Secure Open Source Collaboration: An Empirical Study of Linus' Law

Andrew Meneely and Laurie Williams
North Carolina State University
Department of Computer Science, Raleigh, North Carolina, USA
{apmeneel, lawilli3}@ncsu.edu

ABSTRACT

Open source software is often considered to be secure. One factor in this confidence in the security of open source software lies in leveraging large developer communities to find vulnerabilities in the code. Eric Raymond declares Linus' Law "Given enough eyeballs, all bugs are shallow." Does Linus' Law hold up ad infinitum? Or, can the multitude of developers become "too many cooks in the kitchen", causing the system's security to suffer as a result? In this study, we examine the security of an open source project in the context of developer collaboration. By analyzing version control logs, we quantified notions of Linus' Law as well as the "too many cooks in the kitchen" viewpoint into *developer activity metrics*. We performed an empirical case study by examining correlations between the known security vulnerabilities in the open source Red Hat Enterprise Linux 4 kernel and developer activity metrics. Files developed by otherwise-independent developer groups were more likely to have a vulnerability, supporting Linus' Law. However, files with changes from nine or more developers were 16 times more likely to have a vulnerability than files changed by fewer than nine developers, indicating that many developers changing code may have a detrimental effect on the system's security.

Categories and Subject Descriptors

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

General Terms: Measurement, Security, Human Factors

Keywords: Linus' Law, developer network, contribution network, vulnerability, metric

1. INTRODUCTION

Open source software is often considered to be secure [7, 23]. One factor in this confidence in the security of open source software lies in leveraging large developer communities to find vulnerabilities in the code. In his essay, *The Cathedral and the Bazaar* [19], Eric Raymond declares Linus' Law[1] as

> *"Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone*." [19]

Raymond states more colloquially, "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] in developer collaboration. Consider having many people make a meal: without enough coordination and communication, ingredients get skipped, added twice, or significant steps of the recipe are left out. The meal can suffer as a result of *too many* people. Likewise, perhaps the security of a software project can suffer as a result of unfocused contributions by too many developers.

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. Our research objective, then, is *to reduce security vulnerabilities by providing actionable insight into the structural nature of developer collaboration in open source software*.

We performed an empirical analysis by quantifying developer collaboration and unfocused contributions into *developer activity metrics*. We examine the statistical correlation between the known security vulnerabilities of the open source Red Hat Enterprise Linux 4 kernel and developer activity metrics. We used version control change logs to calculate four developer activity metrics. Forming social networks based on who worked on which file, we use network analysis to form metrics of developer groups and unfocused contributions.

The rest of this paper is organized as follows. Section 2 covers background. Sections 3 and 4 describe the case study, and derivation of the metrics themselves. Section 5 presents the results of the case study 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.

[1] In this context, the word "law" is used to mean a repeated observation [4].

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.

CCS'09, November 9-13, 2009, Chicago, Illinois, USA.
Copyright 2009 ACM 978-1-60558-352-5/09/11…$10.00.

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 [2] 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 [2]. 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 [2] 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.

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 area under the Receiver Operating Characteristic (ROC) curve.

- **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)*.
- **Area under the ROC Curve (AUC):** represents the proportion of the time that a classifier ranks a vulnerable file higher than a neutral file. AUC is calculated by integrating a ROC curve, usually by a summation approximation [24].

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. A classifier with an AUC of 75% means that, given one randomly-chosen neutral and vulnerable file, the classifier would choose the correct file 75% of the time.

3. CASE STUDY: LINUX KERNEL

We performed a case study on the Linux kernel[2] as it was distributed in the Red Hat Enterprise Linux 4 (RHEL4) operating system[3]. A summary of the RHEL4 kernel is found in Table 1. The entire project is over three million lines of C and assembly code. The security data is a labeling of whether or not a source code file was patched with a post-release vulnerability ("vulnerable" or "neutral"). The developer activity metrics were gathered from version control change logs.

