Data-driven Insights from Vulnerability Discovery Metrics

Nuthan Munaiah
Department of Software Engineering
Rochester Institute of Technology
Rochester, NY 14623
Email: nm6061@rit.edu

Andrew Meneely
Department of Software Engineering
Rochester Institute of Technology
Rochester, NY 14623
Email: axmvse@rit.edu

***Abstract*—Software metrics help developers discover and fix mistakes. However, despite promising empirical evidence, vulnerability discovery metrics are seldom relied upon in practice. In prior research, the effectiveness of these metrics has typically been expressed using precision and recall of a prediction model that uses the metrics as explanatory variables. These prediction models, being black boxes, may not be perceived as useful by developers. However, by systematically interpreting the models and metrics, we can provide developers with nuanced insights about factors that have led to security mistakes in the past. In this paper, we present a preliminary approach to using vulnerability discovery metrics to provide insightful feedback to developers as they engineer software. We collected ten metrics (churn, collaboration centrality, complexity, contribution centrality, nesting, known offender, source lines of code, # inputs, # outputs, and # paths) from six open-source projects. We assessed the generalizability of the metrics across two contextual dimensions (application domain and programming language) and between projects within a domain, computed thresholds for the metrics using an unsupervised approach from literature, and assessed the ability of these unsupervised thresholds to classify risk from historical vulnerabilities in the Chromium project. The observations from this study feeds into our ongoing research to automatically aggregate insights from the various analyses to generate natural language feedback on security. We hope that our approach to generate automated feedback will accelerate the adoption of research in vulnerability discovery metrics.**

***Index Terms*—metric, threshold, security, vulnerability, interpretation**

I. Introduction

Vulnerability discovery metrics, having been empirically-validated to be associated with historical vulnerabilities, have incredible potential to provide insights into the engineering failures that may have led to the introduction of vulnerabilities. We have well over 300 metrics in the literature that can be used to quantify security attributes of software [1] of which at least one hundred metrics, by our preliminary review, have been proposed to assist in vulnerability discovery. Despite empirical evidence of the association between metrics and vulnerabilities, their adoption in practice has been limited. An attribute often cited as inhibiting metrics' adoption is their ineffectiveness as explanatory variables in a vulnerability prediction model [2].

In prior vulnerability discovery metrics literature, there seems to be an overwhelming emphasis on optimizing precision and recall of vulnerability prediction models at the expense of interpretability, usability, and actionability of the insights from such models. However, by using the performance of a vulnerability prediction model to infer the utility of metrics, we are ignoring the metrics' ability to tell a story, as Fenton and Neil suggested in their software metrics roadmap almost twenty years ago [3]. Even if we had a vulnerability prediction model with a precision and recall of 90% in one project, we may not be able to apply it to predict vulnerabilities in a different project [4]. The insights from such a model and its metrics, however, may be transferable as being indicators of engineering failures that may have led to vulnerabilities in the past. For instance, consider a metric that identifies if a file is on the approximated attack surface [5] or one that quantifies the proximity of a function to the attack surface [6]. These metrics may not yield a marked improvement in precision and recall but provide valuable insights to contextualize the change that developers make to the software.

Engineering secure software is a nuanced endeavor requiring developers to think, oftentimes speculate, about obscure execution scenarios that may compromise users. In such scenarios, developers can benefit from being aware of the engineering failures that have potentially led to the introduction of vulnerabilities in the past.

Our research vision is to assist developers in engineering secure software by providing a technique that generates scientific, interpretable, and actionable feedback on security as the software evolves [7]. Our goal in this paper is to propose an approach to generate natural language feedback on security through the interpretation of vulnerability discovery metrics. In using data- and metrics-driven insights to provide feedback to developers, particularly in a context where developers can have a *conversation* about security (like in a code review), we hope to raise the awareness of developers to the (global) impact their changes may have on attributes associated with historical vulnerabilities.

We address the following research questions:

RQ 1 Generalizability Are vulnerability discovery metrics similarly distributed across projects?

RQ 2 Thresholds Are thresholds of vulnerability discovery metrics effective at classifying risk from vulnerabilities?

II. Metrics

We collected ten metrics shown to be associated with historical vulnerabilities. The choice of the metrics was informed by the pilot phase of a systematic literature review [8] on vulnerability discovery metrics being conducted in parallel.

