A New Approach to Predict Security Vulnerability Severity in Attack Prone Software Components Using Architecture and Repository Mined Change Metrics

Journal:	*Transactions on Dependable and Secure Computing*
Manuscript ID	Draft
Manuscript Type:	Regular Paper
Keywords:	D.4.6.e Invasive software < D.4.6 Security and Privacy Protection < D.4 Operating Systems < D Software/Software Engineering, D.2 Software Engineering < D Software/Software Engineering, D.2.11 Software Architectures < D.2 Software Engineering < D Software/Software Engineering, D.2.15 Software and System Safety < D.2 Software Engineering < D Software/Software Engineering, D.2.19 Software Quality/SQA < D.2 Software Engineering < D Software/Software Engineering, K.6.5 Security and Protection < K.6 Management of Computing and Information Systems < K Computing Milieux

A New Approach to Predict Security Vulnerability Severity in Attack Prone Software Components Using Architecture and Repository Mined Change Metrics

Daniel Hein, *Member, IEEE* and Hossein Saiedian, *Senior Member, IEEE*

Abstract—Billions of dollars are lost every year to successful cyberattacks that are fundamentally enabled by software vulnerabilities. Modern cyberattacks increasingly threaten individuals, organizations, and governments, causing service disruption, inconvenience, and costly incident response. Given that such attacks are primarily enabled by software vulnerabilities, this work examines the efficacy of using change metrics, along with architectural burst and maintainability metrics, to predict modules and files that might be analyzed or tested further to excise vulnerabilities prior to release. Traditional code complexity metrics, along with newer frequency based churn metrics (mined from software repository change history), are selected specifically for their relevance to the residual vulnerability problem. We compare the performance of these complexity and churn metrics to architectural level change burst metrics, automatically mined from the `git` repositories of the Mozilla Firefox Web Browser, Apache HTTP Web Server, and the MySQL Database Server, for the purpose of predicting attack prone files and modules. We offer new empirical data quantifying the relationship between our selected metrics and the severity of vulnerable files and modules, assessed using severity data compiled from the NIST National Vulnerability Database, and cross-referenced to our study subjects using unique identifiers defined by the Common Vulnerabilities and Exposures (CVE) vulnerability catalog. Specifically, we evaluate our metrics against the severity scores from CVE entries associated with known-vulnerable files and modules. We use the severity scores according to the Base Score Metric from the Common Vulnerability Scoring System (CVSS), corresponding to applicable CVE entries extracted from the NIST National Vulnerability Database, which we associate with vulnerable files and modules via automated and semi-automated techniques. Our results show that architectural level change burst metrics can perform well in situations where more traditional complexity metrics fail as reliable estimators of vulnerability severity. In particular, results from our experiments on Apache HTTP Web Server indicate that architectural level change burst metrics show high correlation with the severity of known vulnerable modules, and do so with information directly available from the version control repository change-set (i.e., commit) history.

Index Terms—Software Vulnerability, Vulnerability Prediction, Software Security Metrics, CVSS Metrics, Mining Software Repositories

1 Security Vulnerabilities and Software

The economic externalities stemming from exploited security vulnerabilities in software are a multi-billion dollar problem [35]. Each year, world-wide economies, corporations, and individuals shoulder financial damage stemming from successful attacks on computers and networks–attacks ultimately enabled by software vulnerabilities. Such attacks lead to lost productivity, data disclosure, and identity theft. In response, several organizations have adopted secure software development practices, including practices such as code review, use of static analysis tools, and penetration testing to remove these attack enabling vulnerabilities.

The goal of secure software development practices is to eliminate vulnerabilities before they enter the field. However, exhaustive application of secure development practices for all code is often not feasible or cost effective. While it may seem reasonable for an organization to apply secure development practices to all newly written code, one must simultaneously acknowledge that not all code is new. The sheer size of modern software, along with the reuse of existing open source modules, complicates the questions of where to look, and in what order to look, for security vulnerabilities.

The problem addressed by this research is the residual vulnerability problem – the presence of attack enabling security vulnerabilities in released software. Residual vulnerabilities in modern software systems such as mobile phones, personal computers, Web servers, and IoT devices allow attackers to compromise both the host systems and the networks to which they are connected. Secure software engineering practices seek to identify and remove these vulnerabilities prior to release, but the application and management of these practices may be nontrivial in practice. Project size, maturity, and development team culture all play a part in effectively applying secure engineering practices.

The level of effort to unearth vulnerabilities and remove them under time-to-market pressure in modern software is at odds with the level of effort required by attackers to find and exploit vulnerabilities. The effort required by an attacker to violate software security defenses is often linear, requiring an attacker to find a single weakness to exploit.

- *Dan Hein, Ph.D., is with Garmin International, Olathe, KS, 66062. E-mail: daniel.hein@garmin.com*
- *Hossein Saiedian, Ph.D., is with the University of Kansas.*

This work was supported in part by Garmin International

As Bellovin [7] states, "whatever the defense, a single well-placed blow can shatter it." On the other hand, the effort required to *assure* that software is secure is exponential, requiring exhaustive and comprehensive knowledge of the software and all its possible interactions with its environment; this simply isn't tractable [7].

1.1 Significance

The significance of this research is in optimizing resource utilization. One of the most expensive and limited resources in modern software development is developer time. Additional review and testing of various software artifacts (e.g. architectural designs, drivers, libraries, modules, and application code) is only warranted if there is reason to believe those artifacts may be attack prone. Moreover, the supply of time and expertise from security-competent developers and testers is arguably more restricted due to the specialized knowledge required to unearth security vulnerabilities. Reviewing code from a security perspective requires a level of expertise in vulnerability assessment and attacker-minded thinking that few developers possess [40]. This research aims to intelligently focus these energies on those artifacts most likely to exhibit vulnerabilities.

This research complements existing tools and predictive approaches which have already examined the use of code-based metrics as predictors for implementation defects. However, this work more closely examines evolutionary and architectural aspects of the software by studying the relationship between A) code change and vulnerability severity, and B) module interconnections and vulnerability severity.

Consider a legitimate vulnerability identified in the system, such as in third party software component. Understanding its *likelihood of its execution* is critical to driving remediation. Likelihood of execution is a practical aspect more recently acknowledged by Younis, Malaiya, and Ray [44][45].

We estimate likelihood of execution using metrics that characterize a component's connection to system entry points [22], such as front line functions [11], exit points, channels, and untrusted data. Attacker-formed data will enter the system at these boundaries.

The addition of architectural considerations, and metrics reflecting the same, better enable the results from vulnerability prediction models to be used as tool for security improvement within secure software engineering life cycles. We evaluate the feasibility of using metrics that capture architectural relationships in prioritizing attack-prone code units to better guide review and penetration testing efforts. Section 2.2 provides additional background on using CVSS as a measure of severity.

1.2 Contributions

The key contributions of this research are as follows:

- Introduce practical techniques based on CVSS and `git` to automate training set construction for vulnerability prediction,
- Examine how change metrics correlate with residual vulnerability severity,
- Examine how metrics characterizing module interconnections and their interfaces correlate with vulnerability severity,
- Publish empirical data on change burst metrics to add to the growing body of literature on defect and vulnerability prediction in open source projects.

Vulnerability identification and removal is made more tractable and cost effective by prioritizing components suggested for extended security review and penetration testing. Establishing priority is important because it recognizes that not all vulnerabilities are created equal. Paraphrasing a common saying, "Nothing is top priority when everything is top priority".

1.3 Organization

Section 2 - Role of Metrics in Prediction Models covers relevant terms and concepts, as well as related work in vulnerability detection. Section 3 - Vulnerability Prediction Modeling and Evaluation provides a more detailed view of the prediction model building process. We also provide additional detail on metrics extracted. Section 4 - Repository Mining Approach outlines our research approach and data mining methodology. Section 5 - Experimental Results, and Analysis of Vulnerability Predictions describes experimental results and analysis for each software case study. Section 6 - Contributions and Future Work concludes the work with parting thoughts and ideas for future work that may build upon this work.

