 4 (RHEL4) kernel, the PHP

---

2 http://www.redhat.com/rhel
programming language\(^3\) and the Wireshark\(^4\) network protocol analyzer. We performed an empirical analysis by quantifying developer collaboration and unfocused contributions into *developer activity metrics*. We used version control change logs to calculate four developer activity metrics that quantify Linus’ Law and unfocused contributions. We found a statistical association between all four developer activity metrics and vulnerabilities. Lastly, we perform a cross-project analysis to examine if predictive models can be trained on multiple projects and applied to another project.

### 4.1 Developer Activity Metrics

In our case studies, we used the version control logs to analyze development activity. As a project progresses, developers make changes to various parts of the system. With many changes and many developers, changes to files tend to overlap: multiple developers may end up working on the same files around the same time, indicating that they share a common contribution, or a *connection*, with another developer. As a result of which files they contribute to, some developers end up connected to many other highly connected developers, some end up in groups (“clusters”) of developers, and some tend to stay peripheral to the entire network.

Both developers and files become organized into a network structure with some developers/files being the middle of the network, in a cluster, or on the outside. In this section, we quantify the structure of changes in the system using network analysis to create four developer activity metrics. We define our suite of developer activity metrics based on

\(^3\) [http://php.net](http://php.net)

\(^4\) [http://wireshark.org](http://wireshark.org)
two networks: developer networks and contribution networks, as will be discussed in Sections 4.2.1 and 4.2.2, respectively.

A summary of the interpretation for each of the four metrics can be found in Table 1. We empirically evaluate these metrics as indicators of vulnerable files in Section 4.3.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Definition for a file</th>
<th>High values are symptomatic of …</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumDevs</td>
<td>The number of distinct developers who changed the file</td>
<td>Many developers worked on the file</td>
</tr>
<tr>
<td>DNMaxEdgeBetweenness</td>
<td>The maximum of the number of geodesic paths in a developer network which include an edge that the file was on</td>
<td>A file being changed by multiple, otherwise separate developer groups</td>
</tr>
<tr>
<td>NumCommits</td>
<td>The number of commits made to a file</td>
<td>Developers made many changes to the file</td>
</tr>
<tr>
<td>CNBetweenness</td>
<td>The number of geodesic paths containing the file in the contribution network</td>
<td>File was changed by many developers who made many changes to many other files</td>
</tr>
</tbody>
</table>

### 4.1.1 Diversity in Perspectives

In his essay on open source development [49], Eric Raymond states one of the laws colloquially as Linus’ Law: “Given enough eyeballs, all bugs are shallow” with the reasoning that, in a bazaar-like style of software development, having more people work on the project yields a greater diversity in understanding, leading to better improvements. While Linus’ Law includes a broad scope of users, testers, and developers, we focus our study on developer groups as one aspect of Linus’ Law. We use two metrics to quantify the group aspect of Linus’s Law: NumDevs and DNMaxEdgeBetweenness.
The NumDevs metric is the number of distinct developers who made a commit to the file. According to the reasoning behind Linus’ Law, NumDevs should have a positive impact on the security of a file, leading to a hypothesis that neutral files would have contributions by more developers than vulnerable files.

The number of developers contributing to one file, however, is not the only aspect of Linus’ Law we wish to quantify. We can also look at how developer groups (or clusters) form and how strongly connected these clusters are. Two separate groups may be working on similar areas without working together. According to Raymond’s reasoning about diverse perspectives resulting in better quality, files worked on by otherwise-separated developer groups ought to be more likely to be vulnerable because the groups are not fully working with each other.

To empirically analyze developer groups, we need to first measure developer collaboration. The first step we take to formally estimate developer collaboration is to use a developer network [29], [32], [33], [35], [37], [56]. In our developer network, two developers are connected if they have both made a change to at least one file in common during a specified period of time (one month in our studies). The result is an undirected, unweighted, and simple graph where each node represents a developer and edges are based on whether or not they have worked on the same file within a specified period of time.

Next, we examine files between developer groups. In network analysis, the notion of groups is formalized by the term cluster (defined in Section 2.1). A cluster of developers has more connections within the cluster than to developers outside the cluster. The files that are
worked on by otherwise-separated clusters, therefore, may be more likely to have a vulnerability because the two clusters are not working together and embracing diversity in perspectives.

In this study, we are using a developer network cluster metric to identify files that have been worked on by otherwise-separated clusters of developers. To this end, we use the edge betweenness clustering technique (described in Section 2.1) [50], [66] for discovering developer clusters. The motivation for using edge betweenness is that the betweenness of edges within a cluster will be very low since the geodesic paths will be evenly distributed (in most cases, developers are directly connected to each other within clusters). Since files have a many-to-many relationship to edges, we use the maximum of edge betweenness of the files in the developer network, hence $DNMaxEdgeBetweenness$.

Note that improving upon the $DNMaxEdgeBetweenness$ of a file does not require a change in the file itself, but on creating more connections between the two groups. One could create more connections by finding other files that require improvement by both groups. Once more connections are established, the number of geodesic paths from one cluster of developers to the other will be spread out over the new connections, lowering the edge betweenness and, by definition, forming a single cluster. While the optimal developer network need not be a single cluster, one could use the $DNMaxEdgeBetweenness$ metric to identify two clusters of developers who would benefit from working together.
4.1.2 Unfocused Contributions

In the open source community, some developers may choose to make changes to many different parts of the system without collaborating with other developers who could share knowledge about the system and provide feedback on the suggested change. This effect has been referred to as an unfocused contribution [32], [33], [47] and could be a source of security problems.

To empirically analyze unfocused contributions, we use two metrics: NumCommits and CNBetweenness. The NumCommits metric is calculated similarly to NumDevs: taken directly from the version control logs. NumCommits is the number of commits made to the file during the time period under study. Note that NumCommits and NumDevs can vary independently: a file can have many commits and few developers. Also, NumDevs could also be classified as an unfocused contribution metric. If “too many developers” working on a file result in the file being more vulnerable, then the meaning behind the association would support the “too many cooks in the kitchen” notion.

However, NumCommits and NumDevs only represent the number of people and changes, not what else those developers were also working on at the time. Thus, we add a third, more specific metric to our study: CNBetweenness.

The CNBetweenness metric is calculated from a contribution network as defined by Pinzger, et al[47]. Informally, the network represents who contributed changes to which file. Formally, the contribution network employs an undirected, weighted, and bipartite graph with two types of nodes: developers and files. An edge exists where a developer made changes to a file. Edges exist only between developers and files (not from developers to
developers or files to files). The weight of an edge is the number of version control commits a developer made to the file.

We use the betweenness centrality measurement to quantify the focus made on a given file. If a file has a high betweenness, then it was changed by many developers who made changes to many other files. If a file had a low betweenness, then the file was worked on by fewer developers who made fewer changes to other files.

Consider the difference in contributions in Figure 1. For the file `quota.c`, changes were made by developers who worked on only a few other files, some of which were in common with each other. By focusing on a smaller number of files, and (by extension coordinating with fewer developers), the developers of `quota.c` are more focused on `quota.c`, and may be more likely to catch security vulnerabilities. The developers of `eventpoll.c`, however, are also working on many other files themselves, and may not catch security problems in `eventpoll.c`. As a result, `quota.c` had a more focused contribution, and perhaps a lower likelihood of a vulnerability, than `eventpoll.c`.

![Example contribution network.](image)

Figure 1. Example contribution network.
The CNBetweenness of a file is increased by (a) having many developers work on a file, and (b) having developers work on many different files. However, one can also improve (i.e. decrease) a file’s CNBetweenness by changing which developers work on which files rather than just reducing the amount of work for developers. As a result, CNBetweenness can be useful for assigning tasks to developers without adjusting the level of change in a file.

4.2 Case Studies

In the following sections, we describe the three open source projects we performed our case studies on. Note that the version control histories of all three projects contain records of “committers”, which in this study we refer to as “developers”. All three projects use the C programming language and some assembly, so we only included files with the file name extensions: .c, .S, and .h. Table 2 contains metadata on all three projects.

<table>
<thead>
<tr>
<th>Table 2. Summary of Case Study Projects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Total number of files (.c, .S, .h)</td>
</tr>
<tr>
<td>Number of files changed (total studied)</td>
</tr>
<tr>
<td>Number of developers</td>
</tr>
<tr>
<td>Development time</td>
</tr>
<tr>
<td>Percentage of files changed in</td>
</tr>
<tr>
<td>development time frame studied</td>
</tr>
<tr>
<td>Total number of vulnerable files</td>
</tr>
<tr>
<td>Percentage of files changed in</td>
</tr>
<tr>
<td>development time fixed for a</td>
</tr>
<tr>
<td>vulnerability</td>
</tr>
<tr>
<td>Total number of commits</td>
</tr>
</tbody>
</table>
We performed our analysis on three software projects: the Linux kernel, the PHP programming language, and the Wireshark network protocol analyzer [32], [33]. We chose these three projects as representatives of a widely-used open source projects with varying domains and varying developer community sizes. A summary of all three projects can be found in Table 2.

Our data set in each study included a labeling of whether or not a source code file was patched with a post-release vulnerability (“vulnerable” or “neutral”). We chose the label “neutral” for files with no known vulnerabilities because we do not know that the file truly has zero vulnerabilities, just no known vulnerabilities. Gathering the security data involved tracing through the development artifacts related to each vulnerability reported via public issue tracking databases (as specified in Sections 4.2.1 through 4.2.3). Since reported vulnerabilities were all handled slightly differently by each community, we investigated each defect report manually to ensure that the correct post-release patch was made public. For the rare cases in which no connection could be made, we contacted experts of the team to correct the historical records. In all three case studies, we examined releases that were at least three years old to allow time for vulnerabilities to be found, fixed, and documented.

Each of the three projects had various types of security vulnerabilities. Most of the vulnerabilities were small, code-level mistakes in which the fix involved no more than a few lines of code changed. The three most common types of vulnerabilities were buffer
overflows, denial of service (via a variety of means), and information disclosure vulnerabilities.

We gathered the developer activity metrics from version control change logs. We chose the length of our development history based on major changes in the project (e.g. the development since the last major project vision change). Our developer activity metrics applied only to files with development history during the period we studied.

4.2.1 RHEL4 Linux Kernel

We performed a case study on the Linux kernel as it was distributed in the RHEL4 operating system. We collected our security data from the Red Hat Bugzilla database\(^5\), the National Vulnerability Database (NVD)\(^6\), and the RHSR security metrics database\(^7\). Our data set includes 284 vulnerable files traced from vulnerabilities reported between February 2005 and February 2010.