The ten metrics considered in this study are:

1. *Churn* - The total number of lines added, modified, and deleted throughout the history of a file [9]. We collected the metric at the commit level and aggregated it at the file level by computing the sum of the metric.
2. *Collaboration (Centrality)* - The maximum of the edge centrality of edges representing files in a collaboration network [10]. A collaboration network is an unweighted and undirected graph in which nodes represent developers and edges represent files. An edge exists between two developers if they both changed at least one file.
3. *Contribution (Centrality)* - The node betweenness centrality of nodes representing files in a contribution network [10]. A contribution network is a weighted and undirected bipartite graph with two sets of nodes: files and developers. An edge exists between a developer node and a file node if the developer made a change (commit) to the file. The weight of the edge is the number of changes a single developer made to a particular file.
4. *(Cyclomatic) Complexity* - The number of unique decision paths through a function [11]. We collected this metric at the function level and aggregated it at the file level by computing the sum of the metric.
5. *(Known) Offender* - A binary-valued metric that indicates if a file has been fixed for a vulnerability in the past [12].
6. *Nesting* - The maximum nesting level of control structures in a function [11].
7. *Source Lines of Code* - The total number of lines of source code in a file [9].
8. *# Inputs* - The number of inputs that a function uses [11]. We collected this metric at the function level and aggregated it at the file level by computing the sum of the metric.
9. *# Outputs* - The number of functions that a given function calls [11]. We collected this metric at the function level and aggregated it at the file level by computing the sum of the metric.
10. *# Paths* - The number of unique decision paths through a function [11]. We collected this metric at the function level and aggregated it at the file level by computing the sum of the metric.

While we implemented the algorithms to collect the contribution and collaboration centrality metrics, we used `git` to collect the churn metric, a manual process to collect the offender metric, and SciTools Understand to collect the remaining metrics. Since we aim to collect, and analyze, these metrics from a sizable number of projects, we had to make some simplifying changes to the way the metrics are implemented. For instance, in collecting the collaboration and contribution metrics, Meneely *et al.* [10] restricted the time period of changes considered to 15 months. In our implementation of the metrics, we do not apply any such restrictions.

We implemented the metrics as self-contained microservices which will be released as containers as soon as our systematic literature review mentioned earlier is concluded.

III. Methodology

In this section, we describe the methodology used to collect and analyze the data to address our research questions.

A. Data Collection

We collected nine of the ten vulnerability discovery metrics mentioned in Section II from six open-source projects spanning three domains. Being a manually collected metric, offender was difficult and time consuming to collect from all six projects. Therefore, we only collected it for the Chromium project, reusing a considerable portion of the data collected as part of a previous work [13]. The projects that we considered in our study are shown in Table I.

TABLE I
Projects from which the vulnerability discovery metrics were collected

Domain	Project	Language	Size*
Browser	Chromium	C/C++	9,054,450
	Firefox	C/C++	6,977,203
Operating System	Linux	C/C++	13,101,179
	OpenBSD	C/C++	9,147,222
Application Server	Tomcat	Java	326,748
	WildFly	Java	524,240

*Total number of source lines of code across all languages.

We only collected the metrics from file paths ending with the extensions `.c`, `.cc`, `.cpp`, `.cxx`, `.h`, `.hh`, `.hpp`, `.hxx`, or `.inl` for C/C++ projects and `.java` for Java projects.

B. Data Analysis

In the subsections that follow, we describe the series of analysis we performed on the vulnerability discovery metrics collected from the six open-source projects. We used `R` [14] version 3.5.1 to conduct our data analysis.

Assessing Normality

In using statistical methods, we must validate the assumption of normality to inform the choice of parametric or nonparametric statistical methods. We used the Anderson-Darling test to assess if the metrics collected from a project are normally distributed. We found, with statistical significance (p-value $\ll 0.001$), that no discrete- or continuous-valued metric was normally distributed in any of the projects.

Assessing Generalizability

In addressing RQ 1, we wanted to assess if the vulnerability discovery metrics proposed in the literature are generalizable. The interpretation of the term generalizability may be subjective. Therefore, we define the necessary (not sufficient) condition for generalizability as the need to have consistent

distribution. We assessed the generalizability condition of a metric by comparing the distribution of the metric values collected from different projects. We used the same approach that Zhang *et al.* [15] used to assess the impact of contextual dimensions on the distribution of metrics.

We used the nonparametric Kruskal–Wallis to assess if the sample distribution of metric values collected from projects, grouped by contextual dimensions, originated from the same underlying distribution. We consider the outcome from the Kruskal–Wallis test to be statistically significant if p-value $< 2.78e-03 = 0.05/9/2$ ($\alpha = 0.05$ corrected for multiplicity using Bonferroni Correction).