2 Role of Metrics in Prediction Models

This work bridges the fields of secure software engineering (SSE) and empirical software engineering (ESE). Secure software engineering is primarily concerned with methods and techniques to both prevent the introduction of, as well as detect and remove, vulnerabilities prior to release [15]. Empirical software engineering seeks to understand software quality through experimentation, data collection, and analysis [41], [16].

Within ESE, mining software repositories (MSR) has emerged as its own area of research [14]. MSR uses artifacts from a software project for knowledge discovery (e.g. frequent item set and association rule mining) [43] as well as to support or refute investigative questions (e.g. do last minute changes introduce vulnerabilities?). Commonly mined artifacts include the software version control system, bug reports, and mailing lists of software projects [42].

2.1 Metric Based Prediction Models and Related work

Prediction models within empirical software engineering (ESE) typically look at various attributes or properties of a software file (or its history) as opposed to scanning files line by line for a specific problematic pattern; in contrast, commercial static analysis tools scan files line by line for problematic functions, data flow sequences, and control flow sequences. The various attributes examined by prediction models are often colloquially referred to as *predictors* or *features*, but these attributes, or their derivatives, either form the explanatory variables in statistical regression models, or form the inputs for (typically supervised) machine learners.

The following sections provide additional detail on metrics studied by prior fault prediction literature. Related work on the predictive capability of various metrics and applicability for vulnerability prediction is discussed.

2.1.1 Code Metrics

Several studies [27], [39], [29], [9] have examined the relationship between residual defects and static metrics extracted from source code. For example, sheer size of a file in lines of code (LOC), or more commonly in thousands of lines of code (KLOC), has been studied as having a bearing on residual defects based on the premise that larger more complex code is more difficult to understand and comprehend. Another popular metric is McCabe's Cyclomatic complexity which measures the number of paths through a program. The premise behind McCabe's cyclomatic complexity relates to the difficulty in achieving adequate branch and path coverage during testing.

In the area of fault prediction, Gimothy et al. [12] set several precedents for fault prediction studies: use of large, real-world open source software, applying linear and logistic analysis, independent evaluation of univariate predictors, as well as applying machine learning techniques, and 10-fold cross validation for training and testing.

A related work by Janzen and Saiedian [17], [18], [19] considered a large number of software architecture metrics to examine the impact of test-driven development (TDD) on software architecture. Their objective was to provide a comprehensive and empirically sound evidence and evaluation of the TDD impact on software architecture and internal design quality. Their research result demonstrated that software developers applying a TDD approach are likely to improve some software quality aspects at minimal cost over a comparable test-last approach.

2.1.2 Change Metrics: Churn and Change Bursts

We refer to metrics characterizing the change in code-based attributes over time as *change metrics* or *historical metrics* to distinguish them from more traditional static *code metrics*. In contrast to static code metrics, change and historical metrics require a version control system (VCS) for calculation because they are not directly obtainable from a single snapshot of the source code. Two such change metrics are churn (i.e., code churn), introduced by Munson and Elbaum [26], and change bursts. Below, we summarize related work in fault prediction and defect estimation inspires our efforts in the context of vulnerability prediction.

Churn refers to the number of code lines added, deleted, or changed over a specified time interval; a convenient time interval specification is often the time between product releases [29]. Change bursts refer to consecutive changes over a period of time [28].Khoshgoftaar et al. [20] use churn relative to bug changes, as the number of lines added or changed to fix the bug. Nagappan and Ball [29] demonstrated how to use relative code churn as an estimator for system defect density.

Change bursts are described by gap and burst size. The gap is the maximum distance (e.g., in days) between successive changes, such that those changes are considered within the same burst. The burst size is the minimum number of successive changes required to be considered a burst. Nagappan et al. more recently studied change bursts as predictors of residual defects in Windows Vista. For their study on Windows Vista, they found that change burst metrics outperformed all previous predictors, such as code complexity, code churn, and organizational structure, yielding precision and recall values over 90% [28].

2.1.3 Entropy and Historical Metrics

Entropy characterizes patterns and redundancy, and as such, is frequently used to evaluate data compression techniques. The more pattern and structure, the lower the entropy. As the distribution of the expected values of X approach equiprobability, its entropy likewise increases, reaching *maximum entropy* for a uniform random distribution.

Hassan [13] presents several complexity metrics based on historical changes, calculating entropy for the file modifications within a change period (e.g., a week). Hassan's entropy based, historically derived measures were shown to out perform both prior faults and prior modifications as a predictor of future faults for the open source systems he studied.

2.1.4 Architectural Modularity Metrics

Sarkar et al. [34] describe a number of information theoretic metrics that represent module interactions in a system, or modularity. We submit that the modularity principles outlined by Sarkar et al. such as similarity of purpose and encapsulation also echo some of the classic security design principles of Saltzer and Schroeder [33]. For example, Saltzer and Schroeder's design principle of complete mediation, where every object access must be checked for proper authority, is enabled by a design that routes all inter-module call traffic through a well defined API. Sarkar et al.'s Module Interaction Index, (MII), is a modularity metric characterizing the modularity principle of *maximization of API-based inter-module call traffic*–an underlying principle of encapsulation. MII is the ratio of external calls made to a module's API functions relative to the total number of external calls made to the module. Low MII could indicate direct usage of shared memory or direct global memory references. We might expect MII to inversely correlate with security vulnerabilities manifesting from unmediated changes to global variables, ultimately characteristic of poor encapsulation.

2.1.5 Vulnerability Prediction Models

Our study is informed by similar empirical vulnerability prediction studies by Shin [37], Ayanam [6], Gimothy et. al. [12], and Bozorgi et. al. [8]. Our work is most closely related to that of Shin and Ayanam, as both researchers investigated coupling metrics as vulnerability predictors. Shin's and Ayanam's respective works build on a long tradition of complexity metrics used to predict faults. Vulnerability ranking approaches are informed by the work of Bozorgi et. al.

Ayanam studied coupling metrics derived descended from the lineage of metrics inspired by Chidamber and Kemerer [9]. The notion of coupling is embodied in the some of the metrics from Sarkar et. al. [34] that we investigate. As mentioned in Section 2.1.3, we investigate architectural

modularity metrics that characterize economy of mechanism and Complete Mediation. We are also interested in information flow metrics, based on the idea that attacks are often executed by manipulating input data.

A closely related study for vulnerability prediction was provided by Yonghee Shin. The results from Shin's study indicate that certain change and developer oriented metrics are able to discriminate between vulnerabilities and the larger class of standard issue defects. Shin [38], [37] examined churn in addition to several other change metrics mined from software projects' version control systems. Shin's study was likewise focused on security and vulnerability prediction. Shin sought to answer whether or not these metrics could also be used to identify *vulnerable* files. Shin also examined developer oriented graph metrics. Shin's results showed developer oriented metrics and change metrics yielding the best performance on the projects she studied.

2.2 Vulnerability Scoring and Ranking

CVSS scores [23] are listed in the on-line National Vulnerability Database (NVD) [31]. Such scores encapsulate expert vulnerability knowledge and provide a basis for ranking vulnerabilities. We evaluate our vulnerability ranking techniques based on architectural metrics against the order imposed by CVSS *base* scores. We are aware of the criticisms of CVSS base scores by Bozorgi et. al. [8] as a standard against which to evaluate ranking, but we submit that our usage is different in the context of ranking the predictions of residual vulnerabilities. The following paragraphs recapitulate Bozorgi et al.'s critique and then compares the differences in the context of our application and intent.

The following are important differences between our work and that of Bozorgi et. al. [8], considering that their work is concerned with prioritizing patch selection and application to operational systems:

- **Prediction error:** Our prediction models will have some prediction error, such as a given false positive rate. This factor doesn't impact Bozorgi et. al. since their false positive rate with respect to this dimension is 0; that is, they already know the vulnerability exists, as well as the fix.
- **Time independence:** Bozorgi et. al. notes a significant difference on exploitation likelihood based on time.

A key difference between our context and that of Bozorgi et al. is that of time. In their work, the age of a vulnerability was a significant factor in determining exploitation likelihood–attackers may be less likely to exploit a vulnerability the older it gets. In contrast, our context is one where any vulnerability could potentially be a zero day exploit.

3 Vulnerability Prediction Modeling Process and Evaluation

This section describes the detailed steps involved in building and evaluating prediction models. The following sections provide detail on regression models, metric correlation Analysis, and evaluation. Evaluation of metrics is relative to the strength of the rank correlation with advisory CVSS [23].