For the version control data from which developer activity metrics were computed, we used the Linux kernel source repository\(^8\). The RHEL4 operating system is based on Linux kernel version 2.6.9, so we used all of the version control data from kernel version 2.6.0 to 2.6.9, which was approximately 16 months of development and maintenance. In the Linux kernel study, 21 vulnerable file were not changed in the 16 months prior to release, so developer activity metrics were not applied to those files and they were not included in this study.

\(^5\) http://bugzilla.redhat.com
\(^6\) http://nvd.nist.gov
\(^7\) http://redhat.com/security/metrics
\(^8\) http://www.kernel.org
4.2.2 PHP Programming Language

The PHP project is a programming language for web application development. Vulnerabilities in PHP typically entailed insecure built-in functions provided by the core language. Our study does not include vulnerabilities that could arise from writing vulnerable PHP code.

We collected our PHP security data from the NVD, the PHP Bugzilla database, and Bugzilla databases of vendors that support the PHP programming language (e.g. Red Hat Bugzilla). We studied version 4.3 of the PHP language released because it was a major release of the language. Our security data ranged from release in September 2004 to February 2010. We used the version control history from 24 months prior to release based on observing a two-year development lifecycle. In the PHP study, only one vulnerable file was not changed in the 24 months prior to release and was not included in this study.

4.2.3 Wireshark Network Protocol Analyzer

The Wireshark network protocol analyzer is a tool that can be used to aggregate and summarize data transported over a network. Formerly known as Ethereal, Wireshark can be deployed on a system to record network traffic, making the system susceptible to exploits if Wireshark contains any vulnerabilities.

We collected our Wireshark security data from the Wireshark security advisories. We studied version 0.99.4 of Wireshark released in September 2006, including all security vulnerabilities found from release to February 2010. We studied the version of Wireshark based on a major release that coincided with increased record-keeping practices in the

---

9 http://bugs.php.net

32
project. We used the version control history from 24 months prior to release based on several natural development lifecycles. In the Wireshark study, all of the vulnerable files were changed in the 24 months prior to release.

4.3 Empirical Analysis

Our empirical analysis is a statistical correlation study between developer activity metrics and security vulnerabilities. We focus our empirical analysis on four questions in the following four subsections:

- Section 4.3.1: Are developer activity metrics related to vulnerable files?
- Section 4.3.2: Can a “critical point” be found in each metric’s range that is linked to an increase the likelihood of having a vulnerable file?
- Section 4.3.3: How many of the vulnerable files can be explained by the metrics?
- Section 4.3.4: Can data from one project be used to predict on another project?

Statistically speaking, the first question is an association question, the second is a discriminative power question, and the third and fourth are predictive modeling questions as described by Schneidewind’s metric validation methodology [54]. We used SAS\textsuperscript{10} v9.1.3 for our statistical analysis and Weka v3.6.0 [69] for the Bayesian network prediction model.

4.3.1 Association: Are The Metrics Correlated With Vulnerable Files?

To examine how each of the four metrics summarized in Table 3 are related to security vulnerabilities, we examine the difference between the vulnerable files and the neutral files in terms of each metric. As suggested in metrics validation studies [54] for not having a

\textsuperscript{10} http://www.sas.com/
normality assumption, we use the non-parametric **Mann-Whitney-Wilcoxon** (MWW) test. Three outcomes are possible from this test:

- The metric mean is higher for vulnerable files than neutral files;
- The metric mean is lower for vulnerable files than for neutral files; or
- The metric means are not different between neutral and vulnerable files at a statistically significant level ($p<0.05$).

We present the results of our association analysis in Table 3. In all four cases on all three case studies, the metric was higher for vulnerable files than for neutral files at a statistically significant level, providing some mixed results regarding Linus’ Law and unfocused contributions.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Linux Kernel</th>
<th>PHP</th>
<th>Wireshark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Files (n)</td>
<td>10,454</td>
<td>776</td>
<td>2,541</td>
</tr>
<tr>
<td>Number of vuln. files</td>
<td>284</td>
<td>44</td>
<td>67</td>
</tr>
<tr>
<td>DNMaxEdgeBetweenness</td>
<td>18.4</td>
<td>124.2</td>
<td>p&lt;0.0001</td>
</tr>
<tr>
<td>NumDevs</td>
<td>22.2</td>
<td>5.0</td>
<td>p&lt;0.0001</td>
</tr>
<tr>
<td>NumCommits</td>
<td>4.0</td>
<td>14.2</td>
<td>p&lt;0.0001</td>
</tr>
<tr>
<td>CNBetweenness</td>
<td>3602.0</td>
<td>12,673.0</td>
<td>p&lt;0.0001</td>
</tr>
</tbody>
</table>

The DNMaxEdgeBetweenness was higher for vulnerable files, meaning that files developed by multiple, otherwise-separated clusters of developers were more likely to have a vulnerability. This result supports the notion that, when two otherwise-disparate groups of developers have a common interest, multiple connections between the groups ought to be made, which promotes diversity in perspectives.
However, the NumDevs metric was higher for vulnerable files, implying that too many developers changing a single file is associated with an increase in likelihood of a vulnerability. This result supports the unfocused contribution aspect of NumDevs rather than the diversity in perspectives. This result may be surprising as it goes against Linus’ Law, indicating that too many eyeballs may be detrimental to the security of the software.

NumCommits was higher for vulnerable files, meaning that vulnerable files were more likely to have undergone many changes. This result supports the “code churn” effect found in other studies [37], [39], [56] where code undergoing a lot of change tends to have more problems.

CNBetweenness was also higher for vulnerable files than for neutral files, meaning that vulnerable files were more likely to have been worked on by many developers who also worked on many other files. This result supports the unfocused contribution view.

4.3.2 Discriminative Power: Are Some Metric Values Better Than Others?

By evaluating the discriminative power of developer activity metrics, we are examining how well each metric can individually differentiate files as vulnerable or neutral. The primary purpose of discriminative power is to see where a metric is “too high” or “too low”. A secondary advantage of discriminative power is to provide a comparison between each metric. Difference in averages (i.e. association) does not show relative correlation strength from one metric to the next.\textsuperscript{11}

\textsuperscript{11} E.g. NumDevs has a much smaller range of values than CNBetweenness, so the size of the difference in averages should not be compared
We use the term **critical value** of a metric to indicate a specific point that can be used to classify files as either vulnerable or neutral. For example, finding the critical value of NumDevs would answer the question: how many developers is “too many”? The exact critical value of a metric may vary depend on one’s desired precision and recall. As an example of using critical values, consider gathering all files in the Linux kernel by changed nine developers or more (NumDevs $\geq 9$), then 44.4% of those files would be vulnerable, which is considerably high given that only 3% of the system’s files were vulnerable$^{12}$. Thus, using NumDevs provides 14 times ($=44.4/3$) more discriminative power than random selection. Furthermore, for files with fewer than nine developers, (NumDevs$<9$), 2.9% of the files were vulnerable. However, those 44.4% vulnerable files only account for 9.5% (recall) of the known vulnerable files in the system, meaning more metrics with high discriminative power are required.

Table 4 shows some example critical values along with the precision, and recall. In all three case studies, NumDevs, NumCommits, and CNBetweenness all have high precisions when compared to the proportion of vulnerable files of 3%-6% found in Table 2, but the recalls are still low. The result of having all four metrics being correlated (from Section 4.3.1), but having low recalls means, that while the metrics are correlated with vulnerabilities, none of them individually account for all of the vulnerabilities.

---

$^{12}$ Taken from the 3% vulnerable file proportion reported in Section 4.2.1
<table>
<thead>
<tr>
<th>Metric</th>
<th>Example Critical Value</th>
<th>Precision</th>
<th>Recall</th>
<th>Example Critical Value</th>
<th>Precision</th>
<th>Recall</th>
<th>Example Critical Value</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNMaxEdge-Betweenness</td>
<td>350.0</td>
<td>20.2%</td>
<td>15.2%</td>
<td>10.0</td>
<td>26.4%</td>
<td>55.8%</td>
<td>2.5</td>
<td>15.6%</td>
<td>7.5%</td>
</tr>
<tr>
<td>NumDevs</td>
<td>9</td>
<td>44.4%</td>
<td>9.5%</td>
<td>9</td>
<td>42.3%</td>
<td>26.8%</td>
<td>6</td>
<td>13.7%</td>
<td>37.8%</td>
</tr>
<tr>
<td>NumCommits</td>
<td>33</td>
<td>36.0</td>
<td>8.4%</td>
<td>33</td>
<td>47.1%</td>
<td>37.2%</td>
<td>33</td>
<td>14.0%</td>
<td>19.7%</td>
</tr>
<tr>
<td>CNBetweenness</td>
<td>28,000</td>
<td>19.0%</td>
<td>11.8%</td>
<td>3,000</td>
<td>77.8%</td>
<td>16.3%</td>
<td>1,750</td>
<td>17.9%</td>
<td>7.5%</td>
</tr>
</tbody>
</table>

Note also that critical values can vary according to the project being studied. For example, the range of CNBetweenness is directly related to the number of developers and number of files in a system, so one critical value on one project may not have the same meaning in another project. Users of developer activity metrics could choose a desired critical value from historical analysis of their own project.

Alternatively, one could choose a critical value based on a fixed inspection rate. For example, suppose one wished to use the NumDevs metric to select files for security inspection and only had resources to inspect 10% of the system’s files. For the Wireshark data, the critical value of NumDevs for the inspection rate of 10% is five developers. That is, files that were changed by five developers or more account for 10% of the system’s files. Applying our critical value analysis, we find that such a model would have a precision of 31.0% and a recall of 63.4%. Table 5 contains example critical values for all three projects based on the critical value as determined by an inspection rate of 10%.
Our results for fixed inspection rate show that precision and recall vary for each project based on a fixed inspection rate. These results indicate that, while using developer activity metrics helps find vulnerabilities, one cannot expect consistent precisions and recalls from a critical value from a 10% inspection rate.

### 4.3.3 Predictability: How Many Vulnerable Files Can Be Predicted?

The predictability criterion is used to estimate how many vulnerabilities can be explained by combining all of the metrics into a single predictive model. As a secondary purpose, one can use predictability analysis as a simulation of how well one could have predicted vulnerabilities prior to release. Said another way, if the model can predict vulnerable files, then development teams can use the metrics to find vulnerabilities prior to release, and prioritize inspection and fortification efforts accordingly.

A key element of prediction is the *supervised model*. A supervised model is a method of combining multiple metrics into a single binary classification prediction (“neutral” or “vulnerable”)[69]. We used Bayesian networks as our supervised predictive model.