Table 1: Summary of the RHEL4 Linux Kernel

Total number of files	14,286
Number of files changed (total studied)	10,454
Percentage of files changed	73%
Number of developers	557
Development time	15 months
Number of vulnerable files	205
Percentage of changed files with vulnerabilities	1.96%
Total number of commits	9,946

Gathering the security data involved tracing through the development artifacts related to each vulnerability reported in the Linux kernel. When members of the open source community become aware of a possible security vulnerability, members of the Red Hat Security Response (RHSR) team perform the following actions.

1. Create a defect report in the Red Hat Bugzilla database[4]. The majority of the subsequent artifacts can be found or linked to the new defect report.
2. Confirm the existence of the vulnerability in both the current build of the kernel (also called the **upstream** version), and the previous release of the kernel (also called a **backport**).
3. Form patches to fix the problem as necessary. Sometimes an upstream patch would differ from the backport patch since the kernel is always evolving.
4. Determine if the vulnerability is a **regression** (a vulnerability introduced by a patch after release).
5. Register the vulnerability in the National Vulnerability Database (NVD) and the next Red Hat Security Advisory (RHSA). The RHSR Team reports NVD and RHSA data on their security metrics website[5].

We collected our security data from the Bugzilla database, the NVD, and the RHSR security metrics database. Since each vulnerability was handled slightly differently, we examined each defect report manually to ensure that the backport patch was, in fact, needed. Since we are only interested in vulnerabilities that

[2] http://kernel.org/

[3] http://www.redhat.com/rhel/

[4] http://bugzilla.redhat.com/

[5] http://www.redhat.com/security/data/metrics/

existed at the time of release, we did not include regressions in our data set. For vulnerabilities that did not have all of the relevant artifacts (e.g. defect reports, backport patches), we consulted the director of the RHSR team to correct the data. Our data set is a comprehensive list of reported, non-regression vulnerabilities from RHEL4's release in February 2005 through July 2008. We found 205 files to be vulnerable (i.e. patched post-release because of at least one vulnerability), which was 1.96% of the 10,454 files we studied.

For the version control data from which developer activity metrics were computed, we used the Linux kernel source repository[6]. The RHEL4 operating system is based on 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 15 months of development and maintenance. We included in our analysis source code files that had the following file name extension: .c, .S, and .h. The version control data contains records of 557 developers and 9,946 commits over 10,454 source files. Most of the kernel files changed (73%) during those 15 months, including every vulnerable file. Our study focused on the files that were changed 15 months prior to release.

4. 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 4.1 and 4.2, respectively.

In Sections 4.1 and 4.2, we will use the following example. Suppose we are initially given the version control data in Table 2. In our example, we have six developers (Andy, Kelly, Phil, Lucas, Sarah, and Ben) and five files labeled A through E. The data in Table 2 denote who made changes to which file. More examples of the calculation of developer activity metrics can be found in related work [6, 11].

Table 2: Example version control data

Developer	Files Changed (# commits)
Andy	A(1), B (2)
Kelly	B(1)
Phil	B(2)
Lucas	A(1), C(2), D(1)
Sarah	D(2), E(1)
Ben	C(2), E(1)

A summary of the interpretation for each of the four metrics can be found in Table 3. We empirically evaluate these metrics as indicators of vulnerable files in Section 5.

4.1 Linus' Law: Diversity in Perspectives

In his essay on open source development [19], Eric Raymond describes several laws that explain how large open source projects have thrived. 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. Raymond contends that diversity in perspectives ought to be embraced, not avoided.

Table 3: 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

[6] http://git.kernel.org/

Thus, if parts of the project do not have a diverse perspective, perhaps vulnerabilities could arise.

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. Said another way, the NumDevs metric is the size of the developer group who contributed 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 over the entire project and how strongly connected these clusters are. Specifically, as developer clusters form, diversity in perspectives can be lost. 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*. We use the term developer network to be an estimation of the structure of collaboration in a software development project based on developer connections [1, 6, 13]. 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 (e.g. one month). 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. For our example in Table 2, the developer network is shown in Figure 1.