3.1 Building ESE Prediction Models

Shröter, Zimmerman, and Zeller [36], along with Nagappan, Ball, and Zeller [30], as well as Chowdhury and Zulkernine [10], clearly describe the process of building predictive classification and ranking models based on post-release defects. Their model building process is relevant because we are using security advisory reports to identify residual vulnerabilities. Residual vulnerabilities are a subset of the more general post-release defects (i.e., used in the predictive model building process).

Following the steps outlined by Naggappan, Ball, and Zeller, the end result over several versions of a product, is a completed training and evaluation database (TEDB). Each record in the TEDB contains several metrics computed per release. This database is then used to build prediction models by using statistical techniques (e.g. least squares regression) and machine learning classifiers on a portion of the collected data (e.g, the training data). After the prediction models are built, a different portion of the collected data (e.g. the evaluation or test data) is used to test the predictions generated by the models. Since the test data is already labeled as "vulnerable" ($Vuln^+$) or "neutral" ($Vuln^-$), a confusion matrix relating the accuracy of the predictions to the actual values can be generated. Table 1 shows such a confusion matrix, where N represents the total number of samples (e.g., files or modules).

Table 1
Detailed confusion matrix

		Actual		
		$Vuln^+$	$Vuln^-$	Total
Predicted	$Vuln^+$	TP	FP	$TP+FP$
	$Vuln^-$	FN	TN	$FN+TN$
	Total	$TP+FN$	$FP+TN$	N

3.2 Linear and Logistic Regression

In vulnerability prediction, *linear regression* is used to estimate the number of residual vulnerabilities in a file or a module from either a single explanatory variable (i.e., simple linear regression) or multiple explanatory variables (i.e., multiple linear regression). *Logistic regression*, on the other hand, serves as a binary classifier, mapping the the response of the dependent variable into one of two classes:

- $Vuln^+$: a file or module contains one or more vulnerabilities
- $Vuln^-$: a file or module is assumed to be neutral with respect to vulnerabilities

Simple Linear Regression Simple linear regression (SLR), also known as least squares regression, shown in Equation 1, is used extensively in fault prediction literature to perform univariate evaluation of individual metrics, calculating β_1 so as to minimize the sum of squared residuals ($\sum_{i=1}^{n}(y_i - x_i)^2$), where residuals are the difference between sampled observations of the dependent variable y and the explanatory variable x.

$$y = \beta_0 + \beta_1 x \quad (1)$$

Multiple Linear Regression Multiple linear regression is an extension of SLR for more than one variable. Rather than attempting to fit a single line to minimize error, multiple coefficients inside β are used to fit X to Y:

$$\mathbf{Y} = \beta_0 + \beta\mathbf{X} \tag{2}$$

3.3 Correlation Analysis

With our data sets sanitized and labeled, we compute Spearman rank order correlation [25] , ρ (Equation 3), for each of our architectural modularity and maintainability metrics, for each version, with respect to the severity of the advisories logged against affected files and modules. Severity at the module level is calculated as both the sum and average of CVSS scores (y_i) for any advisories logged against the module for each release.

$$\rho = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}} \tag{3}$$

Notice we use the form of Spearman correlation, ρ that accounts for tied ranks. Given the limited range of CVSS values, it is possible that several results "bin" to identical ranks.

3.4 Metrics Extracted and Calculated

This section details the various metrics collected, discussing the implications of using the various metrics for residual vulnerability prediction. Although we are primarily interested in evaluating *architectural* and *change* based metrics, we also include some traditional *code metrics* such as McCabe's Cyclomatic Complexity and KLOC for baseline evaluation and comparison to past studies.

3.4.1 Notation

The following sections elaborate on the various metric categories, providing metric calculation formulas. The formulas are dependent on a notation for a software system, $\mathcal{S}$. Note that we combine various notation schemes from Sarkar et. al. [34] and Hassan [13]:

- $\mathcal{S}$ consists of a set of modules, $\mathcal{M} = \{m_1, m_2, \ldots m_M\}$, where $|\mathcal{M}| = M$, the number of modules.
- f^a denotes a function that belongs to a module's API.
- $K(f)$ denotes calls made to function f. $K_{ext}(f)$ denotes the number of calls to a file from other modules. $K_{int}(f)$ denotes calls made from within the same module, and where $K(f) = K_{ext}(f) + K_{int}(f)$.
- $K_{ext}(m)$ is the number of external function calls made to module m. A module with $f_1 \ldots f_n$ functions will have $K_{ext}(m) = \sum_{f \in \{f_1 \ldots f_n\}} K_{ext}(f)$

3.4.2 Architectural Modularity Metrics

The architectural modularity metrics presented in this section quantify modularity principles enumerated by Sarkar et al. citeSarkarInfoTheoryMetricModularization07. The modularity metrics quantify modularity principles that may likewise impact security properties. The module interaction index, MII, for example, quantifies the extent to which external calls to a module, honor the API provided by the module, as opposed to calling directly into private functions and methods. The MII measures the portion of all calls made to a module that are also routed through that module's API. We submit that such a property is also useful for security. The following paragraphs frame this concept more concretely using a hypothetical authorization module as an example.

Consider a module that is responsible for handling authorization. It is possible for the MII values to range between 0 and 1: $0 \leq MII(authorization) \leq 1$. An authorization module with MII of 0 implies that the module is either not being used, or that any users of the module are bypassing the authorization module's API. In the ideal case, $MII = 1$, indicating that all calls to the authorization module in fact utilize that module's API.

In general, we theorize that MII values closer to 0, and perhaps below some project specific threshold, are inversely correlated with security advisories and patches involving a module or its called functions, even if the module is not directly related to a security feature or function. Unintended side effects resulting from system maintenance or modification would be more likely since encapsulation is violated. Such side effects have the potential to violate security.

The following equations utilize the notation from Section 3.4.1 to describe relevant metrics:

- **Module interaction index (MII)**

$$MII(m) = \frac{\sum_{f^a \in \{f_1^a \ldots f_n^a\}} K_{ext}(f^a)}{K_{ext}(m)} \tag{4}$$

Rationale: The MII was discussed extensively in at the introduction to Section 3.4.2.

- **API function usage index ($APIU$)**

$$APIU(m) = \frac{\sum_{j=1}^{k} n_j}{n * k}; 0 if n = 0 \tag{5}$$

Rationale: $APIU$ gives an indication regarding the maturity and degree to which the module has been vetted. Modules providing a large collections of unused cryptography routines would likely have low $APIU$. We expect this metric to inversely correlate with vulnerabilities. For example, consider this related to a module providing cryptographic routines. A change in a project that introduces a call into such a module to use a previously unused cryptography routine would be treading new territory; the newly called routine is assumed to not have the operational time represented by other public functions in the module.

3.4.3 Change and Churn Metrics

Unless otherwise stated, when referring generically to churn, we mean the sum total of additions, deletions, and modifications. That is: $Churn(E) = NumItem_{additions} + NumItem_{deletions} + NumItem_{modifications}$. As an example: $Churn(FILE) = NumLines_{Added} + NumLines_{deleted} + NumLines_{Modified}$.

NumberOfChanges and *ChurnTotal* are described as follows:

- *NumberOfChanges:* Number of releases, or commits in which the entity E, (e.g., the file or module), has changed.
  Rationale: There is support from fault prediction studies showing that the more a component changes, the more likely it is to have defects; we evaluate this notion in the context of vulnerability prediction. Access to a VCS provides a fine granularity since each check-in (i.e., "commit") can be counted. When only version archives are available, the count is limited to detected changes across snapshots.
- *ChurnTotal:* Total churn over the lifetime of an entity E, i.e., churn(E).
  Rationale: We assume that the more has changed, the higher the likelihood defects will be introduced.