Supervised models require a *training set* and a *validation set*. In this study, we use cross validation to generate training sets and validation sets. For our prediction, we created a Bayesian network model and tested it using ten-fold cross-validation. Ten-fold cross-validation...
validation is performed by randomly partitioning the data into 10 folds, with each fold being the held-out test fold exactly once. The precision, recall, and inspection rate of the models can be found in Table 6.

| Table 6. Bayesian Network prediction, 10x10 cross-validation. |
|-----------------|-----------------|-----------------|-----------------|-----------------|
|                 | Linux Kernel    | PHP             | Wireshark       | Avg             |
| Precision       | 15.1%           | 29.3%           | 12.0%           | 18.8%           |
| Recall          | 31.6%           | 55.8%           | 32.8%           | 40.1%           |
| F₂              | 25.9%           | 47.3%           | 24.4%           | 32.5%           |
| Inspection Rate | 5.3%            | 10.6%           | 7.2%            | 7.7%            |

For the F-measure in our predictive analysis, we set our weight to two, meaning that we care twice as much about recall than precision. We reason that, in security prediction, allowing a vulnerable file escape to the field is worse than wasting effort on inspecting a file with no vulnerabilities.

Our results show a notably higher recall than with the individual metrics at critical points. However, the precision is lower to achieve this higher recall. One note of interest here is the low inspection rate across all three projects. If a team wanted to inspect files using the Bayesian network model on the Linux kernel, then they would only need to inspect 5.3% of the files and would find 31.6% of the vulnerable files.

The results of our predictability analysis show that the four developer activity metrics can be used to predict vulnerable files, but not all of the vulnerable files. This conclusion is a logical one: even if the models were perfect, we have no way of knowing if every vulnerable file caused by unfocused contributions or Linus’ Law.
In terms of general prediction models, other models outperform ours [15]. However, that developer activity metrics alone can predict a large percentage of vulnerable files (40.1% on average) is a useful result in terms of developer activity.

4.3.4 Cross-Project Analysis: Can Predictive Models be Transferred?

One of the drawbacks of using a predictive model in practice is that a model requires a set of examples (i.e. training data) to work form. Without training data, a predictive model could not be used on the first release of a product. However, since our three case studies show that developer activity metrics are consistently associated with security vulnerabilities, perhaps a general predictive model can be made.

In this analysis, we perform our prediction analysis with three-fold cross validation. Instead of the folds being randomly assigned, each fold is the full set of data set from a given project.

The precision, recall, and inspection rate for each model is shown in Table 7. Each model is denoted by what was used in the test set. For instance, the model where we trained on both the PHP and Wireshark data, then tested on the Linux kernel data is the “Linux Kernel” column. The fourth column is the non-weighted average of the first three columns.
Table 7. Cross-project Bayesian network prediction

<table>
<thead>
<tr>
<th></th>
<th>Linux Kernel</th>
<th>PHP</th>
<th>Wireshark</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision</td>
<td>12.7%</td>
<td>34.3%</td>
<td>15.2%</td>
<td>20.73%</td>
</tr>
<tr>
<td>Recall</td>
<td>47.9%</td>
<td>53.5%</td>
<td>25.4%</td>
<td>42.27%</td>
</tr>
<tr>
<td>$F_2$</td>
<td>30.8%</td>
<td>48.1%</td>
<td>22.4%</td>
<td>33.8%</td>
</tr>
<tr>
<td>Inspection Rate</td>
<td>9.5%</td>
<td>8.6%</td>
<td>4.4%</td>
<td>7.50%</td>
</tr>
</tbody>
</table>

The results of cross-project analysis show that, on average, both precision and recall increased when the model was trained on other projects. Even on an individual comparison, recalls in particular increased substantially in the cases of PHP and Wireshark. The largest decrease in performance was the Linux kernel, which decreased precision and inflated the inspection rate to achieve a higher recall.

These results are particularly surprising given the diverse set of projects we chose. The Linux kernel had 557 developers and Wireshark only had 19 (from Table 2), so the developer communities are quite different. Despite this difference, a model could be trained on these projects and used on another. From a prediction standpoint, this result indicates that vulnerability prediction is possible for a new development project with no history of documented vulnerabilities.

4.4 Limitations of Security Correlation Study

All of our developer activity metrics require version control data, and therefore change in the system. For developer networks, if a file has no commits to it during the period of study, the file has no developers in its history and therefore no measurement can be made. In the
three case studies, the RHEL4 kernel has 21 vulnerable files not changed prior to release, whereas PHP had one, and Wireshark had none. In such cases, those vulnerable files could not be used in our models and analysis.

All three of our case studies are in the C programming language, so our results should only be generalized to software developed in C.

Furthermore, since our data only includes known vulnerabilities, we cannot make any claims about latent, undiscovered vulnerabilities. As a result of the latent, undiscovered vulnerabilities, we cannot say that a low precision (i.e. a high occurrence of false positives) is actually indicative of real false positives, or that our model is finding more vulnerabilities in the system that have not yet been confirmed.

Lastly, security experts [28] claim that half of all security vulnerabilities are code-level and the other half are design-level. Our data sets are primarily a recording of code-level vulnerabilities from post-release maintenance. Thus, our results only apply to code-level vulnerabilities.

4.5 Contributions of Security Correlation Study

The objective of this research is to reduce security vulnerabilities by providing actionable insight into the structural nature of developer collaboration in open source software. In three case studies, we analyzed four metrics related to Linus’ Law and unfocused contributions. An empirical analysis of our data demonstrates the following observations:

- Source code files changed by multiple, otherwise-separated clusters of developers are more likely to be vulnerable than changed by a single cluster;
• Files are likely to be vulnerable when changed by many developers who have made many changes to other files; and

• A Bayesian network predictive model can be used on one project by training it on other projects, possibly indicating the existence of a general predictive model.

While the results are statistically significant, the individual correlations indicate that developer activity metrics do not account for all vulnerable files. From a prediction standpoint, models are likely to perform best in the presence of metrics that capture other aspects of software products and processes. However, practitioners can use these observations about developer activity to prioritize security fortification efforts or to consider organizational changes among developers.
5 Perception Corroboration Study

Over the last several years, social network analysis (SNA) has emerged as a popular method for studying the collaboration and organization of large software development teams. Researchers have been modeling networks of developers based on socio-technical connections found in software development artifacts. Most SNA studies apply graph theory to socio-technical networks of developers (often called developer networks) to generate SNA metrics. Connections between developers in socio-technical networks often originate from software development artifacts, such as version control change logs. Recent empirical case studies of several well-known software products have shown that SNA metrics are predictive of faults [40], [47], failures [37], [70], and vulnerabilities [32], [33]. Furthermore, researchers are developing visual tools [3], [53] and techniques [6] to aid practitioners in viewing, analyzing, and organizing software development teams via developer networks.

While prediction studies and tool studies showcase the utility of SNA metrics, we must ask the question: do SNA metrics measure what they purport to measure? Both researchers and practitioners alike need to know the extent to which the developer network and SNA metrics represent the reality of a software development project. For example, if the version control logs show that two developers are working on the same code in the same month, are they collaborating? Are well-connected developers (i.e. central according to SNA metrics) viewed as experts in the project by teammates? The answers to these questions underlie many of the assumptions of SNA research and its recommendations.
While development artifacts can provide a historical view of the development project, the developers themselves can provide another valuable perspective. Developer perceptions could support that the connections observed in the development artifacts represent actual socio-technical relationships. A study of the comparison between developer perceptions and the artifact-based developer network would help researchers and practitioners draw sound conclusions when analyzing a complex ecosystem of developers.

Therefore, the objective of this research is to investigate if social network analysis metrics represent socio-technical relationships by examining if developer networks can be corroborated with developer perceptions. To measure developer perceptions, we developed an online survey that is personalized to each developer of a development team [35]. The personalization is based on the SNA metrics taken from version control change logs of the developer’s project. Developers answered questions about other members of the team, such as identifying collaborators and the project’s experts.

A total of 124 developers responded from three open source projects: the Linux kernel, the PHP programming language, and the Wireshark network protocol analyzer. In this study, we provide an empirical analysis of those responses and their relation to the developer network and its derived SNA metrics.

The main contribution of this study is the empirical support that developer networks represent the real-world socio-technical concepts of collaboration, distance, and expertise. Specifically, we have found empirical support for all three of the following research questions:
• Q1: Developer network edges. If two developers work on the same source code file in the same month, then do they perceive they are collaborating?

• Q2: Developer network distance. Does the distance between two developers in the developer network represent the perceived degree of separation between those two developers?

• Q3: Developer network centrality. Does having a high centrality in the developer network indicate being reputed as a project expert by other developers?

5.1 **Developer Network Used in Perception Corroboration Study**

To define our developer network, we are looking for records of social or technical connections in the context of the project. We use the logs from the version control system to determine such connections. Our motivation is that if a developer knows enough about the code to enact a change, then two developers changing the same code means a socio-technical relationship (e.g. shared knowledge of that code) likely exists.

Our developer network is a graph where the vertices represent a developer on the team. Edges exist where two developers made a version control commit to the same source code file within one month of each other. Additionally, a single edge between two developers might seem too coarse for measuring socio-technical distance. To provide a finer granularity in distance measurement, we added a weight to the edges in our developer network. The weights of our edges are defined as the number of source code files in the system that the developers did not work on together. For example, if there are 1000 files in a system, and two developers worked on 10 different source code files together, then the weight on the edge
(representing distance) would be 990. In this study, we evaluate both weighted and unweighted edges separately.

5.2 Research Methodology

In this section, we discuss our online survey and the steps we took to conduct this study.

5.2.1 Online Survey

Our online survey consisted of six questions posed to the developers. The entire survey can be found in Appendix A.

For each question, the respondents were given the opportunity to provide feedback on the questions itself. On some of the questions, the developers clarified how they interpreted the question (e.g. the meaning of “work with” described in Question 2). We found that some questions were often interpreted differently among the developers. Based on that feedback, we only analyze three of the six questions in the survey.

Our online survey is personalized for each developer of a given project. The possible answers for each developer is slightly different based on the SNA metrics of the respondent.

The developers of each development team were each sent a solicitation email for the survey. Each email had a personalized link to the survey. Before the survey begins, the respondent must first verify his or her own identity, as shown in Figure 2.
Figure 2. Screenshot of the respondent verifying his or her identity in the survey.

After verifying his or her identity, each respondent answered a series of questions. The specific questions used in this analysis are discussed further in Section 5.4.

Our survey system was developed in Java, JSP, and JavaScript, running in Tomcat 5.5, MySQL 5, on a server running Red Hat Enterprise Linux 5.

