

Strengthening the Empirical Analysis of the Relationship between Linus' Law and Software Security

Andrew Meneely and Laurie Williams

North Carolina State University

{apmeneel, lawilli3}@ncsu.edu

ABSTRACT

Open source software is often considered to be secure because large developer communities can be leveraged to find and fix security vulnerabilities. Eric Raymond states Linus' Law as "many eyes make all bugs shallow", reasoning that a diverse set of perspectives improves the quality of a software product. However, at what point does the multitude of developers become "too many cooks in the kitchen", causing the system's security to suffer as a result? In a previous study, we quantified Linus' Law and "too many cooks in the kitchen" with *developer activity metrics* and found a statistical association between these metrics and security vulnerabilities in the Linux kernel. In the replication study reported in this paper, we performed our analysis on two additional projects: the PHP programming language and the Wireshark network protocol analyzer. We also updated our Linux kernel case study with 18 additional months of newly-discovered vulnerabilities. In all three case studies, files changed by six developers or more were at least four times more likely to have a vulnerability than files changed by fewer than six developers. Furthermore, we found that our predictive models improved on average when combining data from multiple projects, indicating that models can be transferred from one project to another.

Categories and Subject Descriptors

D.2.8 [Software Engineering]: Metrics – *process metrics, product metrics.*

General Terms

Measurement, Security, Human Factors

Keywords

Developer network, contribution network, vulnerability, metric

1. INTRODUCTION

Open source software is often considered to be secure [7, 22]. One factor in this confidence in the security of open source software lies in leveraging large developer communities to find vulnerabilities in the code [7, 22]. 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* [17]. 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. Therefore, our research objective is *to reduce security vulnerabilities by providing actionable insight into the structural nature of developer collaboration in open source software.*

In a previous study [11], we performed an empirical analysis of Linus' Law and unfocused contributions in the open source Red Hat Enterprise Linux 4 (RHEL4) kernel¹. 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.

To examine if our previous results generalize beyond a single case study, we perform our analysis on two additional projects: the PHP programming language² and the Wireshark³ network protocol analyzer. We also updated our RHEL4 data set with 18 additional months of vulnerabilities found since our previous analysis to examine if our results still hold. Lastly, we perform a cross-project analysis to examine if predictive models can be trained on multiple projects and applied to another project.

The rest of this paper is organized as follows. Sections 2 and 3 cover background of our analysis and developer activity metrics. Section 4 describes the three case studies. Section 5 presents the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Empirical Software Engineering and Measurement, September 16-17, 2010, Bolzano-Bozen, Italy. Copyright 2010 ACM XXXXXXXX...\$5.00

¹ <http://www.redhat.com/rhel>

² <http://php.net>

³ <http://wireshark.org>

results of the empirical analysis and a discussion. Sections 6, 7, and 8 discuss limitations, related work, and summarize the study.

2. BACKGROUND

Our empirical analysis involves quantifying measures of social networks and binary classification. In this section, we provide background with regard to network analysis and binary classification.

2.1 Network Analysis

In this paper, we use network analysis to quantify how developers collaborate on projects. We use several terms from network analysis [3] and define their meaning with respect to developer groups and unfocused contributions in Section 4. In this section, we define terms used in both analyses of developer groups and unfocused contributions.

Network analysis is the study of characterizing and quantifying network structures, represented by graphs [3]. In network analysis, vertices of a graph are called nodes, and edges are called connections. A sequence of non-repeating, adjacent nodes is a path, and a shortest path between two nodes is called a geodesic path (note that geodesic paths are not necessarily unique). In the case of weighted edges, the geodesic path is the path of minimum weight. Informally, a geodesic path is the “social distance” from one node to another.

Centrality metrics are used to quantify the location of a node or edge relative to the rest of the network. In this study, we use the **betweenness** metric to quantify the centrality of a node in a network. The betweenness [3] of node n is defined as the number of geodesic paths that include n . Similarly, the edge betweenness of edge ℓ is defined as the number of geodesic paths which pass through ℓ . A high betweenness means a high centrality.

2.2 Binary Classification

To study the security of a system, we use a nominal metric defined over each file: whether or not a file is vulnerable or neutral. We consider a file to be vulnerable if the file was found to have at least one vulnerability that required a patch after release. A vulnerability is “an instance of a [fault] in the specification, development, or configuration of software such that its execution can violate an [implicit or explicit] security policy”. [8]. We consider a file with no *known* vulnerabilities to be “neutral”.

Since our security metric is nominal, our analysis is based on binary classification. A binary classifier can make two possible types of errors: false positives (FP) and false negatives (FN). 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). For evaluating binary classification, we use recall, inspection rate, and the F-measure.

