Interactive Churn Metrics: Socio-Technical Variants of Code Churn

Andrew Meneely
Department of Software Engineering
Rochester Institute of Technology
134 Lomb Memorial Drive, GOL-1547
Rochester, NY 14623-5608
andy@se.rit.edu

Oluyinka Williams
Department of Software Engineering
Rochester Institute of Technology
134 Lomb Memorial Drive
Rochester, NY 14623-5608
oaw1625@rit.edu

ABSTRACT
A central part of software quality is finding bugs. One method of finding bugs is by measuring important aspects of the software product and the development process. In recent history, researchers have discovered evidence of a "code churn" effect whereby the degree to which a given source code file has changed over time is correlated with faults and vulnerabilities. Computing the code churn metric comes from counting source code differences in version control repositories. However, code churn does not take into account a critical factor of any software development team: the human factor, specifically who is making the changes. In this paper, we introduce a new class of human-centered metrics, "interactive churn metrics" as variants of code churn. Using the `git blame` tool, we identify the most recent developer who changed a given line of code in a file prior to a given revision. Then, for each line changed in a given revision, determined if the revision author was changing his or her own code ("self churn"), or the author was changing code last modified by somebody else ("interactive churn"). We derive and present several metrics from this concept. Finally, we conducted an empirical analysis of these metrics on the PHP programming language and its post-release vulnerabilities. We found that our interactive churn metrics are statistically correlated with post-release vulnerabilities and only weakly correlated with code churn metrics and source lines of code. The results indicate that interactive churn metrics are associated with software quality and are different from the code churn and source lines of code.

Categories and Subject Descriptors
D.2.8 [**Metrics**]: Process and Product Metrics

General Terms
Measurement, Security, Human Factors.

Keywords
code churn, socio-technical, interactive churn

1. INTRODUCTION
A central part of software quality is finding bugs. From developers holding code review meetings to testers trying to break the system, the entire software development team is constantly on the hunt for new bugs to fix. One method of finding bugs is by measuring important aspects of the software product and the development process. For example, software developers can use coupling metrics to identify potential design flaws, or software testers can use complexity metrics to prioritize their system tests.

Among the metrics studied the current field of software engineering research, the code churn metric has been consistently shown to be correlated with several indicators of overall software quality, such as faults [1–4] and vulnerabilities [5–7]. Code churn is a metric used to measure the degree to which a given source code file has changed over time. The idea is that files with high churn have a higher probability that a new bug was introduced recently. Code churn is measured by processing revisions in a version control system and counting the total number of lines changed.

One drawback of code churn is that it does not account for human factors, despite being a process metric. Developers asked to act on a code churn metric must differentiate between "good" changes and "bad" changes, without using any information about the most important part of software process: people. As a result, one could argue that code churn is not actionable, as sometimes code requires some change prior to release.

In an effort to make code churn more actionable, we apply a socio-technical (human-centered) perspective to how code churn is computed. To investigate the relationship between interactive churn metrics and software quality, we examined one indicator of overall software quality: post-release vulnerabilities. Thus, *the objective of this work is to improve software quality by providing actionable, human-centered process metrics that can be used for finding quality problems.* Specifically, we examine whether or not a developer was changing his or her own code, or another developer's code. We call the two categories of code churn: "self churn" and "interactive churn". We derive and present several metrics from this concept that can improve the utility of code churn for developers. Additionally, we conducted an empirical analysis of our interactive churn metrics on the PHP programming language and its post-release vulnerabilities.

This paper is organized as follows. Section 2 defines various terms used in this paper. Section 3 covers related work in the areas of code churn and empirical analysis of vulnerabilities. Section 4 covers the interactive churn metrics. Sections 5 and 6 cover our empirical case study of PHP. Section 7 covers limitations of both the metrics and the study. Finally, we summarize our study in section 8.

2. BACKGROUND
In this paper, we use several terms regarding version control systems and process metrics. The type of metrics we are proposing are **socio-technical** metrics, a term we borrow from sociology to describe the connection between two people in the context of work-related collaboration [8], [9], which is of a social and technical nature. The term "technical" is not referring to technology-related activities, but to the more general idea of technicality, skill, and labor. Thus, a socio-technical metric focuses on people and their interactions with others.