5.2.2 Conducting the Study

We executed the following steps to perform the study.

**Step 1: Obtain version control data.** For each of the case studies, we obtained the version control data from the project websites. In each case study, the version control data was captured from July 1st 2008 through July 1st 2010 as the most recent two years of data available at the time of the study.
Step 2: Compute the developer network. We computed our developer network according to the definitions in Section 2 using a combination of our own Java scripts, the JUNG2 framework, and the MySQL relational database.

Step 3: Calculate geodesic path distances. Our geodesic path distance measures were calculated with both weighted and unweighted edges. The weights of our edges are defined as the number of source code files in the system that the developers did not work on together.

Step 4: Obtain emails for each developer. We obtained developer emails from public artifacts, including the version control system, mailing lists, and issue tracking databases. In all three studies, we were able to trace every version control ID to an email, although the process was manual.

Step 5: Develop online survey. See Section 5.2.1 for a description of our survey.

Step 6: Load geodesic path distances from the developer network into the survey. The formulation of some questions depended gathering developers of varying socio-technical distances from the respondent. We used the weighted geodesic path lengths from the developer network (defined in Section 5.1) as our measure in the survey.

Step 7: Solicit developers to take the survey. We sent out a solicitation email to every developer in each developer network, asking them to follow the link to our survey. The link contained a key that was specific to each developer so that the survey system could track the respondent’s answers. As an incentive, we put each developer in a drawing for a gift certificate to Amazon.com.
**Step 8: Process and analyze responses.** Before analyzing the data, we read all of the comments that people left on each question and processed the data. In some situations, we removed some responses as being non-participatory in the study (those responses are not counted in this study). We removed responses where respondents were attempting to circumvent input validation, or where respondents attempted to bias the results by providing obviously false information. We also removed questions from our analysis based on developer feedback.

### 5.3 Case Studies

In this section, we describe the details of the three case studies we performed. We intentionally chose projects from different domains with varying community sizes. In all three case studies, when we refer to “developers”, we are referring to the core group of people who are actively working on the project. In most open source development, these core developers are called “committers” because they have access to making commits to the development project.

Additionally, in all three case studies we only included commits to source code, which we defined as files ending in .c, .cpp, .S, and .h. We used the previous two years of development history, ranging from July 2008 to July 2010. Table 8 shows meta-data on our three developer networks, along with the response rates to the survey.
Table 8. Case study metadata.

<table>
<thead>
<tr>
<th></th>
<th>Linux Kernel</th>
<th>PHP</th>
<th>Wireshark</th>
</tr>
</thead>
<tbody>
<tr>
<td># Developers</td>
<td>226</td>
<td>76</td>
<td>28</td>
</tr>
<tr>
<td># Edges</td>
<td>827</td>
<td>492</td>
<td>203</td>
</tr>
<tr>
<td># Commits</td>
<td>46,201</td>
<td>7,975</td>
<td>10,659</td>
</tr>
<tr>
<td>Network Diameter</td>
<td>9</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td># Respondents</td>
<td>90 (40%)</td>
<td>18 (24%)</td>
<td>16 (57%)</td>
</tr>
</tbody>
</table>

5.3.1 Linux Kernel

We performed our case study on the most recent version of the Linux kernel at the time of this writing (Version 2.6.35). For the version control data from which developer activity metrics were computed, we used the Linux kernel Git source control repository.\(^{13}\)

In the Linux kernel, the version control repository makes an explicit distinction between the “committer” of change, and the “author” of a change. For example, if Alison submits a patch, but Rob commits the change, then Alison is credited as the author, and Rob is the committer. We used the “author” field in this study because the author is intended to be the person directly involved in the change to the system.

The Linux kernel had over 4,000 authors in the time period we examined. The vast majority of these people, however, made very small changes to the system. Since we are focusing on the core group of developers in this study, we only included the 226 developers who were specified as authors of at least 50 commits in the past two years. Of all the 46,201 commits in the previous two years, the 226 authors were on 67% (31,081) of the total commits. We obtained all 226 developer emails directly from the version control system.

\(^{13}\)http://git.kernel.org
Figure 3 depicts the Linux kernel developer network. The high density of edges highlights the large volume of development activity being done by multiple developers. Also, the entire network is spread out with a single group of central developers in the center. The wide diameter of the network (9 as shown in Table 8), supports the visual result that the network is spread out.

Figure 3. Visualization of the Linux kernel developer network
5.3.2 PHP Programming language

We performed a case study of the PHP programming language. The PHP programming language project includes development on the language itself, as well as extensions and modules. For this study, we focused on the developers working on the language itself. Unlike the Linux kernel, the main group of developers is more precisely defined as the people who are granted commit access to the Subversion repository. Thus, we focused our study on the group of committers to the PHP language itself.

We obtained the 76 email addresses for each developer by manually investigating their version control user ID in public sources such as the profiles and mailing list archives on PHP.net.

Figure 4 depicts the developer network for the PHP programming language. As with the Linux kernel, the central developers within this team are quite apparent in this layout. However, the overall spread of the network is much tighter, supported by the diameter metric being 4.
5.3.3 Wireshark network protocol analyzer

The Wireshark network protocol analyzer is a tool that can be used to aggregate and summarize data transported over a network. We focused our study on the group of committers using their Subversion repository.

We obtained the 28 email addresses for each developer by manually investigating their version control user ID from their public websites.

Figure 5 shows a visualization of the developer network for Wireshark. The central developers are not as easily seen, mostly because of the high degree of most developers.
Additionally, this community is much smaller (28 committers), so the spread of the network is much smaller, as supported by having a diameter of 3.

Figure 5. Visualization of the Wireshark developer network

5.3.4 Criterion for Developer Network Edges
For our developer networks, we defined edges where two developers made version control commits to the same source code files within the same month. That one month time period, which we will refer to as the edge window, is a parameter that can be tweaked for different software projects depending on the process and team culture.
Other studies [4], [53], [70] implicitly use an edge window of infinity, thus defining an edge where two developers change the same source code file at any time in history. We believe that such a definition did not incorporate the ephemeral nature of collaboration: as code changes, so do the socio-technical connections amongst the teammates. With an infinite edge window, a developer could potentially be connected to other developers for years after the code has completely changed. Thus, we sought to find a minimal, meaningful edge window according to the collaboration we observed in our case studies.

We arrived at our choice of the one month edge window through a manual investigation of the frequency of regular committers to features in all three case studies. Through reading version control logs, mailing lists, and issue tracking data, we identified regular committers to each project. We observed that even the most frequent committers to a project would sometimes drop off in commits for multiple weeks at a time. The main reasons we could glean from the artifacts included: temporarily working on other projects, vacation, and the product undergoing stabilization prior to release. Based on these observations, we decided that a 30-day edge window was a reasonable choice.

To understand the impact of the edge window parameter on number of developer edges, we plot edge window values in days against the number of developer edges in Figure 6.
We observed that the number of edges grows rapidly for 0-60 days, then slows down as the window widens out to a year. Interestingly, we found a strong logarithmic relationship ($R^2 > 97\%$ in all three case studies) between the width of the window and the number of edges in the developer network. These results indicate that, in our three case studies, the growth in the number of edges changes little when the edge window parameter becomes several months wide.

While we observed one month to be a reasonable edge window for our case studies, we do not believe that one month ought to be unilaterally applied to all developer networks. Developers are working on all three of our case studies commit code constantly, which

Figure 6. Width of the edge window and number of edges
cannot be said about all software projects. We believe that the edge window parameter ought to be individually determined for each project.

5.4 Perception Corroboration Analysis

Our main goal of analyzing this data was to examine if developer networks can be corroborated with developer perceptions. We examined specific research questions that were motivated by potential limitations of related work. Three questions we wish to answer are:

- **Q1: Developer network edges.** If two developers work on the same file in the same month, then do they perceive they are collaborating?
- **Q2: Developer network distance.** Does the distance between two developers in the developer network represent the perceived degree of separation between those two developers?
- **Q3: Developer network centrality.** Does having a high centrality in the developer network indicate being reputed as a project expert by other developers?

We address these three questions in the following sections.

5.4.1 Collaborators and Edges

Several studies employing developer networks have focused on the notion of collaboration (including our own studies), claiming that the developer network is an estimation of collaboration. The reasoning is that if two developers are working on the same source code around the same time, then some type of socio-technical relationship likely exists between the two developers, perhaps a collaboration connection.
To evaluate perceived collaboration in this study, we asked the developers whom they collaborate with in the context of project. Figures 6 and 7 show the research and survey question. The list of names from the auto-suggest list comes from the list of committers from the version control logs.

**Research Question:** If two developers work on the same file in the same month, then do they perceive they are collaborating? (Q1)

**Survey Question:** Whom on the [project] do you work closest with? Consider both online and in-person collaborators.

**Survey Answer:** Enter text of a developer’s name (auto-suggest list comes up with other developers on the project)

Figure 7. Question regarding collaborators and edges.

![Figure 8. Screen capture of the question regarding collaborators and edges](image-url)
To analyze this question, we first processed everybody’s responses for misspellings of names and other input validation concerns.

Of the 460 named collaborators across all projects, 87 (19%) of those collaborators were not found in the version control change logs. In many of those cases, a manual investigation of those names revealed that they were project organizers or some other role not related to development.

Next, for the collaborators found in the developer network, we examined unweighted geodesic distance between each respondent and his or her named collaborators. For example, if a respondent was directly connected to a named collaborator in the developer network, then the unweighted geodesic distance was one. We aggregated each of those unweighted geodesic distances between a respondent and the named collaborators and report the results in Table 9.

Table 9. Unweighted distances between respondents and their named collaborators

<table>
<thead>
<tr>
<th></th>
<th>Linux kernel</th>
<th>PHP</th>
<th>Wireshark</th>
</tr>
</thead>
<tbody>
<tr>
<td># Named Collaborators</td>
<td>241</td>
<td>51</td>
<td>50</td>
</tr>
<tr>
<td>% Collaborators with unweighted distance=1</td>
<td>53%</td>
<td>43%</td>
<td>84%</td>
</tr>
<tr>
<td>Mean unweighted distance</td>
<td>1.8</td>
<td>1.6</td>
<td>1.2</td>
</tr>
<tr>
<td>95% CI (normal)</td>
<td>(1.7, 1.9)</td>
<td>(1.4, 1.8)</td>
<td>(1.1, 1.3)</td>
</tr>
</tbody>
</table>

In all three case studies, at least 43% of the named collaborators are directly connected in the developer network. Furthermore, the mean unweighted distance from a developer to his or her collaborators was considerably close to one. Assuming a normal distribution of the unweighted distances, the confidence around the average distance is fairly tight. Thus, if a
developer is not directly connected to a named collaborator, then they typically are connected to someone who is connected to that collaborator (i.e. an weighted distance of 2).