3.4.4 Change Burst Metrics

Change Bursts represent a family of metrics, characterized by change bursts, $CB(\mathcal{G}, \mathcal{B})$, with gap, $\mathcal{G}$, and burst $\mathcal{B}$ parameters (See Section 2.1). We may refer to the change burst as simply CB, without the parameters ($\mathcal{G}$, $\mathcal{B}$, in cases where we are discussing concepts and the particular parameter settings are unimportant. The notation $bursts(E)$ corresponds to the bursts for element E. That is, element E has a change history $E = \langle e_1, e_2, \ldots \rangle$ and its bursts are $bursts(E) = \langle B_1, B_2, \ldots \rangle$. The following are burst metrics presented by Nagappan et al. [28], adapted slightly for our context:

- *NumberOfConsecutiveChanges* – Number of consecutive builds, versions, or releases for a given gap size, $\mathcal{G}$. This is $|bursts(E)|$, with $\mathcal{B} = 0$.
  Rationale: Accounts for all consecutive changes for a given gap size.

- *NumberOfChangeBursts* – Number of change bursts corresponding to a particular gap, $\mathcal{G}$, and burst size, $\mathcal{B}$. The cardinality of CB, or $|bursts(E)|$.
  Rationale: Burst patterns are indicative of risky behavior.

- *TotalBurstSize* – Number of changed releases in all change bursts, i.e. $\sum_{B \in bursts(E)} |B|$.
  Rationale: Assuming that change bursts indicate risky activities, a high number of changes during these bursts could be particularly risky.

In the following definitions, let $churn(e_i)$ be the number of lines that were added, deleted, or modified during the changes to the entity e_i. By extension, let us also apply churn to sets, as in $churn(E) = \sum e_i \in E churn(e_i)$ [28].

- *TotalChurnInBurst* – Total churn in all change bursts, i.e., churn(bursts(C)).
  Rationale: The amount of change involved may be particularly predictive.

- *MaxChurnInBurst* – Across all bursts, this is the maximum churn, $max\{|churn(B)||B \in bursts(E)\}$.
  Rationale: Looking for extremes across change bursts.

3.4.5 Entropy Based, Historical complexity metrics

A Family of metrics, presented by Hassan [13], denoted as HCM, that utilize the entropy of files changed over a change period. Note that the change period can be established as a burst with a gap and burst size. Higher values in these metrics reflect more scattered and widespread changes. Lower values of HCM correspond to smaller, more isolated changes to a few files. We expect that more widespread and scattered changes, characterized by larger HCM values, will be more likely to introduce vulnerabilities.

We evaluate Hassan's entropy based historical complexity metrics, HCM, for vulnerability prediction. We feel that entropy based metrics may be especially well suited for vulnerability prediction since:

- Entropy based historical change metrics outperformed prior faults as a predictor of future faults–in experiments to date,
- The level of entropy will increase as changes become more scattered across files and modules,
- The change period can be determined automatically using change bursts, and
- the presence of said change bursts may themselves be indicative of a large development push or refactoring effort where vulnerabilities may likely be introduced.

3.4.6 Code Metrics

The code metrics included here are less extensive than other studies. We include some of the better performing coupling and complexity metrics for comparison with other studies. Metrics are also selected based on our conjectures regarding how these metrics might be compared with our architectural modularity and maintainability metrics. For example, we include the Henry Kafura (HK) metric because it mirrors the information flow concept also embodied in Anan et al.'s [4] module maintainability index, MMI.

In general, we are interested in metrics that have the potential to characterize information flow through entities such as functions, files, and modules, as well as complexity metrics that might provide barriers to human comprehension. We reason that security vulnerabilities may manifest as the combination of information flow and difficulties in comprehension, such as defects introduced unknowingly by developers because parameter passing or variable accesses from one function (or module) to another is dubious or suspect. The code metrics that we are interested in are presented in Section 5 along with our experimental results.

4 Repository Mining Approach

In this section, we examine the problem more deeply, providing details relevant to our approach. After a brief overview is provided, we provide details of our investigation and identify key constructs, equations, tools, and resources.

4.1 Investigation Overview

Our study subjects are selected due to their significance and relevance, source code accessibility, and popularity among

ESE researchers. Our study subjects are large, widely used, real world software projects. They are Mozilla FireFox, Apache HTTP Server and MySQL.

Using data from a project's issue tracking system ITS (e.g., Bugzilla) and software version control system VCS (e.g., git), we trace bug reports back to particular file revisions and software releases. The bug reports, along with affected files, enable us to build a training and evaluation database (TEDB) where each record consists of a file (or other entity of interest), various metrics, and a classification label of vulnerable($Vuln^+$) is applied to a file if its code underwent modification as the result of patching a vulnerability noted in a published advisory.

Several extraction tools exist for examining software repositories as well as analyzing the code to automatically compute traditional metrics. Architectural recovery of dependency graphs and calculation of various metrics is known as fact extraction. Related studies, as well as our own research have guided our selection of SciTools Understand and Anaconda Python.

SciTools Understand is a commercial tool that provides architectural recovery of call graph information and several complexity metrics [1]. In addition to being used widely in the ESE literature, Understand provides Python APIs to access the fact database it generates for a project [3], [2].

4.2 Data Mining Activities

This section provides a more detailed examination of our data mining activities. Aside from representing an important (and non-trivial) part of this work, we provide this discussion for two reasons. First, we wish to inform other ESE and MSR researchers seeking additional information and insight into our methods. Secondly, these activities are important because our analysis and results depend on the data available for each software project studied. The following steps highlight the mining activities carried out for this research:

1) Mine NIST NVD [31] for security advisories applicable to the project
2) Analyze mined advisory data to determine versions of study (i.e., the version range)
3) Mine the project VCS to extract facts related to each version
4) Determine affected files (i.e., our vulnerable files)

Advisory data in this work consists primarily of CVE [24] entries extracted from the NVD for each project. CVSS [23] scores related to specific advisories are extracted from NVD records. For each project, we limit our study to a viable version range.

We use different methods to determine the set of files modified to fix a vulnerability. This set of files modified is used for marking our training data with the vulnerable classification ($Vuln^+$). For convenience, we refer to this set as *the vulnerable file set*. A common pre-requisite for each method is Web scraping (and/or crawling) of the project's release notes and security advisories. We developed spiders using the ScraPy Web crawling package. The method used to arrive at the vulnerable file set depends on the information available in the release notes (or security advisory Web-page) of each project and information available in the project's ITS and VCS:

- **Download from ITS.** This method is used when a bug ID is provided in the release notes and patches are available from the project's ITS. The ITS is indexed with the bug id (and further scraped) to determine the vulnerable file set.
- **Extract from VCS.** This method is used when patches are not accessible from the project's ITS and/or when a specific bug id is not provided in the release notes.

For projects where the vulnerable file set can be downloaded from the ITS, our spiders obtain resolution and status terms, as well as perform automated text classification to determine which of the attached patches contain the vulnerable file set.

For projects where the vulnerable file set cannot be determined from the ITS, we use one of the following techniques to determine the vulnerable file set by searching the VCS:

- When a bug ID can be obtained, automatically search the VCS change logs (commit history) for references to the bug (i.e., in the commit message), or
- When a bug id cannot be obtained, query associated CVE summary for keyword stems that are subsequently used to search the VCS change logs in a semi-automated fashion.

4.3 Discussion Regarding Training Set Construction

Note that the construction of a suitable training set from the mined information sources is non-trivial. Real world data sources contain information of varying maturity, completeness, and accuracy. The variance and inconsistency in vulnerability data presents multiple challenges to building a viable data set for training supervised machine learning algorithms. Figure 1 shows a Venn diagram to visualize the required vulnerability information. For each software application, we require information about individual, *publicly disclosed*, security vulnerabilities. Such information includes the vulnerable file set, affected versions, and a severity score (from CVE entry).