- Recall (R) is defined as the proportion of vulnerabilities found: $R = TP / (TP + FN)$.
- Inspection Rate (IR) is the proportion of total files that were classified as vulnerable: $IR = (TP + FP) / (TP + TN + FP + FN)$.
- Precision (P) is defined as the proportion of correctly predicted vulnerable files: $P = TP / (TP + FP)$.

Optimally, IR is minimized, but Precision, Recall, and AUC 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 β 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 β represents the weight to which one assigns recall over precision [23].

$$F_{\beta} = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{\beta^2 \cdot \text{precision} + \text{recall}} \quad (1)$$

3. DEVELOPER ACTIVITY METRICS

In our case study, 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.

From a source code perspective, some files are contributed to by many developers who are also making contributions to many other files. Other files are essentially “owned” by one or a small number of developers.

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 two networks: developer networks and contribution networks, as will be discussed in Sections 3.1 and 3.2, 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 5.

3.1 Diversity in Perspectives

In his essay on open source development [18], 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’ 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

Table 1: Developer activity metrics

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

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 over the entire project and how strongly connected these clusters are. Two separate groups may be working on similar areas without working together. According to Raymond’s reasoning, 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* [1, 6, 12]. 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*. A cluster of nodes is a *set* of nodes such that the number of intra-set connections greatly outnumber the number of inter-set connections [3]. A cluster of developers, then, has more connections within the cluster than to other developers. 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 [5] for discovering developer clusters. Edge betweenness is defined similarly to node betweenness, only for edges: the number of geodesic paths in the network that include a given edge. 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 a 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.

3.2 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* [17] 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 *who* is making changes of an unfocused contribution. Thus, we add a third, more specific metric to our study: *CNBetweenness*.

The CNBetweenness metric is calculated from a *contribution network* [17]. 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`.

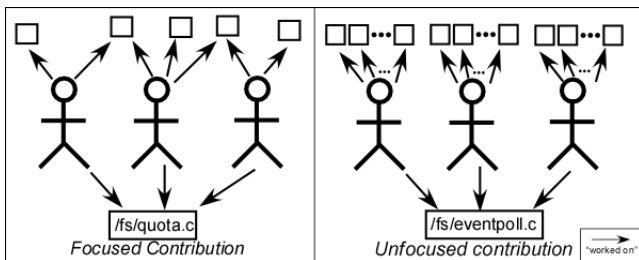


Figure 1: Examples of focused and unfocused contributions

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. CASE STUDIES

We performed our analysis on three software projects: the Linux kernel, the PHP programming language, and the Wireshark network protocol analyzer. 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 4.1 through 4.3). 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. 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`.

4.1 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 database⁴, the National Vulnerability Database (NVD)⁵, and the RHSR security metrics database⁶. In our previous study, our data set had 205 vulnerable files traced from vulnerabilities reported between February 2005 and July 2008. Our updated 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 repository⁷. 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.

4.2 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 database⁸, 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 <http://bugzilla.redhat.com>

⁵ <http://nvd.nist.gov>

⁶ <http://redhat.com/security/metrics>

⁷ <http://www.kernel.org>

⁸ <http://bugs.php.net>

Table 2: Summary of case study projects

	RHEL4 Linux kernel	PHP language	Wireshark network analyzer
Total number of files (.c, .S, .h)	14,285	1,039	2,688
Number of files changed (total studied)	10,454	776	2,541
Number of developers	557	84	19
Development time	16 months	24 months	24 months
Percentage of files changed in development time studied	73%	75%	95%
Total number of vulnerable files	284	44	67
Percentage of files changed in development time fixed for a vulnerability	3%	6%	3%
Total number of commits	9,946	4,771	6,846

4.3 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 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.

5. EMPIRICAL ANALYSIS

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

Section 5.1: Are developer activity metrics related to vulnerable files?

Section 5.2: 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 5.3: How many of the vulnerable files can be explained by the metrics?

Section 5.4: 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 [19]. Of the four questions, the first three were initially discussed in our original study [11], and the fourth question is new to this study.

In this study we use the three validation criteria (association, discriminative power, and predictability) to evaluate the strength of the relationship between the developer activity metrics and

security vulnerabilities. We used SAS⁹ v9.1.3 for our statistical analysis and Weka v3.6.0[23] for the Bayesian network prediction model.

5.1 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 other metrics validation studies [19], we use the non-parametric Mann-Whitney-Wilcoxon (MWW) test for difference in averages. Three outcomes are possible from this test:

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

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

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.