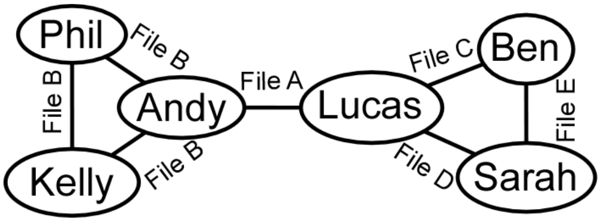

Figure 1: Resulting developer network from the Table 2

Second, we need to examine files between developer groups. In large software projects, groups of developers can form based on many factors, such as geographic location or feature of the product. A developer group need not be formally defined; a group can form out of a common need or affinity in the project. 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 outnumbers the number of inter-set connections [2]. A cluster of developers, then, has more connections within the cluster than to other developers. Having many clusters in a network can be an indicator that, while developers are collaborating within groups, the groups are not collaborating with each other. The files that are worked on by otherwise-separated clusters, therefore, may be more problematic. In this study, we are using a developer network cluster metric to measure the diversity in perspectives on a file.

Cluster metrics of developer networks can be used 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).

As an illustration, consider a network where nodes are houses and edges are streets. Large clusters of houses (e.g. cities) are generally connected by well-traveled (highly between) streets: highways. Conversely, streets that are within cities tend to be less traveled as there are many direct routes within a city. By identifying the highly between "highways" in a network, one can use the exits of the highways to infer the locations of the cities. Note that a city be composed of several "inner" clusters, (e.g. neighborhoods) and cities can have drastically different sizes. In fact, the notion of "clusteredness" is a varying concept depending on the situation, so defining the *exact* clusters of a network is a somewhat subjective exercise. However, regardless of how one defines exact clusters, edges of highest betweenness are always between clusters and are the focus of our metric.

In this study, we are interested in identifying the files that lie between clusters on the highly between edges of the developer network. 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.

In our example developer network in Figure 1, the edge of highest betweenness is the connection between Lucas and Andy (betweenness is nine). The Lucas-Andy edge, connects two clusters: Lucas/Sarah/Ben, and Andy/Kelly/Phil.

4.2 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 the notion of "too many cooks in the kitchen," 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.

Note that a high NumCommits on a file can be unavoidable: sometimes code inevitably needs to be changed to support new features, enhancement, and maintenance [9-11]. A similar argument could be made about NumDevs. However, if a file is suffering from an unfocused contribution, the change activity should be high, resulting in a high NumCommits and NumDevs. However, since NumCommits and NumDevs only represents *the number of* people and changes, not *who* is making changes of an unfocused contribution, we add a third, more specific metric to our study: CNBetweenness.

The CNBetweenness metric is calculated from a **contribution network** [17, 18]. A contribution network is an abstraction of version control logs represented by a network. 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.

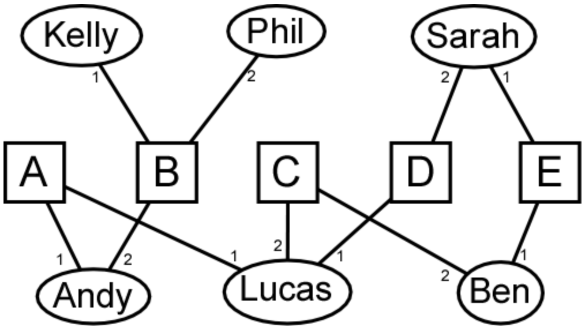

Figure 2: Contribution network from the example

The contribution network from the given example can be found in Figure 2.

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 3. 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`.