Because we train our models to differentiate vulnerable from non-vulnerable files and modules within each version of a software, it is necessary to accurately associate a publicly disclosed vulnerability with the vulnerable file set in each version. CVE entries from NVD contain a list versions of the software that are affected by the specific vulnerability disclosed in each CVE. However, in the case of the Apache HTTP Server, we found that in more than one instance, the release notes (aka security report) for version 2.2 [5] described a set of affected versions that were not otherwise found in the corresponding CVE entry. In the case of Apache HTTP Server, we chose to use the affected version information from the Apache-specific security advisory pages for version 2.2. In the cases of Mozilla Firefox and MySQL, the security advisories and release notes only listed the fix version, as opposed to a detailed list enumerating each individual version. We performed a study of the affected versions listed in CVE entries corresponding to Firefox and MySQL and found that in our version ranges of interest (i.e., the inspection interval $\mathcal{I}$) for each application, all versions

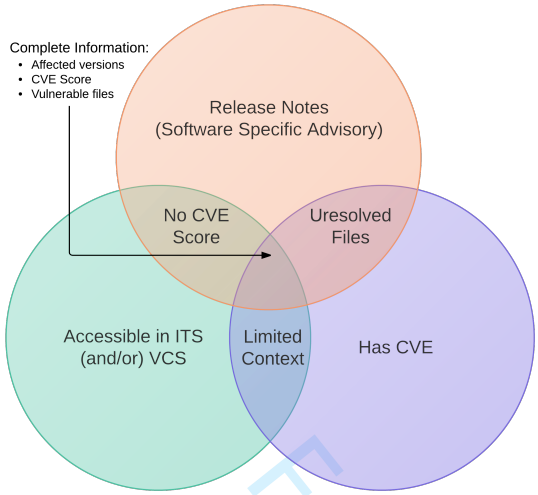

Figure 1. Required vulnerability information

prior to the listed fix version were affected. When building our training data for MySQL and Firefox, we therefore assumed that a vulnerable file was also vulnerable in previous versions, of course requiring that the file actually existed in the earlier version. We will refer to this key assumption as the *Previously Vulnerable Assumption*. Below, we describe additional detail around our investigation of the affected versions listed in CVE data to validate this assumption.

In the course of undertaking the aforementioned investigation related to affected versions listed in the CVE data from the NIST NVD, we also noticed a change in the completeness of the affected version list in said CVE entries. For example, Firefox CVE entries corresponding to version 36 and earlier (from years 2011 to 2015) often enumerate all affected versions, listing each one with its own `<vuln:product>` tag. However, CVE entries after and including CVE-2015-2706 (for years 2015 and 2016) only list one affected version, (not counting so called ESR, or extended service release versions).

As mentioned previously, our additional validation effort led us to the *Previously Vulnerable Assumption*, and a new view of training set construction which is discussed at length in later sections.

Previously Vulnerable Assumption. In the absence of trustworthy data related to specific versions of a software that are impacted by a vulnerability fixed in version N, *we assume that all prior versions, over the inspection interval $\mathcal{I}$, up to and including version $N-1$ are also impacted*. Impacted versions are thus expressed as V_i, where:

- V_i denotes a vulnerable version number for $i = \{i_1, i_2, \ldots, N-1\}$
- $\mathcal{I}$ denotes the inspection interval, $\mathcal{I} = \{R_{Min}, \ldots, R_{Max}\}$,
- R_{Min} and R_{Max} correspond to the minimum and maximum release versions studied respectively.
- Note that $i \in \mathcal{I}$, therefore $\forall i, i \geq R_{Min}$ and $i < N \leq R_{Max}$

Note that the *Previously Vulnerable Assumption* is contingent on validation over interval $\mathcal{I}$. Using Firefox as an example, interval $\mathcal{I}$ corresponds to the inclusive range $6..49$.

Table 2
Marking technique for vulnerable files

Time	Version → 1	2	3	4	5
1	`foo.c` not in Vuln		FixedIn,0	0	0
2			0	0	0
3	1	1	0	0	0
4	1	1	0	0	0
5	2	2	1	1	FixedIn,0
6	2	2	1	1	0
7	2	2	1	1	0

Our additional goal of relating the severity of vulnerabilities to various features of our training data requires the use of a standardized measure of severity. At the outset of this work, we decided to use the CVSS score for each vulnerability (i.e., a CVE entry in the NIST NVD) as an indicator of the severity. Two different cross-reference consistency complications related to the use of the CVSS score from the CVE presented itself in practice: unknown CVE identifiers and missing CVE identifiers. We describe each in additional detail below.

Unknown CVE Identifiers. Unknown identifiers result from inconsistent mapping among security advisories, release notes, and ITS entries. This mapping inconsistency presents an issue when attempting to cross-reference a CVE identifier from a mined ITS entry.

Missing CVE Identifiers: Missing identifiers occur when a security vulnerability is noted on a security advisory page (or in release notes), but does not list an associated CVE. That is, the advisory page may list only the ITS entry, but fail to list any associated CVE identifier.

The above complications limit our ability to reliably use CVSS scores in all cases. Because we are able to resolve the ITS entry to the individual affected files, we can classify them as vulnerable, but we are unable to perform correlation analysis related to severity.

4.4 Vulnerable File Marking Approach

The combination of the Previously Vulnerable Assumption and the notion of CVE identifiers attributing a scored severity, leads to a vulnerable entity marking technique. We present a view of this marking pattern in Table 2.

Table 2 depicts a marking for a single vulnerable file in the TEDB. The marking shown above is for an associated CVE. File-to-CVE relationships are stored in the TEDB outside this table. The table is interpreted as follows, for some single file `foo.c`, in iterating through mined vulnerability fix data for over the inspection interval $\mathcal{I}$.

1) At version 1, `foo.c` is not in the vulnerability table; no record or marking; same at version 2
2) At version 3, the table building logic observes mined data indicating a security fix for version 3, and

determines the most recent change to `foo.c`, prior to version 3, occurred in version 1

- The table building logic adds entries for `foo.c` in versions 1 and 2
- The table building logic increments CveCount to 1 in records corresponding to versions 1 and 2, and adds any corresponding CVE relationship the File-to-CVE table

3) At version 4, no mined security fix information is available for `foo.c`
4) At version 5, the steps at version 3 are repeated for the new fix information by applying the Previous Vulnerability Assumption; most recent prior change is still version 1 because the fix at version 3 was for a different vulnerability.

- Table building logic back-propagates the relationship for any newly associated CVE fixed in 5
- The CveCount for prior entries of `foo.c` are incremented, resulting in the pattern depicted in Table 2

Our approach seeks to better understand factors occurring in development of a software product that leads to severe vulnerabilities, as opposed to measuring the development effort applied to the development of the security fixes themselves.

4.4.1 Vulnerability Distributions and Related Signals

One must ensure that the vulnerability table reflects an accurate *vulnerability* distribution at any point in time and *not* the *fix distribution* (e.g., over time when characterized as successive version slices). This is a phenomenon we experienced in our model building process where a sanity check of correlation values revealed an echo of the *fix distribution*, rather than accurately characterizing desired relationships with the *vulnerability distribution*, i.e., within a vulnerability slice. As we discovered, this issue is especially acute for repository mined change metrics counting number of files changed or LOC, as they will directly measure the development activity coinciding with the creation of security fixes for known vulnerabilities, thereby artificially inflating correlation results when evaluating change based metrics for vulnerability prediction. Thinking of change based metrics across version slices leads to interpretation of the overall patterns as vulnerability and fix signals respectively. We can think of this phenomenon as failure to remove the "security fix signal", which forms a vulnerability oracle and has the effect akin to including a label as a feature during training. In other words, including labeled observations (or oracle entries) as features when feeding data to a learning model. A model trained on oracle data for its test set would show 100% precision, but would be completely useless in accurately predicting vulnerabilities in practical real world scenarios.

Assuming a typical scenario where a security fix for version $N+1$ is developed in the prior version N, correlation evaluation of size based repository mined change metrics will show inflated results. Because change metrics measure the interval $N-1$ to N, an observer at version N, making predictions for version $N+1$ from mined data, will have a vulnerability oracle for the fix included with the associated change based metric (measured over the interval $N-1$ to N).

4.5 Module and API Identification

Recall the module metrics MII and APIU are relative to a module and characterize interaction with other modules. In both metrics, there is a concept of identifying a public API for the module. We note that for a large projects, containing millions of lines of code, it is necessary to devise an approximation for what constitutes the public API, since it is unfeasible to search through the entire codebase. Our method relies in part on information from the Understand tool, as well as heuristics of our own design.

As we process functions (or methods) in the codebase using the understand tool, we identify front line functions [11] from the standard C library, according to the list given by Manadhata [22]. We consider such front line functions part of a `STDLIB` meta-module. All front line functions thus identified are considered part of the `STDLIB` public API. Automated identification of the functions in the public API of other modules (that is, modules that are not the `STDLIB` meta-module) are based on the filename, its extension, or information reported by the Understand tool.