A statistically significant outcome from Kruskal–Wallis test would indicate that at least one metric distribution is different from the rest. To explain the difference(s) further, we supplement the outcome from Kruskal–Wallis test with observations from pairwise comparison of distribution of metric values using the nonparametric Mann–Whitney–Wilcoxon test.

We consider the outcome from the Mann–Whitney–Wilcoxon test to be statistically significant if p-value $< 7.94e-04 = 0.05/63$ ($\alpha = 0.05$ corrected for multiplicity using Bonferroni Correction). The inference from a statistically significant outcome from Mann–Whitney–Wilcoxon test is that the metric distributions being assessed are different. To quantify the magnitude of the difference in distributions, we used Cliff's δ [16], a nonparametric effect size measure. The difference between distributions is considered negligible when $\delta < 0.147$, small when $0.147 \leq \delta < 0.33$, medium when $0.33 \leq \delta < 0.474$, and large when $\delta > 0.474$ [17].

In summary, we consider a metric to satisfy the generalizability condition if it is similarly distributed irrespective of contextual dimensions and between projects within a domain with negligible to medium effect size (same as the threshold of large effect size used by Zhang *et al.* [15]).

Computing Thresholds

Vulnerability discovery metrics are typically evaluated using a supervised approach requiring data on historical vulnerabilities to train a prediction model that uses the metrics as explanatory variables. While some projects may have a curated list of historical vulnerabilities to satisfy this prerequisite, a model trained with data from one project may not be directly applicable to discover vulnerabilities in a different project [4]. An unsupervised approach is, therefore, needed.

An intuitive interpretation of a metric is to establish, and use, thresholds to label an entity being measured to exhibit certain attribute to a higher or lower degree. While computing a universal threshold for all projects is intractable, Lanza and Marinescu suggest using statistical information rankings to determine explicable thresholds [18].

In our study, we used the methodology proposed by Alves *et al.* [19] to compute the thresholds. Despite being an unsupervised approach, the methodology proposed by Alves *et al.* was found to be effective in the prediction of fault-proneness in comparison with other supervised approaches [20].

We compute the thresholds for only those metrics that satisfied our generalizability condition. As proposed by Alves *et al.* [19], we compute the thresholds by taking metric values from all projects together and use the same 70%, 80%, and 90% of the weighted (using the source lines of code metric) metric value averaged over all projects to determine the risk levels. We use the following risk levels to assess risk using a metric (m): low ($m < 70\%$), medium ($70\% \leq m < 80\%$), high ($80\% \leq m < 90\%$), and critical ($m \geq 90\%$).

In addressing RQ 2, we use the metrics' thresholds to classify the risk from metrics collected from known (historically) vulnerable files (determined by the offender metric). We quantify the effectiveness of such risk classification by expressing the coverage (i.e. percentage of vulnerable files covered) of and assessing the odds of finding a vulnerable file in each of the non-trivial risk levels (i.e. risk levels other than low).

IV. Results

In the subsections that follow, we present our empirical analysis and observations to address the research questions.

RQ 1: Generalizability

Question: *Are vulnerability discovery metrics similarly distributed across projects?*

Motivation. The literature has no shortage of security metrics with over 300 metrics proposed over the years [1]. However, as Morrison *et al.* report, a considerable portion of these metrics have either been evaluated solely by the authors who proposed the metrics (85%) or not empirically evaluated at all (40%) [1]. Furthermore, the empirically-evaluated metrics may have been evaluated in isolation in the context of a few projects with limited potential for generalizability. A key first step toward realizing the (near) universal adoption of these metrics is to assess if these metrics are generalizable across multiple projects. The assertion being that the insights that one can gain from a generalizable metric can be transferred from one project (perhaps one that has had engineering failures in the past) to another.

The motivation for the generalizability research question is to assess if security metrics, particularly vulnerability discovery metrics, are generalizable across projects.

Approach. The interpretation of the term generalizability may be subjective. In our study, we use the similarity in distribution of metrics across projects as a proxy for generalizability. The approach to address the research question involved supplementing insights from Krukal–Wallis test with that from Mann–Whitney–Wilcoxon test and Cliff's δ.