Note also that a respondent could be connected to developers whom he or she did not name as a collaborator. We did not assume that a developer would be able to remember or even know who all of his or her collaborators are. Therefore, this particular question can only be used to confirm the positive identifications of collaborators, not the negatives. We address the non-existence of collaborator in our analysis of section 6.2.

These results indicate that perceived collaborators are being captured by edges in a developer network based solely on the version control change logs. In other words, collaborators in these three open source projects were often working on the same code within one month of each other. While this result substantiates that developer edges are linked to collaborators, one can see that not every named collaborator is represented by an edge. When interpreting developer networks, then, one should not assume that every collaborator is directly connected in the developer network, although empirically the distance between collaborators tends to be less than two.

5.4.2 Distance between Two Developers

Some of the most commonly-used SNA metrics in related work are centrality metrics. Specifically, betweenness (defined in Section 2.1) has been used heavily to measure a developer’s centrality in a network. The concept of betweenness is based on the notion of geodesic paths: if a developer is on a geodesic path between two other developers, then that
developer must be central to the network. Therefore, studying the validity of the length of a geodesic path helps us study the validity of centrality metrics as well.

To examine developer perceptions of distance, we provided the following question in our survey in Figures 9 and 10.

| Research Question: Does the distance between two developers in the developer network represent the perceived degree of separation between those two developers? (Q2) |
| Survey Question: In the context of the [project], what is your connection to the following people? |
| Survey Answer options (for each developer): |
| • A: I have never heard of this person before |
| • B: I recognize this name, but I don’t know much about them |
| • C: I know who this person is, but I have not worked with them directly |
| • D: I have worked with this person on this project |

Figure 9. Question socio-technical distance between two developers.
For each person, the survey showed 10 teammates from the developer network. Each respondent was given four options for each person (shown in Figure 8 as options A through D). We came up with the perceived distance levels and treat those levels on an ordinal scale with A being the most distant, D being the closest.

The survey chose those 10 teammates in the following manner. First, the system calculated the respondent’s weighted geodesic distance to each of the other developers in the project’s developer network. The survey then ranked all of the other developers by that weighted distance, and then chose developers of varying distances in the ranking. For example, if there were 50 other developers, then the system would rank those 50 developers by weighted geodesic distance to the respondent, and then return the developers ranked 1, 5,
10,…up to rank 50. The actual list of 10 developers was then shuffled before displaying in the survey.

Our motivation for this algorithm was this: if the geodesic distance to other developers represents the perceived distance then the ranking of developers by geodesic distance ought to match the ranking of developers by perceived distance. To measure the degree of agreement between two ranks, we use the Spearman rank correlation coefficient. The Spearman coefficient is useful when the underlying distribution of the random variables is unknown, yet they are still on the ordinal scale.

We calculated the Spearman coefficient for each respondent’s answer to the question, giving us a value in the range [-1,1]. We report the mean of the squared Spearman correlation coefficients.

We report in Table 10 the mean of Spearman coefficients between the reported perceived distance from the survey and the unweighted distance from the developer network.

<table>
<thead>
<tr>
<th>Respondents (10 data points per respondent)</th>
<th>Linux kernel</th>
<th>PHP</th>
<th>Wireshark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean Spearman coefficient $R^2$</td>
<td>0.40</td>
<td>0.55</td>
<td>0.36</td>
</tr>
</tbody>
</table>

In each case study, the mean Spearman coefficient was significantly greater than zero (using the formula shown in Section 2.4), indicating the existence of a correlation between the perceived socio-technical distances and the developer network distances. Although the
correlations are statistically significant, the strength of the correlations are not considerably close to one, indicating that the unweighted geodesic distance in a developer network does not always match developer perceptions.

Additionally, we applied the same analysis to the weighted distances to examine if the finer granularity in weighted edges more closely match developer perceptions. Table 11 contains means of the Spearman coefficients between the reported perceived distance from the survey and the weighted distance from the developer network.

**Table 11. Mean of squared Spearman rank correlation coefficients between weighted and perceived distances**

<table>
<thead>
<tr>
<th>Respondents (10 data points per respondent)</th>
<th>Linux kernel</th>
<th>PHP</th>
<th>Wireshark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Respondents (10 data points per respondent)</td>
<td>90</td>
<td>18</td>
<td>16</td>
</tr>
<tr>
<td>Mean $R^2$ (Spearman)</td>
<td>0.23</td>
<td>0.47</td>
<td>0.32</td>
</tr>
</tbody>
</table>

The means appear to be higher for weighted edges, however, in all three case studies, the difference were not statistically significant between weighted and unweighted means, indicating that there is no statistically significant difference between using weighted developer distances and unweighted developer distances. Thus, we do not have enough evidence to claim that weighted edges are any better than unweighted edges in measuring what developers perceive as socio-technical distance.

These results indicate that the perceived socio-technical distance between two developers is statistically correlated with the geodesic distance in the developer network.
5.4.3 Centrality and Reputed Expertise

Now that we have shown that geodesic paths in the developer network represent developer distances, the meaning of the betweenness centrality measure becomes more intriguing. Experts in SNA in other disciplines have argued that the betweenness centrality measures represent authority and visibility in the network [8].

According to our experience, open source projects deal with visibility and authority of developers quite often. Proponents of open source development have discussed the importance of a “meritocracy” in organizing development efforts [49]. The idea is that the more a developer works on the code, the more their work will be noticed and trusted, the more authority they will have on steering the project. Therefore, in the context of open source projects, visible authority is often tied to being an expert developer on a given project.

Open source developers often measure this level of merit by counting the commits a given developer made. However, as an example, if developer Peter works on code in relative isolation from the rest of the group, then he would have a large number of commits but have no collaborations with other members of the team, and therefore a low centrality. On the other hand, if developer Linus makes many commits to files that many other people are working on, he will likely have a higher centrality.

We hypothesize that, in the context of a meritocracy, central developers are regarded as experts in the project. Our question regarding expertise is shown in Figures 11 and 12.
**Research Question:** Does having a high developer centrality in the developer network indicate being a project expert? (Q3)

**Survey Question:** Whom on this team do you consider to be an expert on the [project]. Your answers can include anybody involved with the [project] (i.e. you don’t need to have worked with them, or even know them).

**Survey Answer:** Enter text of a developer’s name (auto-suggest list comes up with other developers on the project)

To analyze this question, we first processed everybody’s responses to ensure that the person they named was someone in the developer network. Furthermore, we manually checked for misspellings of names and other input validation concerns.
We then counted up the number of votes for each named expert. To be considered an expert, we only considered people with three votes or more. Self-voting was allowed, and one person could not vote for a person more than once.

Some respondents named experts who were not in the developer network, which we did not include in this study. None of the 15 named experts who were not in the developer network was named more than once, so this effect did not skew our results.

Table 12 shows the breakdown of the number of experts identified and how many of the total votes that the top five experts obtained.

<table>
<thead>
<tr>
<th>Table 12. Summary of Votes for Experts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux kernel</td>
</tr>
<tr>
<td>Number of votes</td>
</tr>
<tr>
<td>Number of experts with votes≥3</td>
</tr>
<tr>
<td>% of votes for the top 5 experts</td>
</tr>
</tbody>
</table>

In all three case studies, the top five experts accounted for the majority of the votes on expertise. In the case of PHP, this percentage was lower, indicating that the general regard for who was the expert was less widely agreed-upon than the other two case projects.

To examine if centrality is correlated with reputed expertise, we used the non-parametric Mann-Whitney-Wilcoxon (MWW) test. We tested for a statistically significant difference in our two centrality metrics developers regarded as experts and developers not regarded as experts. We apply our test to our two centrality metrics: degree and betweenness. Table 13 contains the results of those tests.
Table 13. Association between centrality and reputed expertise

<table>
<thead>
<tr>
<th>Centrality Metric</th>
<th>Statistic</th>
<th>Linux kernel</th>
<th>PHP</th>
<th>Wireshark</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Degree</strong></td>
<td>Mean for Expert</td>
<td>14.0</td>
<td>33.6</td>
<td>19.9</td>
</tr>
<tr>
<td></td>
<td>Mean for Non- Expert</td>
<td>6.2</td>
<td>8.8</td>
<td>11.4</td>
</tr>
<tr>
<td></td>
<td>Difference significant (p&lt;0.05)?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Betweenness</strong></td>
<td>Mean for Expert</td>
<td>0.02</td>
<td>0.06</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>Mean for Non- Expert</td>
<td>&lt;0.0001</td>
<td>&lt;0.0001</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td>Difference significant (p&lt;0.05)?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The betweenness measure can be interpreted as the percentage of total geodesic paths on which a developer can be found. Being on just 2% of the total geodesic paths, for example, is quite central for a single developer. As one can see from the results, experts tend to be at least 2% of the total geodesic paths on average. In every case study and for both degree and betweenness, reputed experts were statistically more likely to have a higher centrality than non-experts. Thus, our results indicate that developer network centrality is associated with being a reputed expert in the project.

5.5 **Limitations of the Perception Corroboration Study**

The largest limitation to this work is the validity of developer perceptions. Developers can be wrong about whom they collaborate with, which could influence our results positively or negatively. While perceptions are not perfectly trustworthy, we believe that comparing
developer perceptions with the artifacts brings us closer to better measuring the overall structure of collaboration in a software development team.

Not every developer on the project responded to the survey solicitation, and we do not know if our sample of developers was biased in some way. To mitigate this, we examined three different projects in an effort to achieve more generality.

Our questions regarding experts and collaborators required the user to remember people’s names. A respondent might be able to recall the name of an expert or collaborator. To mitigate this, we provided an auto-suggest list of the committers. Additionally, in our analysis, we did not assume that the named collaborators are a complete list of collaborators.

Lastly, since our approach included only committers of a project, our results do not generalize to the structure of non-coding participants surrounding a software development project. Such participants can include users, managers, and non-coding support. Our reasoning for this particular scope was based on the information found in the artifacts we chose to study.

5.6 Contributions of the Perception Study

The objective of this research is to evaluate the validity of SNA metrics by corroborating developer networks with the perceptions of developers of the projects involved. Our results indicate more support for researchers’ assumptions on what developer networks represent. In particular, we found empirical substantiation for the following questions:

- **Q1: Developer network edges.** If two developers work on the same file in the same month, then do they perceive they are collaborating?
• **Q2: Developer network distance.** Does the distance between two developers in the developer network represent the perceived degree of separation between those two developers?