4.6 Change Burst Detection

Recall that a change burst, $CB(\mathcal{G},\mathcal{B})$ is parameterized with with gap size $\mathcal{G}$ and burst size $\mathcal{B}$. The gap size, $\mathcal{G}$, is the maximum distance between successive changes, such that those changes are considered within the same burst. The burst size is the minimum number of successive changes required to be considered a burst.

Adjusting the the gap size and burst size enables additional filtering on the related metrics. Increases in burst size decreases across the maximum absolute values for related metrics as the shorter burst sequences are eliminated from consideration. Increases in gap size result in longer burst sequences that in turn yield increasingly larger maximum absolute values.

Our mining software uses the git "commit date" associated with each changeset in order to perform comparisons against a gap size, $\mathcal{G}$, which is specified in days. A phenomenon complicating the construction of the training database is the fact that dates associated with changesets in a VCS do not necessarily follow the sequence in which those changesets are merged into the VCS. This often occurs when as different branches are merged together, or may also result when a changeset is under review for a long period. Additionally, at least under git, it is also possible to manipulate the commit date. We encountered this phenomenon, which we refer to as "a reversal", for both Firefox and MySQL repositories. A reversal occurs when the date associated with a subsequent change $Change_{i+1}$, precedes the current change $Change_i$. That is, we expect the mapping between change sequence and its corresponding date sequence to maintain relative ordering, $Date(Change_i) \preceq Date(Change_{i+1}), i \in \mathcal{I}$. However, a reversal is the violation of this expected ordering.

In the case of Firefox, we attempted to use supplementary information available to correct the date. Because it is not possible to determine a correction in all cases, we log this phenomenon as two new burst measurements we call *CountReversals* and *MaxSeqByDate*. In cases where the mining software encounters no date reversals in the VCS between versions $N-1$ and N, then *MaxSeqByDate* is equivalent to the number of commits (i.e., *CountTouches*) over the interval defined by $N-1$ to N. In cases where a reversal occurs, *MaxSeqByDate* represents the largest contiguous sequence before encountering a date reversal.

5 Experimental Results, and Analysis of Metrics and Vulnerability Predictions

In this section, we enumerate the experimental results for our software case studies. Results within each section are prefaced by an overview of the the project and important characteristics, such as versions studied, project size, and the approximation of the project's true vulnerability density given by our historical training tables (i.e., that define the vulnerable training set).

5.1 Test Harness and Evaluation Approach

For each case study, we use the same experimental framework for evaluation. Our framework for evaluation consists of a training and test harness built with Scikitlearn [32] that provides a ten-fold stratified random split of the data, withholding 33% of the data for test evaluation. The framework is applied on select metrics, iterating the ten-fold stratified split for 10 iterations (e.g. 10 x 10-fold cross-validation), as is a widely accepted practice in vulnerability model evaluation.

Because we are searching for residual vulnerabilities, their true population distribution is unknown by definition. Therefore we are always assuming that any statement about the vulnerability population is an approximation. Moreover, from the samples we have (i.e., the files we've already labeled as vulnerable and neutral) it is evident that the distributions are heavily skewed, with neutral files outnumbering vulnerable files by a factor of roughly ten to one. Shin[37], as well as Chowdhury and Zulkernine [10] note the danger of imbalances in the training data and the danger of overfitting the model to the non-vulnerable classification. Hence, following the approach of of the aforementioned researchers, we use the random stratified sampling technique to preserve the skewed distribution in our datasets, while training across the ten folds prevents our model from overfitting on neutral files (i.e., the majority class).

5.2 Training Labels and Metrics Definitions

This section provides a roadmap for the case studies. We describe the training scores and labels associated with vulnerable files and review the definitions for the metrics analyzed.

Vulnerable and neutral entities (i.e., files and modules) are distinguished based on a binomial column label Vuln. We set Vuln $\bar{1}$ for vulnerable entities. Neutral entities, or entities where no vulnerability has yet been found are labeled with Vuln = 0.

Table 3
Definitions for classification labels and expected scores

Metric	Definition
Vuln	The binomial label indicating whether or not the entity is vulnerable or neutral. For modules, this is set to 1 if the module contains at least one vulnerable file.
RawCount	The ordinal count of security issues (most commonly counted as ITS entries) associated with a file in version $N+1$. For modules, this is the sum of the RawCount across all files within the module.
CveCount	The ordinal count of CVE entries associated with a file in version N. For modules, this is the sum of the CveCount for all files in the module.
AvgScore	TotalScore ÷ CveCount. Simply, the average score; refer to TotalScore and CveCount defined in other rows. AvgScore is equal to the CVSS Base score metric in cases where CveCount is 1, and is 0 when CveCount is 0.
MaxScore	The maximum CVSS score, from among associated CVEs (floating point). For modules, this is the maximum across all files contained in the module.
TotalScore	The sum of CVSS scores from associated CVEs (floating point). For modules, this is the sum of the TotalScore across all files contained in the module.

5.2.1 Counting Vulnerabilities and Quantifying Severity

In addition to the binomial Vuln label, we can also count how many security related issues that the file was associated with from the ITS, VCS, or determined vulnerable using some other reference source. Other reference sources in our context are primarily release notes or CVE entries. The number of security related issues thus determined is stored as RawCount. The RawCount for an entity in version N, is the number of security related issues fixed in the *next* version, $N+1$, and represents the likelihood that the entity is vulnerability prone.

The CveCount is an ordinal value indicating the number of unique CVE entries with which an entity is associated, irrespective of version. For example, if a file (`foo.c`) is associated with three different CVE entries in version N, then CveCount = 3.

Note that RawCount indicates the number of security issues with which the entity was associated, without requiring the entity to be resolved (cross-referenced) to a CVE. For those entities which we were able to cross-reference to a CVE entry, we define additional CVE-based counts and scores. These additional CVE-based scores are defined in Table 3, along with Vuln and RawCount.

RawCount and CveCount may differ for a vulnerable entity, especially when that entity is involved in more than one vulnerability. For our purposes, RawCount helps to identify vulnerable entities we learned about through our mining activities, but were not resolved to a particular CVE entry. Entities with a RawCount $>$ CveCount, and where CveCount $= 0$, represent entities that we have learned are vulnerable, but for which severity cannot also be ascribed.

Throughout this work, we use CveCount interchangeably with "vulnerability count". By standardizing on CveCount, we build on the already well-formed definition for CVE entries (and their use in industry) as synonymous with identified security vulnerabilities in software. Finally, our decision to standardize on CveCount additionally facilitates

Table 4
Definitions for file complexity metrics at version N

Metric	Definition
CountDeclFunction	Number of functions.
CountLineCode	Number of lines containing source code. [LOC]
CountLineCodeDecl	Number of lines containing declarative source code.
CountLinePreprocessor	Number of preprocessor lines.
CountStmt	Number of statements.
CountStmtDecl	Number of declarative statements.
CountStmtExe	Number of executable statements.
MaxCyclomaticModified	Maximum modified cyclomatic complexity of nested functions or methods.
MaxCyclomaticStrict	Maximum strict cyclomatic complexity of nested functions or methods.
MaxNesting	Maximum nesting level of control constructs.
RatioCommentToCode	Ratio of comment lines to code lines.
SumCyclomatic	Sum of cyclomatic complexity of all nested functions or methods. [aka WMC]
SumEssential	Sum of essential complexity of all nested functions or methods.

our use of CVSS scores that are exclusively associated with CVE entries as *severity quantification metrics*. Note that when CveCount = 1, the average, max, and total are all equal. Table 3 summarizes the quantification metrics.

5.2.2 Metrics Extraction and Calculation

Complexity Metrics . All file level complexity metrics are extracted relative to the `git` commit revision corresponding to our collected version N. As such, all metrics thus mined form a set of observations at N for predicting $N + 1$.

We used SciTools Understand tool to extract file level complexity metrics. The understand tool exports several metrics, detailed descriptions of which can be found at [1]. We list the top few metrics that were repeatedly selected by the ANOVA [25] filter in our classification experiments (i.e., primarily using the logit classifier), in addition to a few metrics that are widely recognized across defect and vulnerability literature.