Observations. We use a traditional violin plot to compare the distribution of the nine discrete- and continuous-valued metrics in the six projects considered in our study. The plot provides us an inkling of the metrics that are likely to be generalizable. Shown in Figure 1 is a plot comparing the distribution of churn and collaboration metrics.[1] As can be inferred from the figure, churn appears to be similarly distributed across all projects (irrespective of domain and language). However, collaboration centrality seems to be similarly distributed between

[1] We chose churn and collaboration as exemplars to present our observations

TABLE II
MAGNITUDE OF DIFFERENCE IN THE DISTRIBUTION OF CHURN AND COLLABORATION METRICS SEPARATED BY TWO CONTEXTUAL DIMENSIONS (DOMAIN AND LANGUAGE) AND BETWEEN PROJECTS WITHIN A DOMAIN

Dimension	X	Y	Cliffs δ	
			Churn	**Collaboration**
Domain	BR	OS	0.2148 (S)	0.1881 (S)
	BR	AS	0.1928 (S)	0.3062 (S)
	OS	AS	0.0667 (N)	0.3110 (S)
Language	C/C++	Java	0.1497 (S)	0.3078 (S)
Project	Chromium	Firefox	0.0610 (N)	0.1043 (N)
	Linux	OpenBSD	0.2056 (S)	0.9915 (L)
	Tomcat	WildFly	0.1153 (N)	0.9955 (L)

Legend
BR - Browser, OS - Operating System, and AS - Application Server
Effect Size
(N) $\delta < 0.147$ (S) $0.147 \leq \delta < 0.33$ (L) $\delta > 0.474$

projects in the browser domain but differently distributed between projects in both the operating system and application server domains and, consequently, between domains.

We supplemented the qualitative inference from the plot by quantitatively assessing the similarity of distribution using the Krukal–Wallis test. The outcome from Krukal–Wallis test was statistically significant (p-value $< 2.78e-03$) for all metrics (including Churn) when separated by domain and language. The outcome indicates that the distribution of at least one sample of metrics between the three domains and two languages is *different* from the others. To understand the exact nature of the difference(s), we ran pairwise Mann–Whitney–Wilcoxon tests using Cliff's δ to assess the magnitude of difference. The outcome from the pairwise test for churn and collaboration metrics is shown in Table II. The effect size further provides credence to our inference from the plot that churn is similarly distributed across domains and languages and between projects within a domain. Collaboration, on the other hand, is similarly distributed across domains and languages but differently distributed between projects within the operating system and application server domains.

We summarize the observations from the statistical analyses in Table III. We found that, with the exception of the collaboration metric, all metrics are generalizable.

> All metrics, except collaboration, are generalizable (i.e. have similar distributions) across the projects considered in our study irrespective of domain and language.

A. RQ 2: Thresholds

Question: *Are thresholds of vulnerability discovery metrics effective at classifying risk from vulnerabilities?*

Motivation. The benefit of using metrics to support software development is the ability to make objective decisions based on quantifiable aspects of product, process, or people. However, for the metrics to be an effective tool to support decision making, there must be an objective way of saying when the value of a metric indicates specific scenarios. An example of this is the empirical study that found 200 lines of code per hour to be a threshold for individual reviews beyond which there may be degradation in defect discovery [21].

TABLE III
SUMMARY OF THE ASSESSMENT OF GENERALIZABILITY OF THE METRICS WITH ✓INDICATING THAT A METRIC WAS FOUND TO HAVE A SIMILAR DISTRIBUTION WITH NEGLIGIBLE TO MEDIUM EFFECT SIZE

Dimension	Churn	Collaboration	Complexity	Contribution	Nesting	# Inputs	# Outputs	# Paths	Source LOC
Domain	✓	✓	✓	✓	✓*	✓	✓	✓	✓
Language	✓	✓	✓	✓	✓	✓	✓	✓	✓
Project	✓	✗	✓	✓	✓	✓	✓*	✓	✓
Summary	✓	✗	✓	✓	✓	✓	✓	✓	✓

*Mann–Whitney–Wilcoxon p-value $> 7.94e-04$

The motivation for the thresholds research question is to leverage existing approaches to compute thresholds for the generalizable vulnerability discovery metrics considered in our study. While we assess the effectiveness of the thresholds to cover historically vulnerable files, our intention is to use the thresholds as triggers to determine if developers should be shown certain feedback about their change.