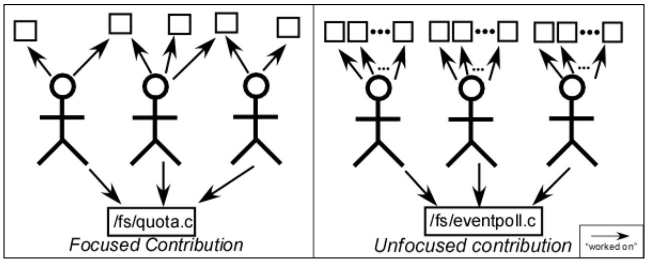

Figure 3: Examples of focused and unfocused contributions

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. For example, one could reduce the CNBetweenness of a file by assigning a task to a group of developers focused on a few files, rather than developers already working on other parts of the system.

5. EMPIRICAL ANALYSIS

Our empirical analysis of Linus' Law and unfocused contributions is a statistical correlation study between developer activity metrics and security vulnerabilities. We focus our empirical analysis on three questions in the following three 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?

Statistically speaking, the first question is an association question, the second is a discriminative power question, and the third is a predictive modeling question [20][7].

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 SAS[8] v9.1.3 for our statistical analysis and Weka[9] v3.6.0 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

[7] The framework from which these validity criteria are defined assumes a ratio metric scale, whereas our study is at a nominal scale. The exact statistical tests may differ but the spirit of the validation criteria remains intact.

[8] http://www.sas.com/

[9] http://www.cs.waikato.ac.nz/ml/weka/

metric. As suggested in other metrics validation studies [20], 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 our results for our four metrics in Table 3.

Table 4: Mann-Whitney-Wilcoxon test

Metric	Neutral Avg	Vulnerable Avg	MWW p-value
DNMaxEdgeBetweenness	18.8	136.6	p<0.0001
NumDevs	2.2	4.9	p<0.0001
NumCommits	4.1	13.7	p<0.0001
CNBetweenness	3662.8	12198.6	p<0.0001

In all four cases, 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 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 underwent many changes. This result is supports the "code churn" effect found in other studies [13, 14, 21] 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 [20] 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[10]. We use an analysis of critical values, and AUC for discriminative power.

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 depending on one's desired precision and recall (perhaps depending on the software development process), so we analyze all feasible critical points of our four metrics. Figure 4 contains plots of the critical values for each of the four metrics. The possible critical value of the metric is on the X-axis, and the two series are precision and recall as if one had used the metric as a discriminator past the critical values.

As an example of using critical values, consider gathering all files that had nine developers or more (NumDevs >= 9), then 33.3% (precision) of those files would be vulnerable, which is considerably high given that only 1.96% of the system's files were vulnerable[11]. Thus, using NumDevs provides 16 times (=33.3/1.96) more discriminative power than random selection. Furthermore, for files with fewer than nine developers, (NumDevs<9), 1.25% of the files were vulnerable. However, those 33.3% vulnerable files only account for 9.2% (recall) of the known vulnerable files in the system, meaning more metrics with high discriminative power are required. Table 5 shows some example critical values along with the precision, and recall.

Note that, in all four plots of Figure 4, when the recall becomes small, the precision has a greater variance. This effect is an artifact of the sample size decreasing as one uses a large, therefore more limiting, critical value.

Table 5: Discriminative power results

Metric	AUC*	Example Critical Value	P*	R*
DNMaxEdgeBetweenness	94.6%	270	7.6%	4.9%
NumDevs	85.7%	9	33.3%	9.2%
NumCommits	85.3%	25	26.7%	17.1%
CNBetweenness	78.6%	40,000	17.4%	7.3%

AUC: area under the ROC curve, P=precision, R=recall.

Another way to compare the metrics in terms of discrimination is with the AUC measurement. The AUC is calculated by finding the proportion of occurrences where a given metric for vulnerable files outrank a neutral file. Said another way, the AUC represents the probability that a metric's value for a randomly-chosen vulnerable file is higher than a randomly-chosen neutral file. The AUC measurement for each metric is given in Table 5.