• **Q3: Developer network centrality.** Does having a high centrality in the developer network indicate being reputed as a project expert by other developers?

In each case, however, not every notion was fully supported by developer responses. For example, while edges are closely linked to collaborations, not every edge represents collaboration and not every collaboration is represented by an edge. Thus, the developer network ought to be treated as an approximation. Nevertheless, the developer network is, in general, supported by developer perceptions. Our results can help researchers and practitioners understand the complex ecosystem of developer activity on software development teams.
6 Synthesis Study

Since beginning this research, many other researchers have begun to approach the study of software development teams from a socio-technical perspective. Most studies involve specific case studies with results that, if generalized, could explain how software development teams can work together effectively. Many of our studies in this dissertation fit closely with these works, so we provide this synthesis study as a means to integrate our work with others and as a foundation for future socio-technical, quantitative software engineering research.

The objective of this study is to guide both high-level to researchers by providing an overview of the status of current socio-technical research of software development teams. The results of this synthesis work also helps guide practitioners in organizing software development teams. We performed a literature survey of the most recent studies of software development teams from a quantitative, socio-technical perspective. Our selection, which covered the past ten years of the most relevant software engineering research venues, resulted in 35 papers. We classified our papers according to the types of results they present. In the course of our study, we found that a network-based paradigm emerged as the predominant form of empirically studying software development teams. We present that network-based paradigm in detail, then synthesize the studies’ results and formulate evidence-based conjectures for socio-technical research. Our conjectures are based on the several research questions that we found in the literature:

- How does the number of developers working on source code affect its quality?
How does developer network centrality relate to roles and responsibilities of individual developers in the development team?

How does developer network robustness relate to risk in software development?

How does the developer network differ between open source and closed source case studies?

This study is organized as follows. We describe our methodology and the results of the survey in Sections 6.1 and 6.2 respectively. We present the network-based paradigm in Section 6.3. We outline the trends and our conjectures in Section 6.4, and summarize our contributions in Section 6.5.

6.1 Methodology

We conducted a search of the academic literature to uncover the latest research that empirically analyzed the software development teams from a holistic viewpoint. We began by examining the past ten years (2000 through March 2011) of the following academic venues:

- (ICSE) International Conference on Software Engineering
- (FSE) Foundations in Software Engineering
- (TSE) Transactions in Software Engineering
- (TOSEM) Transactions on Software Engineering and Methodology
- (CSCW) Conference on Computer Supported Cooperative Work
- (CCS) Conference on Computer and Communication Security
- (MSR) Conference on Mining Software Repositories
We also searched the ACM and IEEE databases as well as Google Scholar, using the following search terms individually: developer metric, development team metric, socio-technical, socio technical, social network analysis, and software collaboration. For every paper we included in our list, we also examined the references of that paper for more potential papers. We included a paper if:

- The paper made use of social network analysis techniques of software development teams; or
- The paper performed an empirical analysis that quantified properties of software development teams, collaboration, or developers; or
- The paper presented a tool that utilizes quantified properties of teams, collaboration, or developers.

### 6.2 Recent Socio-Technical Research

In this section, we present the results of our literature survey, including a summary of the papers according to the types of results they presented and the socio-technical factors they examined.

#### 6.2.1 Literature Summary

Our first step in synthesizing the research was to determine what types of results the authors present. We observed four, non-orthogonal trends among the literature:

- A paper that is a **developer-focused empirical case study** covers a specific software project and its team, quantifying properties of that team, and presents the analysis.
- A paper that uses **social network analysis** is one that defines a graph of developers and uses that in the context of either analysis or tool support.

- A paper that is **tool-focused** is one where the primary purpose of the paper is to present a specific software system that utilizes a socio-technical perspective of software development teams.

- A paper that is **collaboration-focused** is one that has analysis dealing with how developers collaborate with each other, beyond measuring developers on an individual level.

For each of the papers we found, we denote whether the paper has the latter properties. A breakdown of our literature survey sources and these four properties can be found in Table 14.
<table>
<thead>
<tr>
<th>Paper(s)</th>
<th>Uses SNA?</th>
<th>Developer-Focused Empirical Case Study?</th>
<th>Tool-Focused?</th>
<th>Collaboration-Focused?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begel et al. [3]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Bird et al. [4], [6]</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Bird et al. [5]</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cataldo and Nambiar [12]</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chu-Carroll and Sprengle[13]</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ehrlich et al. [14]</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>de Souza and Redmiles [61]</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fritz et al. [17], [18]</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Herbsleb and Mockus [21]</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Herrmann et al. [22]</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Huang and Liu [24]</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hudepohl et al. [25]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Meneely et al. [29], [30], [32], [33], [36]</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Nagappan et al. [40]</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oezebek et al. [43]</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ohira et al. [44]</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Omoronyia et al. [45]</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Pinzger et al. [47]</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Robillard and Murphy [52]</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sarma et al. [53]</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Schroter et al. [55]</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Shin et al. [56]</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Singh [60]</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sindhgatta [59]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weyuker et al. [67]</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wolf et al. [70]</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Ye et al. [72]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yu and Ramaswamy [73]</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zimmerman [74]</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
As the literature has shown, applying a socio-technical perspective to software development teams yields many different kinds of studies. The majority of the literature utilized a network paradigm, which we will describe in detail in Sections 6.3 and 6.4.

6.2.2 Socio-Technical Factors
Throughout the literature, researchers have been approaching their study of software development teams from several different perspectives. Each of these perspectives investigates one or more socio-technical factors that influence how teams work on code. Our study uncovered the following socio-technical factors: project-specific knowledge, cross-project knowledge, expertise, communication, geographic distribution, and organizational complexity.

Developers who work on the same code can also share project-specific knowledge, such as knowing how specific modules work or the intended purpose of some functionality [17], [18]. Conversely, if two developers share project-specific knowledge, they may be more likely to work on the same code since they are familiar with the same parts of the system. Developers with experience in working on multiple projects gain cross-project knowledge, such as which types of software architectures are more appropriate for any given situation. A developer with a high level of cross-project knowledge may be one whom other developers approach for guidance [44]. Beyond knowledge, developers can differ from each other in their specific areas of expertise within a software project. Developers can specialize in different areas, such as security, database development, user interfaces, or the software’s
domain. The specialty of a developer can influence what each person works on, how they are approached by teammates, and whom they work with [14], [24], [59].

Beyond shared knowledge and expertise, developers also directly interact with each other in a variety of ways. Communication is a common socio-technical factor that can influence how teams are organized. Developers who communicate with each other can influence each other’s decisions in both a social and technical way [55], [70]. Even developers who do not directly communicate with each other can have awareness of each other’s actions as the product is developed. Many studies (e.g. [3], [13], [53], [61],[63]) have been devoted to analyzing and assisting developer awareness in a team. The degree to which a developer is aware of his or her teammates work can influence how that developer works on code. For example, if Sally becomes aware that her colleague is currently re-designing a module that Sally uses, she might work on some other part of the code base to avoid losing time.

On a larger level, teams can also be influenced by how they are located, that is, their geographic distribution. Globally distributed teams communicate much differently than collocated teams [1], [42], leading to differences in the way they write code together. One study has shown that global distribution is not as strong of a factor as many have assumed [5]. In addition to geographic distribution, the way that a team is formally organized can also have a significant impact on how the team works [40]. Organizational complexity can include factors like the number of company departments involved in a given piece of source code, or the level of code ownership applied to code. Metrics that quantify how complex the
organization of a given source code file have been shown to be statistically correlated with post-release failures [40].

6.3 The Socio-Technical Software Developer Network

Based upon our review, the predominant socio-technical paradigm for analyzing large groups of software developers is the socio-technical developer network. The term socio-technical originates from studies in psychology and sociology [70] that discuss how teams of people can be arranged to work on a set of tasks. The word “socio-technical” refers to a labor-related connection among people that could be influenced by social or technical factors. The term “technical” refers to the general idea of technicality (i.e. skill-related), not necessarily to technology-related activities. A socio-technical paradigm of software development teams, therefore, must reflect how individual developers are working on the project and, by extension, with each other in the context of their project.

The developer network is typically defined as a graph where nodes are developers and edges between developers come from a socio-technical relationship. Socio-technical relationships can include any labor-related connection between two developers, such as communication, coordination, or working on the same artifacts.

6.3.1 Connecting Developers

How one connects developers via socio-technical relationships determines the essence of the developer network. That is, the specific meaning of a given developer network originates from the way its edges are defined. For example, if the network is supposed to represent
collaboration, then each edge between two developers ought to represent that the two developers are collaborating.

Typically, when researchers implement a developer network they utilize historical records of socio-technical relationships to determine the nodes and edges. Researchers most often use logs from the version control system (e.g., Subversion, Git) as a record of what code developers were working on [4], [6], [24], [32], [33], [47], [53], [55], [56]. The motivation is that if a developer knows enough about the code to enact a change, then two developers changing the same code means they have some shared knowledge. Version control logs have the advantage of being complete: once the version control system is in use, every change to the source code can be logged. Additionally, basing edges on the version control logs means that edges represent specific parts of the system (e.g., a group of source code files). We used this to our advantage with the DNMaxEdgeBetweenness metric in Chapter 4. However, version control logs do not imply that developers are collaborating with, or are even aware of, each other. Our case studies in Chapter 5 showed that when two developers are working on the same code at the same time, they typically (but not always) perceive they are collaborating with each other.

Beyond working on the same code, developers also communicate with each other, which is another type of socio-technical relationship. Some researchers [70] apply communications logs (e.g., issue tracking comments, chat logs, or mailing list archives) to determine their edges in the developer network. Basing the edges on communication artifacts guarantees that two developers are aware of each other. However, communication logs have the
disadvantage of being noisy and incomplete. That is, not every conversation is going to be work-related and not every conversation will be recorded. Furthermore, communication edges would not trace to specific implementation artifacts (e.g. code) in the project, as does using the version control logs.

Having incomplete logs of team communication can significantly impact the structure of the developer network, especially in distributed development situations. Suppose developers operate in small, collocated teams, but those teams are all in different locations. Communication in collocated situations may be verbal and show no record, but communication between those teams may be recorded. The developer network in that situation may not truly represent the structure of communication.

In addition to defining when an edge exists, one can also define a distance between two developers in the form of an edge weight. Edge weights are numbers assigned to an edge that represent the distance between two developers. For example, if two developers only communicated once in a month, then that distance ought to be greater than when two developers communicated thirty times in a month. Defining edge weights have the added challenge of requiring interval scale validity [9], [27], meaning that one can add two distances and have the sum still represent the proper distance. Furthermore, while defining distance at this level of granularity appears to make the network a closer representation, no such edge weight has been introduced in the literature that has shown to be an improvement over unweighted edges. We examined one edge-weighting scheme (See Section 5.4.2) and
found no statistically significant difference in developer perceptions of collaboration between using the edge weights and using unweighted edges.