⁹ <http://www.sas.com/>

Table 3: Mann-Whitney-Wilcoxon test for association

Metric	Linux Kernel			PHP Language			Wireshark Network Analyzer		
	Neut. Avg	Vuln. Avg	MWW p-value	Neut. Avg	Vuln. Avg	MWW p-value	Neut. Avg	Vuln. Avg	MWW p-value
DNMaxEdgeBetweenness	18.4	124.2	p<0.0001	3.2	13.9	p<0.0001	0.4	1.2	p<0.0001
NumDevs	2.2	5.0	p<0.0001	1	6	p<0.0001	3	6	p<0.0001
NumCommits	4.0	14.2	p<0.0001	2	18	p<0.0001	4	13	p<0.0001
CNBetweenness	3602.0	12,673.0	p<0.0001	0.0	588.0	p<0.0001	79.1	630.7	p<0.0001

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 underwent many changes. This result is supports the “code churn” effect found in other studies [12, 13, 20] 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.

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

By evaluating the discriminative power [19] 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 next¹⁰.

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 changed nine developers or more (NumDevs >= 9), then 44.4% of those files would be vulnerable, which is considerably high given that

only 3% of the system’s files were vulnerable¹¹. 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 5.1), but having low recalls means, that while the metrics are correlated with vulnerabilities, none of them individually account for all of the vulnerabilities.

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 developer 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.

¹⁰ E.g. NumDevs has a much smaller range of values than CNBetweenness, so the size of the difference in averages cannot be compared

¹¹ Taken from the 3% vulnerable file proportion reported in Section 3

Table 4: Discriminative Power results with example critical values

Metric	Linux Kernel			PHP			Wireshark		
	Example Critical Value	Precision	Recall	Example Critical Value	Precision	Recall	Example Critical Value	Precision	Recall
DNMaxEdgeBetweenness	350.0	20.2%	15.2%	10.0	26.4%	55.8%	2.5	15.6%	7.5%
NumDevs	9	44.4%	9.5%	9	42.3%	26.8%	6	13.7%	37.8%
NumCommits	33	36.0	8.4%	33	47.1%	37.2%	33	14.0%	19.7%
CNBetweenness	28,000	19.0%	11.8%	3,000	77.8%	16.3%	1,750	17.9%	7.5%

Table 5: Discriminative power for critical values based on fixed inspection rate (IR) of 10%

Metric	Linux Kernel			PHP			Wireshark		
	10% IR Crit. Value	Precision	Recall	10% IR Crit. Value	Precision	Recall	10% IR Crit. Value	Precision	Recall
DNMaxEdgeBetweenness	2.1	11.2%	30.8%	12	26.7%	46.5%	1.9	9.9%	35.8%
NumDevs	4	11.2%	53.0%	5	31.0%	63.4%	5	11.4%	57.6%
NumCommits	9	12.8%	54.8%	13	31.3%	60.5%	18	10.4%	38.5%
CNBetweenness	10,000	9.6%	38.8%	550	29.5%	53.5%	759	11.0%	43.3%

5.3 Predictability: How Many Vulnerable Files Are Explained?

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”) [23]. In our study, we use Bayesian networks as our predictive model. [9, 21]. Bayesian networks use Bayesian inference on a network of metrics, taking into account conditional dependencies between metrics. Bayesian networks also have widespread applications, including gene expression [16] and satellite failure monitoring systems [2].

Supervised models require a *training set* and a *validation set*. In this study, we use cross validation to generate each set. For our prediction, we created a Bayesian network model and tested it using ten-fold cross-validation. Ten-fold cross validation 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%
F ₂	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 in 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 significantly 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 was vulnerable because of 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.

5.4 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

	Test Fold (trained on other two)			Avg.
	Linux Kernel	PHP	Wireshark	
Precision	12.7%	34.3%	15.2%	20.73%
Recall	47.9%	53.5%	25.4%	42.27%
F ₂	30.8%	48.1%	22.4%	33.8%
Inspection Rate	9.5%	8.6%	4.4%	7.50%

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 one project 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.

6. LIMITATIONS

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 [10] 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.

7. RELATED WORK

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. Only one of the studies relate developer activity metrics to security.

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 RHEL4 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.

Meneely et al., [12] 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.

Bird et al. [1] uses a similar approach to ours with the purpose of examining social structures in open source projects. 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.

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.

Gonzales-Barahona and Lopez-Fernandez [6] were the first to propose the idea of creating developer networks as models of collaboration from source repositories. The authors' objective was to present the developer network and to differentiate and characterize projects.

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.

8. SUMMARY

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:

(a) source code files changed by multiple, otherwise-separated clusters of developers are more likely to be vulnerable than changed by a single cluster;

(b) files are likely to be vulnerable when changed by many developers who have made many changes to other files; and

(c) 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.