File Level Churn Metrics. A review of file level churn metric definitions is provided in Table 5 to facilitate quick reference in the following sections. Note that all churn metrics begin collection by walking the reverse topographically sorted revision history provided by the `git log` command from the revision corresponding to the interval $N - 1$ to N. The observation point for prediction is still at version N. As such, all metrics thus mined form a set of observations at N for predicting $N + 1$.

Module Level Burst and Architectural Metrics. Module level burst metrics are defined in Table 6 to facilitate quick reference in the following sections. Note that all churn metrics begin collection by walking the reverse topographically sorted revision history provided by the `git log` command from the revision corresponding to the interval $N - 1$ to N. The observation point for prediction is still at version N. As such, all metrics thus mined form a set of observations at N for predicting $N + 1$.

Table 7 provides an overview of our module metric definitions.

Table 5
Definitions for file churn metrics at version N

Metric	Definition
CountTouches	The number of changsets (e.g., VCS commits) touching the file since $N - 1$ [aka. NumChanges]
CountTouchesP1	CountTouches for this file from $N - 2$
CountTouchesP2	CountTouches for this file from $N - 3$
HCPF	The historical complexity period factor, an entropy based metric indicating the file's degree of contribution to entropy over the interval $N - 1$ to N
LinesAdded	The number of lines added since $N - 1$
LinesModified	The number of lines modified since $N-1$
LinesDeleted	The number of lines deleted since $N - 1$
TotalTouches	The cumulative sum of CountTouches at N, from all prior intervals

Table 6
Definitions for module change burst metrics observed at version N

Metric	Definition
CountTouches	The number of changsets (e.g., VCS commits) touching the module path since $N - 1$ [aka. NumberOfConsecutiveChanges] 3.4.4].
CountReversals	The number of commit date reversals encountered when scanning the (e.g., VCS commits) changes within the module since $N - 1$.
MaxSeqByDate	Largest sequence of consecutive changes by date when CountReversals ¿ 0, otherwise this metric is equivalent to CountTouches.
CountBursts	The total count of bursts in the module since $N - 1$ [aka. NumberOfChangeBursts].
CountFilesChanged	The total count of files changed in the module since $N - 1$.
NetAddedInBurst	A churn metric characterizing the net positive number of lines added since $N - 1$. This is LinesAdded - LinesDeleted, when LinesDeleted $\leq$ LinesAdded, otherwise 0. This quantity is LinesAdded when LinesDeleted = 0.
TotalChurnInBurst	The number of lines modified, as the sum of lines added and lines deleted since $N - 1$ (as reported by git diff).

Table 7
Definitions for module metrics observed at version N

Metric	Definition
APIU	The API function usage index defined in Section 3.4.2; appears in graphs in lowercase as apiu.
k_ext_fa	Number of calls from external modules to the public API exposed by a given module; Refer to Section 3.4.2
k_ext_m	Number of external calls from external modules; Refer to Section 3.4.2.
MII	Module Interaction Index defined in Section 3.4.2; appears in generated graphs in lowercase mii.
N_API	The number of public API functions or methods exported by a module, discussed in Section 4.5

Table 8
Firefox project statistics

Version	LOC	Count Functions	Count Methods	Count Files
6	2,584,376	145,384	82,529	11,329
28	4,473,921	263,480	105,902	20,050
49	5,818,230	355,352	116,753	25,592

5.3 Case Study 1: Mozilla Firefox Web Browser

We studied Mozilla Firefox versions 6 to 49, representing a time span of approximately five years from August, 2011 to August, 2016. Table 8 provides an overview of size measurements across the first, middle, and last versions studied. The measurements shown here were collected from the SciTools Understand tool [1]. The files processed include only source and header files (e.g, .c, .cpp, .h, etc.), and additionally exclude documentation and test folders in the project's working tree.

As depicted in Table 8, there are significant differences in size across the versions studied. Version 28, representing the middle version, is nearly double (1.7 times) the size of version 6 when measured in lines of code (LOC). The last version, 49, has more than double (2.25 times) the LOC than in version 6, and is 1.3 times larger than version 28. Given such large differences, we would expect that Firefox version 49 is a much different piece of software than it was at version 6.

Figure 2 is a visualization the number of vulnerabilities, binned to the nearest whole-numbered (i.e., major) version of Firefox, over our inspection interval $\mathcal{I} = \{6, \ldots, 49\}$. In this figure, VulnCount represents the number of unique CVE entries plotted against the versions to which they apply. As shown, the figure represents view of the historical vulnerability density in Firefox over $\mathcal{I}$. The figure visually depicts the pattern we noticed in CVE data for Firefox that supports the *Previously Vulnerable Assumption*.

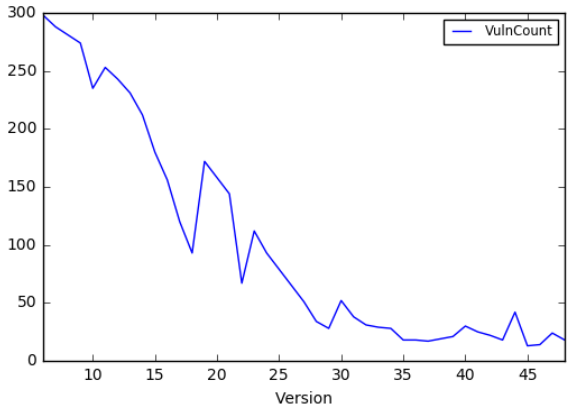

Figure 2. Historical vulnerability density for Firefox versions 6..49

The list of applicable affected versions is derived directly from the CVE entries themselves. Firefox related CVE entries in the NIST NVD, for versions 6 to 36, specifically enumerate all prior versions. The pattern, as depicted by Figure 2 shows the effect of a retroactive, linearly increasing offset that "swells" the VulnCount of earlier versions; colloquially speaking, we feel the catch phrase a "rising tide lifts all boats" fits well here. Early visualization of this data for Firefox informed our vulnerability table construction technique discussed in Section 4.4.

Our overall model building approach and corresponding view of fix and vulnerability signals is reinforced when we plot the number of *fixed* issues from the ITS for each release version (RLS) against counts of Firefox CVE entries per RLS version. This plot is shown in Figure 3, and shows the impact of known security fixes (the blue line) against the the count of unique CVE entries (the green line) affecting each version of Firefox.

Note that Figure 3 simply plots two measured values present in our mined data. We can see that the tip of feature A, between versions 16 to 18, leads its appearance in the CVE data around slice 18.

Because the overall density curve is standardized by the total sum of all CVE counts for all slices, the density signal flattens out and shows the effects of the spike near version 45 from the fix signal much less prominently. This view supports the Previously Vulnerable Assumption because the high counts from the spike at 45 feed backward into prior slices, which works to reduce its impact on the density curve at version 45.

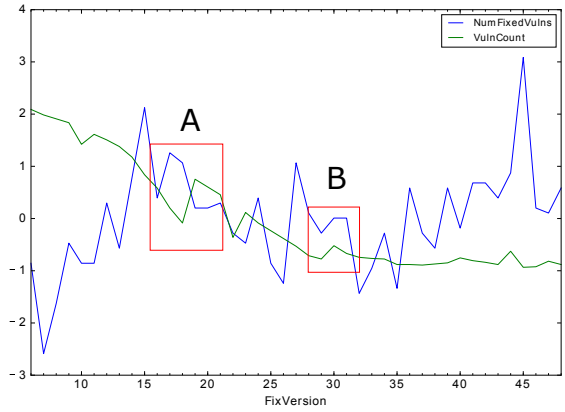

Figure 3. Similar trend observations A and B

5.3.1 File Level Complexity Metrics

Table 9 shows the average Spearman correlations of file complexity metrics with severity quantification metrics for Firefox versions 7 to 26. All correlations have $p < 0.05$.

5.3.2 File Level Churn Metrics

Table 10 shows the average Spearman correlations of file churn metrics with severity quantification metrics for Firefox versions 7 to 26.