6.3.2 Incorporating Time in Developer Networks

In software development, the dynamic of the team can be constantly changing. As the team is reorganized or the scope of the project evolves, teammates may change whom they work with and whom they communicate with. As a result, the analysis of developer networks must take into account the time factor and the ephemeral nature of collaboration.

Fortunately, many of the development artifacts used to form developer networks have timestamps that can be leveraged for time-based analysis of developer networks. When direct communication is logged, the time of the communication can be used in the analysis. When the developer network uses version control logs, three methods have been used in the literature to account for time in the developer network: time window, edge window, and version control branches. Each method can be used in combination with another method. For time window parameters, the entire scope of the version control logs is defined over a fixed time window. A time window can be any period of time, perhaps based on a pre-defined software development lifecycle, or on calendar time. For example, if one used a time window on the version control logs for the year 2011, then every pair of developers who both changed a file in common in 2011 are connected by an edge. Many of the studies in our survey used some form of a time window [4-6], [25], [32], [33], [36], [40], [47], [55], [56], [67], [70].
One of the disadvantages of time windows, however, is that when they are considerably wide (e.g. one year), edges may exist where two developers are not socio-technically connected. A source code file can be completely changed in the course of a few weeks, so when two developers change the same file in the same year, they may not actually be working on the same code. To mitigate this code churn effect, some researchers [32], [33], [36], [56] have added an edge window parameter to their network. With an edge window, edges from the time window are filtered out if the two version control changes for a given edge are not within a specified period of time. For example, a time window of the year 2011 and an edge window of 30 days would mean that an edge can only exist where two developers changed the same file in the year 2011, within 30 days of each other.

Beyond edge windows and time windows, development teams can also manage the same source code on multiple branches in the version control system. Two developers who are working on separate branches, therefore, would typically not be working the same source code. To account for this factor, some researchers have found it appropriate to exclude edges even further when the two developers are working on different branches [36].

6.3.3 An Example Developer Network

To further demonstrate how a developer network can be formed, we provide an example. Table 15 shows an example version control log containing four developers working on source code files on various branches and at different times.
Table 15. Example log representing version control logs (VCLog)

<table>
<thead>
<tr>
<th>ID#</th>
<th>Developer</th>
<th>File</th>
<th>Date</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Laurie</td>
<td>A</td>
<td>December 31st, 2010</td>
<td>/trunk</td>
</tr>
<tr>
<td>2</td>
<td>Annie</td>
<td>A</td>
<td>January 1st, 2011</td>
<td>/trunk</td>
</tr>
<tr>
<td>3</td>
<td>Annie</td>
<td>B</td>
<td>January 1st, 2011</td>
<td>/trunk</td>
</tr>
<tr>
<td>4</td>
<td>Laurie</td>
<td>B</td>
<td>January 29th, 2011</td>
<td>/trunk</td>
</tr>
<tr>
<td>5</td>
<td>Laurie</td>
<td>C</td>
<td>February 15th, 2011</td>
<td>/trunk</td>
</tr>
<tr>
<td>6</td>
<td>Jason</td>
<td>C</td>
<td>February 25th, 2011</td>
<td>/trunk</td>
</tr>
<tr>
<td>7</td>
<td>Annie</td>
<td>D</td>
<td>May 11th, 2011</td>
<td>/trunk</td>
</tr>
<tr>
<td>8</td>
<td>Laurie</td>
<td>D</td>
<td>May 19th, 2011</td>
<td>/trunk</td>
</tr>
<tr>
<td>9</td>
<td>Tao</td>
<td>D</td>
<td>May 23rd, 2011</td>
<td>/trunk</td>
</tr>
<tr>
<td>10</td>
<td>Tao</td>
<td>C</td>
<td>May 26th, 2011</td>
<td>/trunk</td>
</tr>
<tr>
<td>11</td>
<td>Tao</td>
<td>E</td>
<td>August 1st, 2011</td>
<td>/branches/version1</td>
</tr>
<tr>
<td>12</td>
<td>Jason</td>
<td>E</td>
<td>August 22nd, 2011</td>
<td>/branches/version2</td>
</tr>
</tbody>
</table>

For this example, we are using a time window of the 2011 calendar year, an edge window of 30 days, and applying version control branches to filter our edges. Immediately, entry #1 is removed because it is not in the 2011 calendar year as defined by our time window. The first edge is formed with entries #3 and #4, where Annie and Laurie worked on file B within 30 days of each other. Next, entries #5 and #6 give us the next edge between Laurie and Jason on file C. In May, three people (Tao, Annie, and Laurie) all worked on the same file within 30 days of each other, so three more edges are added between all three of those developers. Even though Tao also worked on file C (#10), he did not work on the file within 30 days of Jason working on it (#6), so Tao is not connected to Jason. Furthermore, entries #11 and #12 indicate that Jason and Tao both worked on file E, but on different branches, so they are still not connected. Therefore, the developer network from this example log can be found in Figure 13.
In the course of our own research, we developed a query that generates the edges for a developer network, which can be found in Figure 14. Our query language is a rough representation of the Structured Query Language (SQL), and Table 15 is called VCLog.

```
SELECT Dev1, Dev2, COUNT(*) AS NumFiles FROM
   (SELECT DISTINCT c1.developer AS dev1,
            c2.developer AS dev2
    FROM VCLog v1,VCLog v2
    WHERE v1.file=v2.file
    AND v1.developer <v2.developer
    AND abs(days(v1.date, v2.date)) < 31
    AND year(v1.date)=2011
    AND year(v2.date)=2011
    AND v1.branch=v2.branch)
GROUP BY dev1,dev2;
```

The query returns a list of edges with three columns: Dev1, Dev2, and NumFiles representing the number of files that the two developers worked on together. The inner query
works by first crossing the version control log (VCLog) with itself, looking only for connections between two developers that fit our criteria of within the time window, within 30 days of each other, and the same branch. The query makes Dev1 always alphabetically before Dev2 to prevent duplicate edges. The outer query then aggregates all instances to compute the NumFiles count in case we wish to apply a weighting to the network based on number of files.

### 6.3.4 Variations on the Developer Network

One variation on the developer network is the contribution network [47]. A contribution network has two types of nodes: a developer node and a source code file node. Edges exist where developers made version control commits to files. By definition, the graph is bipartite since developers are connected only to files, and files are only connected to developers. Section 4.1.2 contains a more detailed definition and an example of a contribution network.

The goal of the contribution network is to identify situations where files had an “unfocused contribution” from developers. Sometimes a file may have only a few developers working on it, but those developers are working on my other files at that time. As a result, those files may not receive the focus required to find, faults and vulnerabilities. Using a contribution network, one can use centrality measures to the file nodes to identify files that had an unfocused contribution [47].

One potential alternative that we have observed in the network analysis literature [8] but have not observed in software engineering research is a fully-connected developer network. In a fully-connected developer network, an edge exists between every two developers in the
network, with an edge weight representing the socio-technical distance between the two developers. By default, every edge has a minutely small edge weight (i.e. a large socio-technical distance). As more evidence of socio-technical connections are found, the edge weight would decrease between the two developers. This approach mitigates the risk of having missing or mistaken edges in the network, such as when having two co-workers who only talk to each other in person. The disadvantage of this approach, however is that, depending on the data structure representation of the graph, computation time for computing geodesic paths would also grow.

6.4 Trends and Conjectures

Many of the sources in recent past have covered a variety of open questions and provide empirical evidence that lead us in several different directions. In this section, we present the overarching research questions that the sources have covered and provide a conjecture for each question based upon the current pool of results. With time, we hope to see these conjectures to be confirmed or refuted as more empirical analysis surfaces.

6.4.1 Number of Developers Changing a File

With respect to the socio-technical paradigm, the simplest and most common metric used in the literature is the “number of developers” metric, which we will call NumDevs [25], [32], [33], [40], [47], [67]. Defined as the number of distinct developers who changed a given file, NumDevs typically relies on the version control change logs.

Historically, NumDevs has been introduced as a companion to code churn metrics [39], [47], [56], which are metrics that quantify how much the code has changed. Code undergoing
high churn means that the code is going through many additions and deletions in the source code. Like NumDevs, code churn metrics are also typically gathered from version control change logs. However, NumDevs as a metric can be orthogonal to code churn. A file can be changed by many developers while not undergoing much code churn. Likewise, a file can undergo high code churn by a single developer.

The idea behind NumDevs is to measure how many people are actually changing a file, which could be an indicator of a variety of problems in the team. Is having just one developer also correlated with failures and vulnerabilities? What about 200? If NumDevs is equal to one for a file, then only one developer has been changing that file, which could indicate that the developer’s changes are going unchecked by the rest of the team. On the other extreme, if NumDevs has a considerably high value, for instance 200, then that means hundreds of developers are changing a given piece of source code, making the coordination effort among those developers considerably difficult, possibly lowering the quality of that code. If both a lower bound and an upper bound exist, then NumDevs is not a monotonic metric, but has a “sweet spot” between too few developers and too many.

Several studies observed a statistically significant correlation between NumDevs and external quality measures [25], [40], [67]. In security, our correlation study (see Chapter 5) of three projects found a positive, statistically significant correlation between files changed by many developers and having post-release security vulnerabilities. In reliability, Hudepohl et al. [25] Weyuker et al. [67], and Nagappan [40] each developed a predictive model to find faults in their respective products. In both cases, they found that NumDevs was found to be
positively associated with faults, although Weyuker et al. observed that adding NumDevs to their model did not significantly improve the model’s prediction capability. All of the studies conclude that NumDevs has a general trend to be positively correlated with failures and vulnerabilities, but none so far have examined a lower bound for NumDevs.

Therefore, we present the following conjecture regarding the number of developers:

**Conjecture 1.** The number of developers changing a file has a “sweet spot” such that a file can be changed by either too many or too few developers in any given project.

The actual value of a “sweet spot” would most likely vary from team to team and project to project. The main alternative to this conjecture is that the number of developers is monotonic, that is, with every new developer being added to a file means an increase in the probability of injecting or missing a fault or vulnerability.

### 6.4.2 The Importance of Central Developers

Beyond NumDevs, one of the most commonly-used developer networks metrics deals with centrality. Network centrality is a measure of how a node is connected to the rest of team, where central nodes are “well-connected” by having both many neighbors and connected to other central nodes. Two commonly-used centrality metrics are degree and betweenness (defined in Chapter 2). In terms of a developer network, a central developer is one who works on code that other developers are also working on. Furthermore, a developer may become central by (a) being connected to many other developers; or (b) being connected to other central developers. Social network analysis typically interprets centrality as
“importance” or “authority” [8], so does that mean we ought to interpret developer centrality as having importance and authority in the project?