Regarding version control systems, we use the term **revision** to describe a recorded, individual change to the source code as recorded by the version control system. Some systems call these "commits", or "change sets". Each revision contains a **diff**, or compared difference between a source code file prior to the revision and after the revision. A diff is often computed by a built-in tool, such as GNU Diff[1]. Each revision also contains an **author**, which is the name and email of the person who made the revision. Finally, most modern version control

[1] http://www.gnu.org/software/diffutils/

DOI: 10.1145/2382756.2382785 http://doi.acm.org/10.1145/2382756.2382785

systems have a **blame** tool (e.g. `git blame`, `svn blame`, `cvs annotate`). The blame tool is a command that traverses the history of a given source code file, aggregating all of the changes, to show the most recent author and revision for each line in the file.

3. RELATED WORK

Analyzing version control repositories for code churn is a common occurrence in today's software engineering research. Code churn has been used in several prediction models, and often appears among the stronger correlations with faults and vulnerabilities. In this section we discuss several studies that employ code churn, in both the reliability and security realms.

Munson and Elbaum [10] were among the first to explore the concept of code churn. Analyzing code churn in conjunction with several complexity and size metrics, the authors combined their metrics together to predict faults in a large embedded software system. Regarding code churn by itself, they found a strong correlation (Pearson $r=0.728$, $p<0.05$) between the code churn of modules and the number of faults in those modules. Compared to this paper, their analysis was focused on the number of faults as opposed to binary classification of vulnerable or neutral files (see Section 6).

Nagappan et al. [1], [2] has provided several empirical analyses of code churn on case studies such as Microsoft Windows. In one study, Nagappan et al. [1] proposed that researchers normalize their code churn measure by the number of source lines of code in of the file. In that study, they found that normalized, or "relative" code churn measures were better predictors of product quality metrics. In our empirical study, we analyzed the relative form of code churn. Also, our interactive churn metrics are normalized by lines of code where appropriate.

Casebolt et al. [11] examined the concept of "author entropy" regarding developer collaboration. Author entropy is an analysis of author contribution to a given file. From their research, files with low author entropy have a dominant author and files with high entropy do not have dominant authors. The author entropy metric provides a snapshot contribution summary but does not provide details of the relationship between the developers in a given file. The research only identifies the spread of contribution based on number of commits. Our research provides a finer-grained analysis of authors changing lines of code at the line level.

Gegick et al. [12] modeled vulnerabilities with various metrics, such as source lines of code, alert density from a statistic analysis tool, and code churn. The authors performed a case study on 25 components in a commercial telecommunications software system with 1.2 million lines of code. Their model identified 100% of the vulnerable components with an 8% false positive rate at best.

In previous studies [5–7], we investigated code churn along with complexity and developer activity metrics as indicators of security vulnerabilities. We studied the likelihood of a file being vulnerable by collecting the metrics before a release and predicting them against the reported vulnerabilities post-release. We also developed some socio-technical metrics for the study, such as metrics dealing with contribution to a file and the socio-technical developer network surrounding a file. We conducted studies on the Red Hat Enterprise Linux kernel and Mozilla Firefox. The combined model of code churn, complexity, and developer activity metrics resulted in predicting 80% of known vulnerable files and less than 25% false positives.

4. INTERACTIVE CHURN

In this section, we introduce and define our interactive churn metrics as socio-technical variants of code churn. First, we will demonstrate how code churn is typically computed and then show how interactive churn is computed.

4.1 Code Churn

The code churn metric is typically computed for a given revision of a file. The version control system computes a diff, which is a matching of which lines of code were inserted or deleted in the given source code file. At each line, the version control system provides the lines that were deleted or inserted. Code churn is defined as the number of line insertions plus the number of line deletions.

As an example of computing code churn, Figure 1 (see next page) shows an example diff for a single revision in the PHP programming language[2], using the version control system Git[3]. The revision involved three line deletions and three line insertions, so the code churn in this example is six.