Examining the results, one can see that different metrics have different advantages. DNMaxEdgeBetweenness has a low recall,

[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 1.96% vulnerable file proportion reported in Section 3

implying that DNMaxEdgeBetweenness accounts for relatively few vulnerabilities. NumDevs, NumCommits, and CNBetweenness all have high precisions when compared to the prior probability of 1.96%, but the recalls are still low. With the highest recall is NumCommits, meaning that examining files with 25 commits or more (in the 15 months of development), contain 17.1% of the known vulnerable files.

Furthermore, upon examining those files, about one in four would be vulnerable. 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. Our critical values are specific to the RHEL4 kernel during the time period we studied, so other projects may have different exact critical values.

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 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.

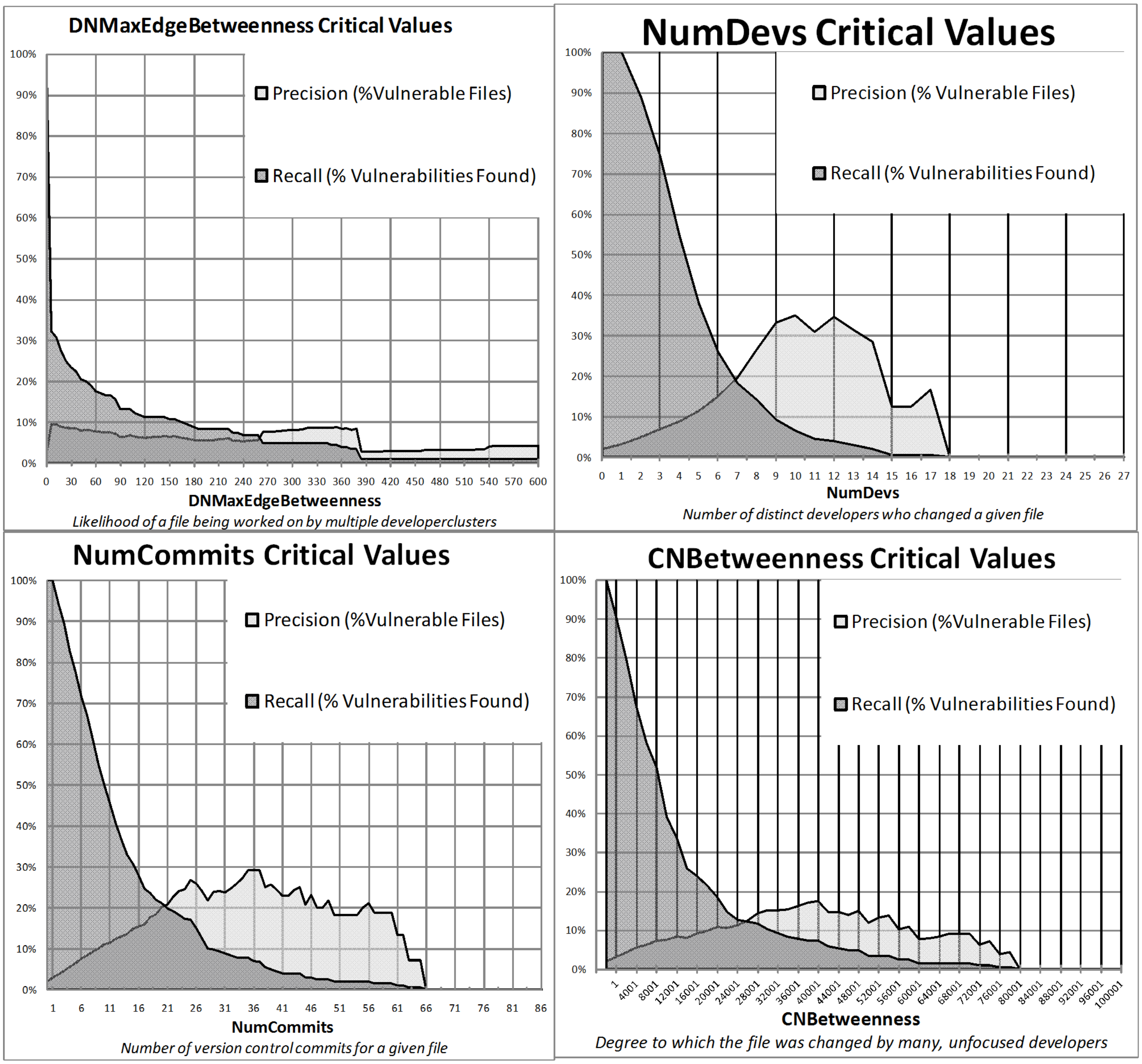

Figure 4: Critical values of metrics for the discriminative power criterion

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") [24]. In our study, we used two modeling methods: multivariate discriminant analysis and Bayesian networks. Discriminant analysis is a modeling method that uses an n-dimensional space to achieve maximum separation of variables. Discriminant analysis has widespread applications, including facial recognition [12, 22]. 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.

Supervised models require a *training set* and a *validation set.* In this study, we use cross validation to generate each set. For discriminant analysis, we used hold-one-out cross validation with recall, precision, and inspection rate as defined in Section 2.2. Hold-one-out cross-validation is performed by iteratively removing each data point from the set, training on all but the removed data point, then predicting for the removed point. For Bayesian Networks we used ten-fold