Approach. We used the methodology proposed by Alves *et al.* [19] to compute the thresholds using metric data collected from all six projects considered in our study. The threshold value for the metrics are chosen at the same quantiles (70%, 80%, and 90%) as that in the paper by Alves *et al.* with the risk levels for a metric (m) being low ($m < 70\%$), medium ($70\% \leq m < 80\%$), high ($80\% \leq m < 90\%$), and critical ($m \geq 90\%$). We assess the effectiveness of the thresholds in two ways, both of which make use of the historical vulnerabilities data from the Chromium project. Firstly, we quantify the distribution of the historically vulnerable files across the risk levels to ascertain the percentage of vulnerable files that each risk level captures. Secondly, we compute the odds of discovering an historically vulnerable file in each of the risk level. We also compute the odds ratio to assess the change in odds as we move from one risk level to the next.

Observations. In addressing RQ 1, we found the collaboration metric to not meet our criteria to be considered generalizable. Therefore, we do not include it in our analysis for RQ 2. Shown in Table IV are the threshold values computed for each of the eight generalizable metrics. By themselves, the metric threshold values provide limited (if any) insights; their utility is to help segregate files into disjoint groups to highlight risk. The risk levels of medium, high, and critical are non-trivial and thus we do not consider the risk level of low in the remainder of this section.

Shown in Table V is the percentage of historically vulnerable files (offenders) in Chromium covered by each of the three non-trivial risk levels. The percentages indicate the effectiveness of the thresholds, which, on average, captured

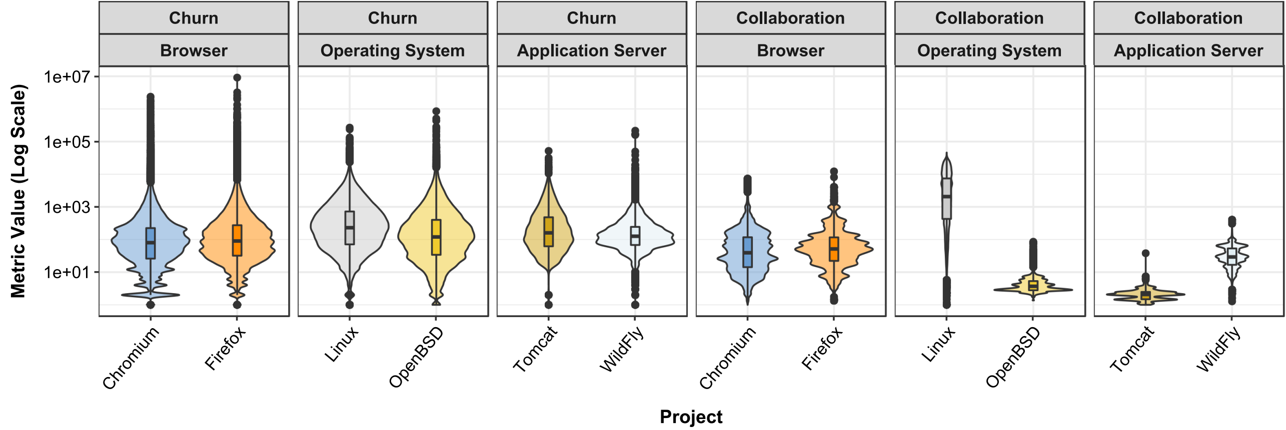

Fig. 1. Comparing the distribution of churn and collaboration metrics across all projects considered in our study

TABLE IV
THRESHOLD VALUE OF THE EIGHT GENERALIZABLE VULNERABILITY DISCOVERY METRICS

Metric	Quantile		
	70%	**80%**	**90%**
Churn	3,403	5,682	12,005
Complexity	197	336	710
Contribution	5.53E+04	1.59E+05	4.95E+05
Nesting	4	5	6
# Inputs	256	412	865
# Outputs	261	415	787
# Paths	4.15E+03	7.86E+04	6.97E+07
Source LOC	1,099	1,826	3,695

23.85% of vulnerable files, going as high as 69.85%, in aggregate, for the contribution metric.