One concern [1] regarding code churn by itself is its relation to the overall size of the file. Code churn of 100 has a much different meaning for a file of 200 lines than 20,000 lines. Thus, in our analysis, we normalize the code churn of the file by the total number of source lines of code (SLOC) in the file after its revision. For example, a file with 100 lines of churn and ended with 200 SLOC would have a churn of 50%. Note that if a file was fully rewritten, it can have a code churn exceeding 100%.

4.2 Percentage of Interactive Churn & Authors Affected

The idea behind interactive churn is to examine *who* is making the changes and *whose* code is being changed. Developers may be revising their own code, or changing someone else's code. While developers may not have records of explicit code ownership practices, the version control system can provide a listing of who was the last person to modify a given line of code via a built-in blame tool. Each line that commit affects was last modified either by him- or herself, or by another developer. This line-level analysis provides a fine-grained record of developers interacting (knowingly or unknowingly) via specific lines of source code.

To compute *interactive churn* for a given revision and a given file, we do the following:

1. **Process the revision diff** to identify the author and the lines of code deleted in the given revision.

2. **Run the blame tool** on the file to identify the authors of the lines deleted, setting the revision history boundary to the given revision.

3. **Look up the lines affected** by the diff in the blame output, counting the number of lines and different authors affected by the deletions.

For example, Figure 1 shows that the author of the commit was Antony Dovgal and that the three line deletions occurred at lines 1644, 1655, and 1661 (from the lines starting with `@@`). To compute interactive churn, we need to know if the three deleted lines were last modified by Antony, or by someone else. Thus, we use the `git blame` tool, which traverses the version history of a given file and determines the most recent developer who changed a given line of code up to a given revision boundary. Figure 2 shows the output of the blame command, filtered by the three lines in question. The output shows that all three of the lines that Antony deleted were lines previously modified by two other developers, Rasmus and Ilia. Thus, we say that the number of interactive churn lines in this commit is three.

From this computation, we define two interactive churn metrics to supplement code churn:

- **Percentage of Interactive Churn (PIC)** is the ratio of the interactive churn lines to the total number of lines of code deleted

[2] http://www.php.net/
[3] http://git-scm.com/

DOI: 10.1145/2382756.2382785 http://doi.acm.org/10.1145/2382756.2382785

```
$ git log -1 -p -U0 7633511047
commit 76335110478f6730230a3bccf6e09cbe2446adcb
Author: Antony Dovgal <tony2001@php.net>
Date:   Sun May 4 11:26:50 2008 +0000

    fix compile warnings

diff --git a/ext/standard/array.c b/ext/standard/array.c
index 6f23da0..c66fb16 100644
--- a/ext/standard/array.c
+++ b/ext/standard/array.c
@@ -1644 +1644 @@
-               add_next_index_stringl(return_value, low, 1, 1);
+               add_next_index_stringl(return_value, (const char *)low, 1, 1);
@@ -1655 +1655 @@
-               add_next_index_stringl(return_value, low, 1, 1);
+               add_next_index_stringl(return_value, (const char *)low, 1, 1);
@@ -1661 +1661 @@
-               add_next_index_stringl(return_value, low, 1, 1);
+               add_next_index_stringl(return_value, (const char *)low, 1, 1);
```

Figure 1. A revision and diff generated from Git

```
$ git blame 7633511047^ -- ext/standard/array.c | grep -e "1644)" -e "1655)" -e "1661)"
45d71e29 (Rasmus Lerdorf 2001-07-09 20:36:47   1644) add_next_index_stringl(return_value, low, 1, 1);
45d71e29 (Rasmus Lerdorf 2001-07-09 20:36:47   1655) add_next_index_stringl(return_value, low, 1, 1);
3a4bf3f9 (Ilia Alshanetsky 2002-12-20 17:16:31 1661) add_next_index_stringl(return_value, low, 1, 1);
```

Figure 2. Filtered results from `git blame` in computing interactive churn

in the revision. If no lines of code were deleted (e.g. only insertions), we define the PIC as zero;

- **Number of Authors Affected (NAA)** is the number of authors (besides the revision author) whose lines were changed by the given revision.