cross-validation. Ten-fold cross validation is similar to hold-one-out, except the data is randomly partitioned into 10 partitions, with each partition being the held-out test set exactly once. Since ten-fold cross-validation is based on random partitions, we performed the cross-validation 15 times and report the average. The precision, recall, and inspection rate of the models can be found in Table 6.

Table 6: Predictability Results

Method	P*	R*	IR*
Multivariate Discriminant Analysis	9.9%	50.7%	10.0%
Bayesian Networks	13.3%	33.2%	4.9%

* P=precision, R=recall, IR=inspection rate,

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. If a team wanted to inspect files using the Bayesian network model, then they would only need to inspect 4.9% of the files and would find 33.3% of the vulnerable files.

The difference in modeling methods shows that Bayesian networks tend to be more precise, requiring a lower inspection rate, but find a smaller percentage of the known vulnerable files than multivariate discriminant analysis. One can use this fact for deciding which modeling method to use if the goal is to predict vulnerable files based on our four developer activity metrics.

Figure 5 shows the recall, inspection rate, and precision for all 15 runs of ten-fold cross-validation Bayesian network models. The figure shows minute variations from trial to trial, meaning that negligible variation in the model performance was due to the random partitioning in cross-validation.

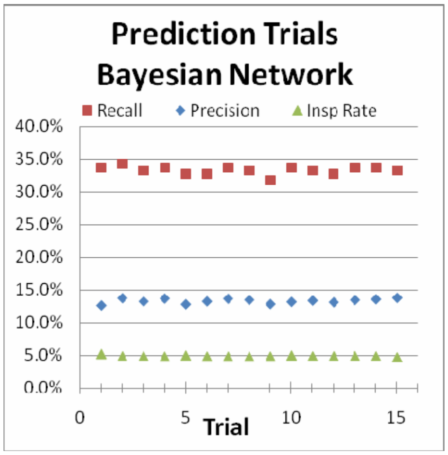

Figure 5: Variation of Bayesian Network in cross-validation

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 poor developer activity.

5.4 Discussion

Our results show a statistically significant correlation in all four developer activity metrics. Three of the four metrics support the notion of unfocused contributions, specifically that the vulnerable files were undergoing a lot of change (NumCommits), by many developers (NumDevs), and by developers who were also working on many other files (CNBetweenness).