Several empirical studies have indicated that central developers are important or authoritative to the project. One of our studies of a Nortel product [37] showed that files changed by non-central developers are more likely to have failures. Furthermore, our perception corroboration study from Chapter 6 showed that central developers are more likely to be regarded by their peers as “project experts”. Lastly, in our case study of the open source OpenMRS healthcare system, we found that central developers are likely to have the authority to approve solutions to issues in the system [29], [34].

The reasons for having centrality and importance be so closely connected may stem from the idea of a meritocracy, discussed by Eric Raymond in the Cathedral and the Bazaar [49]. In open source development, developers are often given authority in a project once they have proven themselves to be competent contributors to the project itself. The evidence of one’s expertise, therefore, lies in making actual code changes to the project itself. The more a developer changes the project, the more visibility they gain from other community members, and therefore they have more authority according to the community. In a developer network, if a developer is able to contribute changes to source code files that central developers are also working on, then they will become both more visible and central to the network. In terms of expertise, working on the same code as a project expert also provides opportunity for communication and knowledge transfer.
Since the developer network is primarily defined around a single project, the kind of knowledge transferred between a central developer and a non-central developer is mostly project-specific knowledge. Therefore, based upon the evidence of the meaning of centrality metrics from the correlations we have observed in case studies, we believe the next step for centrality studies is to confirm or deny the following conjecture:

**Conjecture 2.** Developer centrality indicates the importance and authority of a developer in terms of project-specific knowledge.

If this conjecture proves to be true, we have a way of identifying developers who are critical to the project and ought to be freely associated with in the team. As a result, central developers ought to be more involved in changes that could affect the entire system, such as non-functional aspects of the project like performance or security.

### 6.4.3 Network Robustness and Developer Turnover

Another class of measures used in social network analysis is *robustness*, which is the measure of what would happen to the network if it underwent a change such as losing a node or an edge. In terms of software development teams, a relevant analogy of network robustness is gaining or losing developers. Brooks warns us that adding new developers to a team requires an enormous amount of training time [10]. Conversely, losing an expert developer can also mean significant knowledge loss. Furthermore, losing a developer can mean that the typical communication pathways in a team are disrupted.

The developer network can operate as a model of what happens when a developer is lost from a project. Specifically, robustness measures perform a simulation where one or more
nodes are removed from the network, and then records how the network is affected. For example, if the network becomes disconnected into several components by removing one developer, then perhaps more connections ought to be made between developers. As another example, if losing one developer causes a major disruption to geodesic path calculations, then the communication pathways would be disrupted and more connections ought to be made between developers.

To our knowledge no robustness studies have been performed on software development teams. The concept of losing developers has been discussed as a risk factor, often being called the “truck factor” [7]. That is, what is the risk to the team if a developer is hit by a truck and is no longer on the team? Beyond considering developer loss as a factor, however, no empirical evidence has been presented. Based up on the evidence that developer networks represent the structure of the team, we believe that robustness metrics could be useful in risk assessment activities. Thus, we present the following conjecture:

**Conjecture 3.** Poor developer network robustness represents a high risk to the software development team.

A confirmation of this conjecture could inform risk analysis of losing specific people on a development team and the specific recommendations could be enacted early in the software development lifecycle.

6.4.4 Open Source Development and Proprietary Development

As experts have pointed out [49], [68], open source development and proprietary (i.e. closed source) development have substantial differences in how they operate. Open source
development often relies on utilizing large communities of users who are interested in improving the product, whereas proprietary development involves formally organizing the development effort from its inception. From an organizational and management perspective, these two types of teams are considerably different.

However, in many of our case studies, [30], [32-35], [37], [56] we have noticed that open source and proprietary teams share many similarities in their developer network. Both types of networks contain large disparities in centrality, where the network contains a few central developers and many non-central developers. Additionally, we have noticed that developer networks from both open source and proprietary teams often exhibit the network analysis property of being scale-free [8]. Also called “small world” networks [6], scale-free networks contain a degree distribution that follows a power law, which means that as more nodes are added to the network, the maximum distance between nodes does not grow (i.e. the network diameter does not change). Keeping the networks a small world is often accomplished by taking non-central nodes and incrementally increasing their centrality over time by connecting them to the other central nodes. This tacit “promotion” of a node then allows for many more non-central nodes to be connected to the network and still keep the maximum distance between nodes low. The end result is a tightly-connected, “inner core” of nodes. Researchers software teams have been observed [6], [37] this “inner core” of nodes in several software teams. Scale-free networks are found throughout social network analysis studies, particularly in collaboration networks [2], [20].
Furthermore, the arguments we made for the number of developers and developer centrality can be applied to both open source and proprietary software. Since the developer network is indicative of how the team is changing source code together over time, perhaps the structure of a developer network does not change much from open source to closed source software. Based upon our observations and the observations of several studies, then, we conjecture the following:

**Conjecture 4.** Developer networks are structurally indistinguishable between open source and proprietary software development teams.

If this conjecture proves true, then we know that the developer network represents how people self-organize to solve problems regardless of the process. If the conjecture proves false, then the specific structural differences between developer networks can help better structure development teams.

### 6.5 Summary of the Synthesis Study

Our objective with the synthesis study is to provide a status of the current state of research of development teams that use a quantitative, socio-technical perspective. We provided a synthesis of current research in software engineering with common practices and measurements used in social network analysis. We provided a detailed view on the most commonly-used paradigm in socio-technical research: the developer network. Our synthesis resulted in a collection of conjectures that provide insight into how software development can be improved with quantitative socio-technical studies. Researchers can look to these conjectures as directions to take future research projects. Software developers and managers,
guided by our conjectures, can observe their own teams and look for opportunities to improve the organization of their team based upon the empirical evidence we currently have.
7 Contributions

The two broad contributions of this research are (a) predictive models for finding security vulnerabilities in software prior to its release; and (b) insight into how effective software development teams are organized. The observation of statistically significant correlations between developer activity metrics and post-release security vulnerabilities provides evidence that a relationship between developer collaboration and software security also exists. More specifically:

- Source code files changed by 6 developers or more are four times more likely to have at least one post-release security vulnerability.
- Vulnerability prediction models based on developer activity metrics can be used across different software development projects.
- If two developers changed the same source code within the same month, they typically perceive they are collaborating with each other.
- The degree of separation between two developers in a developer network typically represents their perceived socio-technical distance.
- Having a high centrality in a developer network is associated with being reputed as being a project expert.

In addition to our empirical work, we provided a synthesis of the quantitative research performed on software development teams. In our synthesis, we provided detailed motivation and definitions for the most commonly-used structure of studying development teams: the
developer network. Furthermore, combining the results of different case studies and with common measures of social network analysis, we provided the following four conjectures.

- **Conjecture 1.** The number of developers changing a file has a “sweet spot” such that a file can be changed by either too many or too few developers in any given project.

- **Conjecture 2.** Developer centrality indicates the importance and authority of a developer in terms of project-specific knowledge.

- **Conjecture 3.** Poor developer network robustness represents a high risk to the software development team.

- **Conjecture 4.** Developer networks are structurally indistinguishable between open source and proprietary software development teams.

Providing empirical evidence that can either confirm or deny these conjectures would help practitioners be better informed in how to structure their software development team.
8 Future Work

While we have introduced several metrics for software development teams, we do not claim that our metric suite is comprehensive. Each of our developer activity metrics are designed to capture specific situations in teams, which leaves room for more developer activity metrics in the future. One unexplored socio-technical factor is quantifying cultural differences among globally distributed teams. While one study has shown that globally distributed development is still effective in one project [5], our discussions with practitioners have often indicated that culture is a large factor in a product’s success.

Lastly, developer activity metrics have given us insight into the reputed expertise of a developer as being related to how they are socio-technically connected to other developers. This result gives us a particularly good direction in terms of security. If a central developer is a reputed expert on a project, then perhaps the security training ought to be prioritized toward central developers. If central developers are often working with many other people, then that security knowledge has a better chance of being spread throughout the team, resulting in a better security posture of the product. Our results so far have indicated that central developers are experts, but future research could potentially confirm whether security training on central developers can have a significant impact on the security posture of a software product.
REFERENCES


APPENDIX
8.1 APPENDIX A. Developer Perception Survey

The survey in its entirety is provided here. Based on respondent feedback, we do not present or analyze the results of questions one, two, and four. Additionally, wherever [project] appears, the survey would replace with the name of the project (e.g. Linux kernel project). A description of the type of input the survey allowed is in italics.

Question 1. On the [project], I perform the following tasks (check all that apply).

☐ Write code
☐ Write and/or execute tests
☐ Design software
☐ Manage people
☐ Inspect other peoples' code
☐ Fix defects.
☐ Answer questions from customers
☐ Answer technical questions from fellow developers
☐ Steer the overall direction of the project

The user can check any of the above roles. Not checking any role is allowed.

Question 2. In your estimation, how many different members of this project have you worked with in the last month on the Developer Survey project? Include in your count both in-person and online interactions. Do not include yourself in this count.

The user can enter a positive number for this question.
Question 3. Next, in the context of the [project], what is your connection to the following people? (See Figures 8 and 9 in Section 7.2 for more information).

Question 4. Consider the following scenario. Suppose you are developing a new feature for the Developer Survey project, and you realized that your changes could make the system insecure if your implementation is not correct. You decide to contact some of your colleagues to inspect your feature to ensure that no security vulnerabilities are being introduced.

What factors are most important to you in deciding who to work with in this situation?

Please assign 24 points to each of the following 8 factors, giving a higher weight for a higher importance.

- I work with this person frequently.
- This person is conveniently located near me.
- This person knows a lot about software security in general.
- This person has worked on similar features to this one before.
- In the past, this person has worked on parts of this project with high security risk.
- This person is highly experienced in software engineering.
- Someone I respect recommended this person.
- This person is my superior.

The user can input a positive numbers for each bullet point. The system only allows answers that add up to 24 points.
Question 5. Next, who on the [project] do you work the closest with? Consider both online and in-person collaborators.

The user can input any name or email address. An auto-suggest list pops up with the names and emails of other committers.

Question 6. Next, who on this team do you consider be an expert on the [project]? Your answers can include anybody involved with the [project] (i.e. you don't need to have worked with them, or even know them).

The user can input any name or email address. An auto-suggest list pops up with the names and emails of other committers.