In following with our example from Figures 1 and 2, the PIC of the revision is 100% (all of the deleted lines were last modified by authors other than Antony), and the NAA is two (for Rasmus and Ilia).

The above two metrics are defined at the *revision* level, but they can also be defined at the *file* level. A source code file with one or more revisions can have its interactive churn metrics computed by aggregating the measurements, defined as:

- **Percentage of Interactive Churn for a File (PIC-F)** is the ratio of the interactive churn lines for all revisions of a file to the total number of lines of code deleted for all revisions of a file. If no lines of code were deleted (e.g. only insertions) PIC-F is zero;
- **Average Number of Authors Affected for a File (AvgNAA-F)** is the average of NAA over all of the revisions for a given file.

We note that computing interactive churn metrics means looking up past authors on lines that were deleted. For lines that are inserted, these lines would be counted in future commits if the code is ever touched by a future commit. Thus, not counting insertions is not a limitation, but avoiding double-counting.

In practice, we recommend using PIC and NAA as a supplement to code churn because together they provide a more complete picture of how the code changed. A high code churn with a low PIC and low NAA indicates that, while the code was changed significantly, it was changed mostly by one person working on his or her own code. However, a high code churn with a high PIC but low NAA indicates that a small group of people were working on the code. Finally, a high code churn with a high PIC and high NAA indicates that a large group of people were all modifying the same lines of code. Each of these situations may be desirable or undesirable, depending on the context of the team and the product.

5. CASE STUDY: PHP

For this study, we analyzed the version control repository and vulnerability history of the PHP programming language. We computed the code churn and interactive churn metrics on all of the source code files with names ending in ".c" or ".h" for PHP version 5.3. The code churn and interactive churn we computed were based on the main development from the major release of PHP 5.0 through the release of PHP 5.3, a time span from the year 2004 through 2009. In all, PHP 5.3 has 1,183 ".c" or ".h" files with any code churn during that time. Those files together have 673,905 source lines of code (SLOC), as measured by the CLOC[4] tool.

The PHP team uses Git as their primary version control system. Git differentiates the concept of "authors" from "committers", since one person can construct the wchange and another enact it. For the purposes of this study, we are analyzing the authors, not the committers, because we are interested in the person who is attributed to writing the code.

As one indicator of overall software quality, we examined the post-release vulnerabilities for PHP 5.3. These vulnerabilities are publicly-disclosed, already-fixed security-related faults reported by the PHP team. We manually traced each reported vulnerability back to the fix revision in their version control system. Each source code file that was fixed for a post-release vulnerability in PHP 5.3 was then labeled **vulnerable**. Files not associated with a vulnerability were labeled **neutral**, since no vulnerabilities had been found as of yet. In all, PHP 5.3 had 67 post-release vulnerabilities, which traced to 92 different fix revisions, leading to 89 different ".c" or ".h" vulnerable files.

6. EMPIRICAL ANALYSIS

Our empirical analysis is a statistical correlation study between interactive churn metrics, code churn, and security vulnerabilities. Since vulnerabilities are collected at the file level, we analyze our file-level interactive churn metrics: PIC-F and AvgNAA-F. Additionally, we evaluate the normalized version of code churn and SLOC.

[4] http://sourceforge.net/projects/cloc/

DOI: 10.1145/2382756.2382785 http://doi.acm.org/10.1145/2382756.2382785

We focus our empirical analysis on three specific questions, the first and third are taken from Schneidewind's validation criteria [13], covered in the following three subsections:

Association: Are interactive churn metrics associated with vulnerable files? (Section 6.1)

Co-Linearity: Are interactive churn metrics strongly correlated with code churn and SLOC, or with each other? (Section 6.2)

Predictability: How many of the vulnerable files can be explained by the metrics together? (Section 6.3)

We used the R statistics package for our statistical analysis and Weka v3.6.7 [14] for our Bayesian network prediction model.

6.1 Association: Are The Metrics Correlated With Vulnerable Files?

To examine how interactive churn metrics 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 [13] for not having a normality assumption, we use the non-parametric **Mann-Whitney-Wilcoxon (MWW)** test and compare the means. 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 1. Note that we report code churn as normalized by SLOC, hence the unit being a percentage.