When examining unfocused contributions, one might consider what Frederick Brooks says about too many developers on a project:

> *"If each part of the task must be separately coordinated with each other part, the effort increases as n(n-1)/2.*"[3]

In context, Brooks is specifically discussing Brook's Law, "Adding manpower to a late software project makes it later". However the reasoning bears resemblance to the unfocused contribution argument in that coordinating a large number of developers can require a communication and coordination effort quadratic in the number of developers. While unfocused contributions do not necessarily negate Linus' Law, our results show that they are a legitimate opposing force.

As far as diversity in perspectives, more developers changing the code does *not* mean an increase in security. One interesting observation is in the NumDevs chart in Figure 4: the proportion of vulnerabilities steadily increases as the number of developers increases (until about nine when the recall drops). However, files worked on by disparate, otherwise-separated clusters of developers are more likely to have a vulnerability.

One factor to consider when evaluating software security and Linus' Law is taking ongoing code change into account. Many open source projects are constantly changing and evolving (the Linux kernel is no exception). The developer community must continually keep up with finding vulnerabilities and fortification efforts in an ever-changing, ever-branching project.

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 our case study, all of the vulnerabilities happened to be in files that were changed 15 months prior to release (our time period under study). A vulnerability could be in a file that was not changed, but since our case study had no instance of that situation, this effect did not have an impact on our results.

Also, we cannot claim that developer activity metrics *cause* vulnerabilities. Studies of historical data can only show a statistically significant correlation. Proving causation would require a controlled experiment, which is not feasible for an ongoing project like the Linux kernel. Furthermore, although we used the term "validation criteria", we do not consider developer activity metrics to be fully validated until further case studies support consistent, repeatable results.

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.

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. [21] 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. The study examines a different collection of metrics than this study and combines disparate metrics into a single model.

Meneely et al., [13] 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 [3] 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.

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. [15] 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. Within our case study of the RHEL4 kernel, we found four metrics that empirically support the notions of 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; and

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

Practitioners can use these observations to prioritize security fortification efforts or to consider organizational changes among developers. While the results are statistically significant, the individual correlations indicate that developer activity metrics are likely to perform best for prediction in the presence of other metrics.

9. ACKNOWLEDGMENTS

We thank Mark Cox and the Realsearch group for their valuable support. 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).

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] U. Brandes, and T. Erlebach, *Network Analysis: Methodological Foundations*, Berlin: Springer, 2005.

[3] F. Brooks, *The mythical man-month*: Addison-Wesley, 1995.

[4] A. Endres, and D. Rombach, *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories*: Addison Wesley, 2003.

[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] M. M. Lehman, and L. Belady, *Program Evolution: Processes of Software Change*, London: Academic Press, 1985.

[10] M. M. Lehman, and J. F. Ramil, "Rules and Tools for Software Evolution Planning and Management," Annals of Software Engineering, vol. 11, no. 1, p. 15-44, 2001.

[11] M. M. Lehman, J. F. Ramil, P. D. Wernick *et al.*, "Metrics and Laws of Software Evolution -- The Nineties View," in 4th International Software Metrics Symposium (METRICS '97), Albuquerque, NM, 1997, p. 20-32.

[12] 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.

[13] 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. to appear.

[14] 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.

[15] 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.

[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., 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] M. Pinzger, N. Nagappan, and B. Murphy, "Can Developer-Module Networks Predict Failures?," in Foundations in Software Engineering, Atlanta, GA, 2008, p. to appear.

[19] 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.

[20] N. F. Schneidewind, "Methodology For Validating Software Metrics," IEEE Transactions on Software Engineering, vol. 18, no. 5, p. 410-422, 1992.

[21] 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 TSE.

[22] 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.

[23] B. Witten, C. Landwehr, and M. Caloyannides, "Does Open Source Improve System Security?," IEEE Softw., vol. 18, no. 5, p. 57-61, 2001.

[24] I. H. Witten, and E. Frank, *Data Mining: Practical machine learning tools and techniques*, 2 ed., San Francisco: Morgan Kaufmann, 2005.