9. ACKNOWLEDGMENTS

We thank Mark Cox, director of the Red Hat Security Response team, for helping track down vulnerabilities in the kernel. We also thank the Realsearch group for their valuable feedback, especially Ben Smith's suggestions on the analysis portion. This work was supported by the U.S. Army Research Office (ARO) under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI). This work is also supported in part by the National Science Foundation Grant No. 0716176.

10. REFERENCES

- [1] C. Bird, D. Pattison, R. D'Souza *et al.*, "Latent Social Structures in Open Source Projects," in FSE, Atlanta, GA, 2008, p. p24-36.
- [2] S. Bottone, D. Lee, M. O'Sullivan *et al.*, "Failure prediction and diagnosis for satellite monitoring systems using Bayesian networks," in Military Communications Conference, 2008. MILCOM 2008. IEEE, 2008, p. 1-7.
- [3] U. Brandes, and T. Erlebach, *Network Analysis: Methodological Foundations*, Berlin: Springer, 2005.
- [4] F. Brooks, *The mythical man-month*. Addison-Wesley, 1995.
- [5] M. Girvan, and M. E. J. Newman, "Community Structure in Social and Biological Networks," The Proceedings of the National Academy of Sciences, vol. 99, no. 12, p. 7821-7826, 2001.
- [6] J. M. Gonzales-Barahona, L. Lopez-Fernandez, and G. Robles, "Applying Social Network Analysis to the Information in CVS Repositories," in 2005 Mining Software Repositories, Edinburgh, Scotland, United Kingdom, 2004, p.
- [7] J.-H. Hoepman, and B. Jacobs, "Increased security through open source," Commun. ACM, vol. 50, no. 1, p. 79-83, 2007.
- [8] ISO, *ISO/IEC DIS 14598-1 Information Technology - Software Product Evaluation*, 1996.
- [9] A. M. Martinez, and A. C. Kak, "PCA versus LDA," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 23, no. 2, p. 228-233, 2001.
- [10] McGraw, *Software Security: Building Security In*. Addison-Wesley, 2006.
- [11] A. Meneely, and L. Williams, "Secure Open Source Collaboration: An Empirical Study of Linus' Law " in Computer and Communications Security, Chicago, IL, 2009, p. 453-462.

- [12] A. Meneely, L. Williams, J. Osborne *et al.*, "Predicting Failures with Developer Networks and Social Network Analysis " in Foundations in Software Engineering, Atlanta, GA, 2008, p. 13-23.
- [13] N. Nagappan, and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density," in 27th International Conference on Software Engineering, St. Louis, MO, USA, 2005, p. 284-292.
- [14] N. Nagappan, B. Murphy, and V. R. Basili, "The Influence of Organizational Structure on Software Quality," in International Conference on Software Engineering, Leipzig, Germany, 2008, p. 521-530.
- [15] S. Neuhaus, T. Zimmermann, C. Holler *et al.*, "Predicting vulnerable software components," in Proceedings of the 14th ACM conference on Computer and communications security, Alexandria, Virginia, USA, 2007.
- [16] K. Numata, S. Imoto, and S. Miyano, "A Structure Learning Algorithm for Inference of Gene Networks from Microarray Gene Expression Data Using Bayesian Networks," in Bioinformatics and Bioengineering, 2007. BIBE 2007. Proceedings of the 7th IEEE International Conference on, 2007, p. 1280-1284.
- [17] M. Pinzger, N. Nagappan, and B. Murphy, "Can Developer-Module Networks Predict Failures?," in Foundations in Software Engineering, Atlanta, GA, 2008, p. 2-12.
- [18] E. S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, Sebastopol, California: O'Reilly and Associates, 1999.
- [19] N. F. Schneidewind, "Methodology For Validating Software Metrics," IEEE Transactions on Software Engineering, vol. 18, no. 5, p. 410-422, 1992.
- [20] Y. Shin, A. Meneely, L. Williams *et al.*, "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities," NCSU CSC Technical Report TR-2009-10, submitted to IEEE Transactions in Software Engineering, vol. no. p. 2009.
- [21] K. Tae-Kyun, and J. Kittler, "Locally linear discriminant analysis for multimodally distributed classes for face recognition with a single model image," Pattern Analysis and Machine Intelligence, IEEE Transactions on, vol. 27, no. 3, p. 318-327, 2005.
- [22] B. Witten, C. Landwehr, and M. Caloyannides, "Does Open Source Improve System Security?," IEEE Softw., vol. 18, no. 5, p. 57-61, 2001.
- [23] I. H. Witten, and E. Frank, *Data Mining: Practical machine learning tools and techniques*, 2 ed., San Francisco: Morgan Kaufmann, 2005.