Table 1. Association results

Metric	Mean		MWW p-value
	Vulnerable	Neutral	
SLOC	1,703.6	477.4	$p<0.0001$
Code churn	160.4%	120.0%	$p<0.01$
PIC-F	3.0%	8.4%	$p<0.0001$
AvgNAA-F	1.30	0.92	$p<0.0001$

Based on these results, we conclude that all four metrics are statistically associated with vulnerable files. The statistically-significant results of SLOC and code churn are consistent with previous studies [1], [6], [10] in that larger source code files and more churn are associated with being vulnerable files.

Interestingly, the association direction of PIC-F appears to favor neutral files having *more* interactive churn than vulnerable files. This indicates that neutral files are changed by different people changing the same lines of code. However, AvgNAA-F was higher for vulnerable files than neutral files, and around affecting one other person. Together, these two results indicate that neutral files tend to have smaller groups of people working on each others' lines of code.

6.2 Co-Linearity: Are the Metrics Strongly Correlated with Each Other?

While interactive churn metrics are measuring more socio-technical properties of source code files than SLOC and code churn, we still need to know how closely related all of the metrics are to each other. In particular, we want know if interactive churn metrics are not just surrogate measures of SLOC and code churn, in which case developers may not need the extra metric. For example, if files with high code churn always have a high number of AvgNAA-F, then developers should just use code churn as they did before. Thus, in this co-linearity analysis, a statistically-significant weak correlation would provide evidence that interaction churn metrics are different from code churn and SLOC.

To evaluate the inter-relationships between the metrics, we use the non-parametric Spearman rank correlation coefficient [15] for its lack of assumptions about normality. Each correlation coefficient is a number between negative one and positive one indicating the strength and direction of the correlation, as well as a p-value for statistical significance. Generally speaking, correlation coefficients are considered strong at 0.7 and above because that indicates 50% of the variance is explained [15]. A statistically insignificant result is inconclusive and is not considered the same as a weak correlation. Our resulting Spearman coefficients can be found in Table 2.

Table 2. Spearman rank correlation coefficients

	SLOC	Code Churn	PIC-F
SLOC	1.00		
Code churn	-0.30	1.00	
PIC-F	-0.26	0.06*	1.00
AvgNAA-F	0.17	0.31	0.40

All p-values are $p< 0.0001$, except * where $p=0.035$

While all of the correlations are statistically significant at a 0.05 level, none of the correlations are strong. Thus, interactive churn metrics are not closely related to SLOC or code churn, and not strongly related to each other.

A few interesting additional observations can be made on these results as well. SLOC and code churn are not strongly correlated with each other, indicating that many large files had little churn and/or many small files had large churn. This result is consistent with previous research on those two metrics [6], [10].

Interestingly, SLOC and PIC-F were inversely correlated with each other, meaning that large files had low percentages of interactive churn. This indicates that when files undergo many changes, they generally are developers modifying their own lines of code. This also is consistent with the association result in Section 6.1 that vulnerable files have higher SLOC but lower PIC-F, on average.

Additionally, the correlation between PIC-F and code churn was very weak, indicating that files can have high or low churn, with little bearing on the amount of interactive churn. Finally, AvgNAA-F was moderately correlated with both code churn and PIC-F, which is unsurprising given that more churn and more interactive churn increases the chances of affecting more authors.

6.3 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 (e.g. "neutral" or "vulnerable"). In our study, we used Bayesian networks as our predictive model. Bayesian networks use Bayesian inference on a network of metrics, taking into account conditional dependencies between metrics [14]. Bayesian networks also have widespread applications, including gene expression [16] and satellite failure monitoring systems [17].

Supervised models require a *training set* and a *validation set.* In this study, we use ten-fold cross validation to generate each set. Ten-fold cross validation is performed by randomly partitioning the data into 10

DOI: 10.1145/2382756.2382785 http://doi.acm.org/10.1145/2382756.2382785

folds, with each fold being the held-out test fold exactly once [14]. In this study, we repeated the 10-fold cross-validation 10 times with different randomization seeds and averaged the results.