TABLE V
PERCENTAGE OF VULNERABLE FILES COVERED BY EACH OF THE THREE NON-TRIVIAL RISK LEVELS (MEDIUM, HIGH, AND CRITICAL)

Metric	% Vulnerable Files Covered			
	Medium	**High**	**Critical**	**Aggregate**
Churn	7.68%	8.39%	5.85%	21.92%
Complexity	6.62%	5.96%	4.64%	17.22%
Contribution	15.90%	18.95%	35.01%	69.85%
Nesting	9.24%	5.95%	4.27%	19.47%
# Inputs	4.97%	4.97%	4.30%	14.24%
# Outputs	7.28%	6.62%	6.62%	20.53%
# Paths	5.30%	5.96%	4.97%	16.23%
Source LOC	6.26%	3.44%	1.68%	11.37%
Average	7.91%	7.53%	8.42%	23.85%

While the percentage of historical files covered by the risk levels is interesting, characterizing the effect of a file moving from one risk level to another is essential if we need to be able to use thresholds to support any sort of decision making. Shown in Table VI is the odds of discovering a vulnerable file (from the Chromium project) in each of the three non-trivial risk levels. The odds ratios shown in the table quantify the increase in odds of discovering vulnerable files as a file moves from one risk level to next. We also present the ratio of odds in a risk level to that in low. By interpreting the odds ratios we can make inferences such as a file being 14 times more likely to be vulnerable when it moves from low to medium on the churn threshold scale (Odds $\text{Ratio}_{\text{Low}}$ = 13.9862).

On average, non-trivial risk levels delineated by thresholds of generalizable vulnerability discovery metrics captured 23.85% of the historically vulnerable files in Chromium, providing support for the effectiveness of the thresholds in classifying risk from vulnerabilities.

V. TRANSLATING INSIGHTS TO AUTOMATED FEEDBACK

An essential next step—one which we are actively pursuing—is to translate the interpretation of the vulnerability discovery metrics to automated feedback on security. Our approach is inspired by the need to accelerate the adoption of software engineering research in the industry [22] and to integrate research with practice in as seamless a way as possible. Our approach to achieve this translation is predicated upon addressing three key concerns: (1) **when** should the feedback be shown?, (2) **what** should the feedback contain?, and (3) **where** should the feedback be provided?

In this paper, while we found that the risk levels delineated by the metrics' threshold were effective at capturing a considerable portion of historically vulnerable files in Chromium, we are likely to produce high false positives if we simply flag changes to any file that fall into a particular non-trivial risk level. The utility of the thresholds, however, is in classifying risk, or rather, more importantly, classifying the change in risk. We can use change in risk as a trigger to highlight developers'

TABLE VI
ODDS OF DISCOVERING A VULNERABLE FILE IN EACH OF THE THREE NON-TRIVIAL RISK LEVELS (MEDIUM, HIGH, AND CRITICAL) WITH ODDS RATIO$_X$ BEING THE RATIO OF ODDS IN A PARTICULAR RISK LEVEL TO ODDS IN RISK LEVEL X

Metric	**Medium**		**High**			**Critical**		
	Odds	**Odds Ratio$_{\text{Low}}$**	**Odds**	**Odds Ratio$_{\text{Low}}$**	**Odds Ratio$_{\text{Medium}}$**	**Odds**	**Odds Ratio$_{\text{Low}}$**	**OR$_{\text{High}}$**
Churn	4.57E-02	13.9862	7.96E-02	24.3699	1.7424	1.12E-01	34.3548	1.4097
Complexity	9.48E-02	4.2210	1.61E-01	7.1569	1.6955	2.86E-01	12.7234	1.7778
Contribution	1.84E-02	3.8177	3.52E-02	7.3282	1.9195	1.16E-01	24.1669	3.2978
Nesting	6.36E-02	3.8456	7.39E-02	4.4739	1.1634	8.92E-02	5.3960	1.2061
# Inputs	7.65E-02	3.2952	1.32E-01	5.6655	1.7193	1.97E-01	8.4811	1.4970
# Outputs	7.10E-02	3.2246	8.70E-02	3.9511	1.2253	2.41E-01	10.9488	2.7711
# Paths	6.84E-02	2.9648	9.73E-02	4.2188	1.4230	1.29E-01	5.6069	1.3290
Source LOC	1.89E-01	10.8511	1.93E-01	11.0919	1.0222	2.68E-01	15.4085	1.3892

changes to a particular file addressing the concern on **when** the feedback should be shown.

While the change in risk is useful as a trigger, indicating that a file is now n times more likely to be vulnerable because it moved from medium to high risk is of little use to the developer. The interpretation of the vulnerability discovery metrics is needed to contextualize the change in risk. We propose using interpretable models such as Logistic Regression, Decision Trees, and, the more recent, Decision Sets [23] to characterize the role of each vulnerability discovery metric in a model. We can then use the characterization to generate feedback describing the impact that developers' change may have on factors known to be associated with vulnerabilities addressing the concern on **what** the feedback should contain.