In this study, we used two different kinds of Bayesian network models: the canonical Bayesian network and a variant called a Complement Class Naïve Bayes classifier [18] (called BayesNet and ComplementNaiveBayes in Weka, respectively). The latter model tends to favor low rates of false positives, while the former tends to favor low rates of false negatives.

We evaluate our models with two measures: precision and recall. Since our vulnerability 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). The definitions of precision and recall are as follows:

- **Precision (P)** is defined as the proportion of correctly predicted vulnerable files: $P=TP/(TP+FP)$
- **Recall (R)** is defined as the proportion of all known vulnerabilities found: $R=TP/(TP+FN)$.

In constructing our models, we used two combinations of the metrics to compare the predictability: all four metrics together and our base of metrics SLOC and code churn alone. By comparing the precision and recall of the two types of models, we can examine how much improvement in vulnerability prediction that interactive churn metrics provide. The precision and recall of the four models can be found in Table 3.

Table 3. Prediction results

		Precision	Recall
SLOC, Code Churn, PIC-F, AvgNAA-F	**Bayes Network**	43.6%	26.3%
	Compl. Bayes	18.0%	55.1%
SLOC, Code Churn only	**Bayes Network**	0%	0%
	Compl. Bayes	14.0%	64.4%

The resulting models had moderate performance given the fact that we were only evaluating four metrics and only looking at code churn, size, and interactive churn. We would not expect that code churn and interactive churn to fully explain all of the vulnerabilities as vulnerabilities can arise for many different reasons.

Additionally, we note that in the Bayes model, the prediction completely fails without the presence of the two interactive churn metrics. This indicates that the interactive churn metrics are good supplements to code churn and SLOC in terms of predictability of vulnerabilities.

Within these results, we also note that the two types of models traded off in precision and recall. Developers or researchers thinking of using these types of models may want to consider whether precision or recall is more important, depending on how the metrics are going to be used.

7. LIMITATIONS

Regarding our interactive churn metrics, one limitation of the metric is that if a file has not been changed, no measurements can be made. This is also true of code churn, and the studies of code churn discussed in Section 3 are evidence that this is not a particularly bad limitation. In our case study, none of the known vulnerabilities for PHP 5.3 existed in files that had no code churn, so our data set did not miss any vulnerabilities.

Another minor limitation of the interaction churn metrics is the potential for accidentally classifying an author as "other" when in fact a developer changed name or email address. Modern version control systems such as Git (the system used in this study) have a disambiguation feature to help map old emails to new emails. Furthermore, to mitigate this we conducted a manual inspection of each author in version control logs to ensure that this was not a problem.

Furthermore, while interactive churn shows promise in the area of security, we do not make claims about faults or other elements of software quality. However, Gegick et al. [19] have shown that a significant, but not surrogate, relationship exists between vulnerabilities and non-security-related faults. Thus, the topic of faults and interactive churn is worth pursuing in future studies.

Finally, this paper only covers only one case study of using interaction churn metrics. We chose PHP as a case study because we believe it is representative of many software development projects. However, more case studies on interaction churn are certainly welcome in this area.

8. SUMMARY & FUTURE WORK

In an effort to make code churn more actionable, we take a socio-technical approach to code churn, a classic code churn metric. We introduced the concept of interactive churn, which is defined that the number of lines in a given revision where the author was revising code last modified by somebody else. From this concept we derived two revision-level metrics: PIC and NAA, along with their file-level counterparts, PIC-F and AvgNAA-F. To investigate the relationship between interactive churn metrics and software quality, we examined one indicator of overall software quality: post-release vulnerabilities. We conducted an empirical analysis of PHP 5.3, including a comparison to code churn and SLOC. Our analysis of the association, co-linearity, and predictability of interaction churn metrics indicates that they are statistically correlated with post-release vulnerabilities. Software developers can use these metrics and other derivative metrics to examine how much they are (knowingly or unknowingly) interacting with their teammates at the fine-grained level of a single line of code. Future studies will expand upon this by (a) investigating correlations in other case studies for replication, (b) investigating correlations to other quality indicators (e.g. faults), and (c) deriving more metrics from the interactive churn concept, such incorporating as the element of time into interactive churn.