Developers prefer their tools to integrate seamlessly into the their existing workflows [24]. Code review, being a commonplace in most mature software engineering workflows today, provides the ideal opportunity to provide feedback on security [25]. TRICORDER, a program analysis ecosystem at Google, does just this but with static analysis warnings [26]. We propose using code reviews as a medium to provide the automated feedback on security addressing the concern on **where** the feedback should be provided. The use of code review also enables developer-in-the-loop improvements to our thresholds and/or feedback.

Developers tend to have specific expectations of the tools that they use [27], [24], [28]. The vulnerability discovery metrics, when used appropriately, have the potential to meet some, if not all, of these requirements. In using the thresholds and interpretation of the vulnerability discovery models to provide security feedback as code review comments, we hope to assist developers in engineering secure software in a way that they perceive to be useful. As Lakkaraju *et al.* [23] state "Interpretable models ... bridge the gap between domain experts and data scientists.", we, as researchers, should work toward translating our metrics to interpretable and actionable insights for developers. We hope that, in the near future, when a developer submits a change to a file `foo.c` that increases its risk from low to medium as triggered by the churn metric, we can provide a feedback like "The change to `foo.c` has churned 200 lines of code increasing its odds of needing a fix for a security vulnerability by 5.48%."

VI. LIMITATIONS

Generalizability is a concept that is much broader than demonstrating similar distributions. However, in using similarity in distribution as a necessary condition for metric generalizability, we can reason about the potential for statistically-derived thresholds from one project to translate to another. In Section V, we alluded to our approach using developer-in-the-loop to help continually improve our feedback technique and we hope this loop will help strengthen the evidence of generalizability as developers provide their comments on the threshold-driven feedback.

Although we chose projects spanning three application domains and two programming language, we may need to consider more projects to be able to apply our technique at a broader scale. Some of the projects not represented in our sample are small projects with short histories, projects developed in interpreted languages, and closed-source (proprietary) project. Fortunately, most of our analysis is unsupervised and we hope can be easily applied to additional projects.

VII. SUMMARY

Our research vision is to assist developers in engineering secure software by providing a technique that generates scientific, interpretable, and actionable feedback on security as the software evolves. In this exploratory research study, we collected ten vulnerability discovery metrics from six open-source projects and assessed their generalizability in terms of similarity in distribution and computed thresholds for the discrete- and continuous-valued metrics using an unsupervised technique proposed by Alves *et al.* [19]. With the exception of one, all discrete- and continuous-valued metrics satisfied our criteria for generalizability. We also found the thresholds to be effective at classifying risk from historically vulnerable files in the Chromium project. The work described here is the first step in a much broader project to develop an automated vulnerability discovery interpretation technique to provide developers valuable security insights as they develop software.

ACKNOWLEDGMENTS

Our work is supported in part by the DARPA Small Business Innovation Research program, the U.S. National Science Foundation CyberCorps® Scholarship for Service program (1433736), and the U.S. National Security Agency Science of Security Lablet at North Carolina State University (H98230-17-D-0080/2018-0438-02).

REFERENCES

[1] P. Morrison, D. Moye, R. Pandita, and L. Williams, "Mapping the field of software life cycle security metrics," *Information and Software Technology*, vol. 102, pp. 146–159, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S095058491830096X

[2] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with Applying Vulnerability Prediction Models," in *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, ser. HotSoS '15. New York, NY, USA: ACM, 2015, pp. 4:1–4:9. [Online]. Available: http://doi.acm.org/10.1145/2746194.2746198

[3] N. E. Fenton and M. Neil, "Software Metrics: Roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 357–370. [Online]. Available: http://doi.acm.org/10.1145/336512.336588

[4] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 91–100. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595713

[5] C. Theisen, K. Herzig, P. Morrison, B. Murphy, and L. Williams, "Approximating Attack Surfaces with Stack Traces," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, ser. ICSE '15, vol. 2, May 2015, pp. 199–208.

[6] N. Munaiah and A. Meneely, "Beyond the attack surface: Assessing security risk with random walks on call graphs," in *Proceedings of the 2016 ACM Workshop on Software PROtection*, ser. SPRO '16. New York, NY, USA: ACM, 2016, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/2995306.2995311

[7] N. Munaiah, "Assisted Discovery of Software Vulnerabilities," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 464–467. [Online]. Available: http://doi.acm.org/10.1145/3183440.3183453

[8] N. Munaiah and A. Meneely, "Systematization of Vulnerability Discovery Knowledge: Review Protocol," *arXiv e-prints*, p. arXiv:1902.03331, Feb 2019. [Online]. Available: https://arxiv.org/abs/1902.03331

[9] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista," in *2010 Third International Conference on Software Testing, Verification and Validation*, apr 2010, pp. 421–428.