ACKNOWLEDGEMENTS

We thank the RIT Department of Software Engineering for funding this research. We also thank Harshavardhan Srinivasan for his valuable feedback throughout the whole project.

REFERENCES

[1] N. Nagappan and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density," in *27th international Conference on Software Engineering (ICSE)*, St. Louis, MO, USA, 2005, pp. 284–292.

[2] N. Nagappan, "Toward a Software Testing and Reliability Early Warning Metric Suite," in *Proceedings of the 26th International Conference on Software Engineering*, 2004, pp. 60–62.

[3] J. P. Hudepohl, S. J. Aud, T. M. Khoshgoftaar, E. B. Allen, and J. Mayrand, "Emerald: Software Metrics and Models on the Desktop," *IEEE Softw.*, vol. 13, pp. 56–60, Sep. 1996.

[4] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting Failures with Developer Networks and Social Network Analysis," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, Atlanta, Georgia, 2008, pp. 13–23.

[5] A. Meneely and L. Williams, "Strengthening the Empirical Analysis of the Relationship Between Linus' Law and Software

Security," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, Bolzano-Bozen, Italy, 2010, pp. 1–10.

[6] Y. Shin, A. Meneely, L. Williams, and J. Osborne, "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities," *TSE*, vol. 37, no. 6, pp. 772–787, 2011.

[7] A. Meneely and L. Williams, "Secure Open Source Collaboration: an Empirical Study of Linus' Law," in *Int'l Conference on Computer and Communications Security (CCS)*, Chicago, Illinois, USA, 2009, pp. 453–462.

[8] E. L. Trist and K. W. Bamforth, "Some social and psychological consequences of the longwall method of coal-getting," *Technology, Organizations and Innovation: The early debates*, p. 79, 2000.

[9] T. G. Cummings, "Self-Regulating Work Groups: A Socio-Technical Synthesis," *The Academy of Management Review*, vol. 3, no. 3, pp. 625–634, Jul. 1978.

[10] J. C. Munson and S. G. Elbaum, "Code churn: a measure for estimating the impact of code change," in *Software Maintenance, 1998. Proceedings. International Conference on*, 1998, pp. 24 – 31.

[11] J. R. Casebolt, J. L. Krein, A. C. MacLean, C. D. Knutson, and D. P. Delorey, "Author entropy vs. file size in the gnome suite of applications," in *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, 2009, pp. 91 –94.

[12] M. Gegick, L. Williams, J. Osborne, and M. Vouk, "Prioritizing software security fortification through code-level metrics," in *Proceedings of the 4th ACM workshop on Quality of protection*, New York, NY, USA, 2008, pp. 31–38.

[13] N. F. Schneidewind, "Methodology for Validating Software Metrics," *IEEE Transactions on Software Engineering (TSE).*, vol. 18, no. 5, pp. 410–422, 1992.

[14] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, 1st ed. Morgan Kaufmann, 1999.

[15] P. V. Rao, *Statistical Research Methods in the Life Sciences*, 1st ed. Duxbury Press, 1997.

[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, pp. 1280 –1284.

[17] S. Bottone, D. Lee, M. O'Sullivan, and M. Spivack, "Failure prediction and diagnosis for satellite monitoring systems using Bayesian networks," in *Military Communications Conference, 2008. MILCOM 2008. IEEE*, 2008, pp. 1 –7.

[18] J. D. Rennie, L. Shih, J. Teevan, and D. Karger, "Tackling the poor assumptions of naive bayes text classifiers," in *MACHINE LEARNING-INTERNATIONAL WORKSHOP THEN CONFERENCE-*, 2003, vol. 20, p. 616.

[19] M. Gegick, P. Rotella, and L. Williams, "Toward Non-security Failures as a Predictor of Security Faults and Failures," *International Symposium on Engineering Secure Software and Systems*, pp. 135–149, 2009.

DOI: 10.1145/2382756.2382785 http://doi.acm.org/10.1145/2382756.2382785