[10] A. Meneely and L. Williams, "Secure Open Source Collaboration: An Empirical Study of Linus' Law," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 453–462. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653717

[11] A. Younis, Y. Malaiya, C. Anderson, and I. Ray, "To Fear or Not to Fear That is the Question: Code Characteristics of a Vulnerable Function with an Existing Exploit," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '16. New York, NY, USA: ACM, 2016, pp. 97–104. [Online]. Available: http://doi.acm.org/10.1145/2857705.2857750

[12] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates, "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," *International Symposium on Empirical Software Engineering and Measurement*, pp. 65–74, 2013.

[13] N. Munaiah, B. S. Meyers, C. O. Alm, A. Meneely, P. K. Murukannaiah, E. Prud'hommeaux, J. Wolff, and Y. Yu, "Natural Language Insights from Code Reviews that Missed a Vulnerability," in *Engineering Secure Software and Systems: 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings*, E. Bodden, M. Payer, and E. Athanasopoulos, Eds. Cham: Springer International Publishing, 2017, pp. 70–86. [Online]. Available: https://doi.org/10.1007/978-3-319-62105-0_5

[14] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2018. [Online]. Available: https://www.R-project.org/

[15] F. Zhang, A. Mockus, Y. Zou, F. Khomh, and A. E. Hassan, "How Does Context Affect the Distribution of Software Maintainability Metrics?" in *2013 IEEE International Conference on Software Maintenance*, Sep. 2013, pp. 350–359.

[16] G. Macbeth, E. Razumiejczyk, and R. Ledesma, "Cliff's Delta Calculator: A non-parametric effect size program for two groups of observations," *Universitas Psychologica*, vol. 10, pp. 545 – 555, 05 2011. [Online]. Available: http://www.scielo.org.co/scielo.php?script=sci_arttext&pid=S1657-92672011000200018&nrm=iso

[17] J. Romano, J. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys," in *Annual meeting of the Florida Association of Institutional Research*, 2006, pp. 1–33.

[18] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer Berlin Heidelberg, 2006. [Online]. Available: https://doi.org/10.1007/3-540-39538-5

[19] T. L. Alves, C. Ypma, and J. Visser, "Deriving Metric Thresholds from Benchmark Data," in *2010 IEEE International Conference on Software Maintenance*, 2010, pp. 1–10.

[20] A. Boucher and M. Badri, "Software metrics thresholds calculation techniques to predict fault-proneness: An empirical comparison," *Information and Software Technology*, vol. 96, pp. 38–67, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584916303135

[21] C. F. Kemerer and M. C. Paulk, "The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 534–550, July 2009.

[22] I. Beschastnikh, M. F. Lungu, and Y. Zhuang, "Accelerating Software Engineering Research Adoption with Analysis Bots," in *Proceedings of the 39th International Conference on Software Engineering: New Ideas and Emerging Results Track*, ser. ICSE-NIER '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 35–38. [Online]. Available: https://doi.org/10.1109/ICSE-NIER.2017.17

[23] H. Lakkaraju, S. H. Bach, and J. Leskovec, "Interpretable Decision Sets: A Joint Framework for Description and Prediction," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 1675–1684. [Online]. Available: http://doi.acm.org/10.1145/2939672.2939874

[24] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*, may 2013, pp. 672–681.

[25] A. Bacchelli and C. Bird, "Expectations, Outcomes, and Challenges of Modern Code Review," in *2013 35th International Conference on Software Engineering (ICSE)*, ser. ICSE '13, May 2013, pp. 712–721.

[26] C. Sadowski, J. v. Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a Program Analysis Ecosystem," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, may 2015, pp. 598–608.

[27] L. Layman, L. Williams, and R. S. Amant, "Toward Reducing Fault Fix Time: Understanding Developer Behavior for the Design of Automated Fault Detection Tools," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, sep 2007, pp. 176–185.

[28] S. L. Foss and G. C. Murphy, "Do Developers Respond to Code Stability Warnings?" in *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '15. Riverton, NJ, USA: IBM Corp., 2015, pp. 162–170. [Online]. Available: http://dl.acm.org/citation.cfm?id=2886444.2886469