5.3.3 Module Level Burst and Architectural Metrics

Table 11 shows the average Spearman correlations of change burst churn metrics CB(2,2) with severity quantification metrics for Firefox versions 7 to 26. Table 12 shows the average Spearman correlations of module metrics with severity quantification metrics for Firefox versions 7 to 26.

Table 9
Average Spearman ρ for file complexity metrics; Firefox 7..26

Metric	CveCount	AvgScore	MaxScore	TotalScore
AltAvgLineCode	0.127	0.034	0.018	0.064
AltCountLineCode	0.287	0.059	0.063	0.154
AvgCyclomatic	0.111	0.007	0.041	0.073
AvgCyclomaticModified	0.116	0.009	0.042	0.075
AvgCyclomaticStrict	0.114	0.012	0.037	0.072
AvgEssential	0.125	0.007	0.049	0.083
CountDeclFunction	0.260	0.076	0.041	0.118
CountLineCode	0.278	0.069	0.051	0.141
CountLineCodeDecl	0.286	0.049	0.076	0.162
CountLineCodeExe	0.254	0.064	0.048	0.127
CountLineComment	0.293	0.075	0.048	0.148
CountLinePreprocessor	0.260	0.111	0.004	0.093
CountStmt	0.278	0.064	0.054	0.144
CountStmtDecl	0.279	0.050	0.071	0.155
CountStmtExe	0.254	0.071	0.040	0.121
MaxCyclomatic	0.191	0.053	0.029	0.091
MaxCyclomaticModified	0.197	0.057	0.029	0.093
MaxCyclomaticStrict	0.193	0.058	0.027	0.090
MaxEssential	0.195	0.058	0.031	0.091
MaxNesting	0.190	0.054	0.029	0.089
RatioCommentToCode	0.001	0.017	0.020	0.004
SumCyclomatic	0.256	0.063	0.051	0.128
SumCyclomaticModified	0.260	0.067	0.049	0.128
SumCyclomaticStrict	0.255	0.065	0.049	0.126
SumEssential	0.268	0.066	0.055	0.134

Table 10
Average Spearman ρ for file churn metrics; Firefox 7..26

Metric	Cve Count	Avg Score	Max Score	Total Score
CountTouches	0.317	0.006	0.139	0.219
CountTouchesP1	0.301	0.010	0.136	0.210
CountTouchesP2	0.284	0.016	0.134	0.202
HCPF	0.302	0.011	0.117	0.197
LinesAdded	0.305	0.007	0.118	0.202
LinesModified	0.299	0.022	0.106	0.186
LinesDeleted	0.307	0.001	0.125	0.208
TotalTouches	0.273	0.055	0.070	0.138

Table 11
Average Spearman ρ for change burst metrics CB(2,2); Firefox 7..26

Metric	Cve Count	Avg Score	Max Score	Total Score
CountTouches	0.387	0.071	0.197	0.368
CountReversals	0.361	0.059	0.192	0.344
MaxSeqByDate	0.335	0.082	0.175	0.324
CountBursts	0.373	0.07	0.194	0.356
CountFilesChanged	0.403	0.11	0.231	0.395
NetAddedInBurst	0.396	0.1	0.236	0.39
TotalChurnInBurst	0.397	0.099	0.233	0.389

Table 12
Average Spearman ρ for module metrics; Firefox 7..26

Module	Cve Count	Avg Score	Max Score	Total Score
APIU	0.272	0.298	0.036	0.16
k_ext_fa	.096	0.32	0.172	0.173
k_ext_m	0.167	0.206	0.163	0.22
MII	0.105	0.18	0.025	0.061
N_API	0.38	0.256	0.331	0.427

5.3.4 *Firefox Prediction Experiments*

In the limited number of experiments we were able to perform with churn and traditional complexity metrics, we noted a marked difference in prediction results, despite the churn metrics p value from the Welch test exceeding 0.05.

5.3.5 *Firefox experiment comparing complexity and churn*

Table 13 shows prediction performance of our prediction test harness selecting the best three complexity features and performing vulnerability predictions for Firefox versions 7 to 26. The three best selected features were, in order: SumCyclomaticModified, SumEssential, and CountDeclFunction.

Table 14 shows prediction performance of our prediction test harness selecting the best three file churn features and performing vulnerability predictions for Firefox versions 7 to 26. The features selected were, in order: CountTouches, CountTouchesP1, and TotalTouches.

Comparing predictions from Table 13 and Table 14, we note generally higher precision values and tighter variance in prediction accuracy within 2σ, or the 95% confidence interval.

5.4 Case Study 2: Apache Web Server

We studied Apache HTTP Server versions 2.2.0 to 2.2.29, representing a time span of over eight and a half years from December, 2005 to August, 2014. We studied the micro (patch) versions of Apache HTTP Server 2.2.x. Table 15 shows the adoption of the 2.2 version as of this writing, thus reinforcing our decision to study 2.2.x.

Table 16 provides an overview of size measurements across the first, middle, and last versions studied. The measurements shown here were collected from the SciTools Understand tool and exclude documentation and test folders

Table 13
Mean precision (M_{prec}) performance using complexity metrics; $K = 3$; Firefox versions 7..26

Version	M_{prec}	+/– 95%
7	0.60	0.17
8	0.48	0.51
9	0.47	0.41
10	0.57	0.41
11	0.62	0.10
12	0.61	0.15
13	0.62	0.11
14	0.68	0.11
15	0.58	0.25
16	0.62	0.16
17	0.51	0.24
18	0.55	0.28
19	0.54	0.13
20	0.61	0.16
21	0.59	0.12
22	0.58	0.23
23	0.44	0.39
24	0.53	0.21
25	0.50	0.25
26	0.48	0.19

Table 14
Mean precision (M_{prec}) performance using file churn metrics; $K = 3$; Firefox versions 7..26

Version	M_{prec}	+/– 95%
7	0.69	0.13
8	0.69	0.10
9	0.67	0.11
10	0.71	0.12
11	0.71	0.08
12	0.68	0.09
13	0.73	0.10
14	0.73	0.11
15	0.69	0.09
16	0.70	0.12
17	0.70	0.11
18	0.70	0.09
19	0.66	0.10
20	0.67	0.08
21	0.67	0.12
22	0.65	0.08
23	0.66	0.10
24	0.63	0.12
25	0.59	0.09
26	0.65	0.11

Table 15
Apache adoption by version [21]

Number of Websites	Apache Version
22,970,250	2.2
8,098,197	2.4
328,257	2.0
307,879	1.3

Table 16
Apache HTTP Server project statistics

Version	LOC	CountFunc-tions	CountFiles
2.2.0	113,138	2,799	305
2.2.15	118,183	2,898	309
2.2.29	120,950	2,977	323

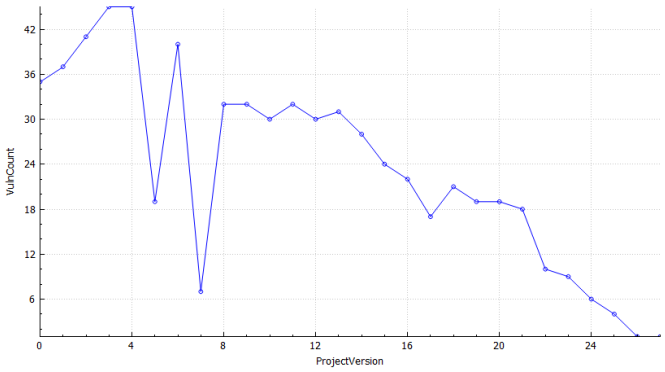

Figure 4. Historical vulnerability density in Apache HTTP versions 2.2.x, 0..29

in the project's source code tree. Note that we've excluded the count of methods from this table because Apache HTTP is implemented exclusively in the C programming language (i.e., contains functions only).

As shown by Table 16, the various sizes are relatively stable across the versions studied. There were 18 files added between version 2.0.0 and 2.2.29, or a 6% increase overall. Likewise, there is an approximate 7% increase in the number of lines between versions 2.2.0 and 2.2.9.

5.4.1 File Level Complexity Metrics

Table 17 shows the average Spearman correlations of file complexity metrics with severity quantification metrics for Apache HTTP versions 2.2.0..29. All correlations have $p < 0.05$.

5.4.2 File Level Churn Metrics

Table 18 shows the average Spearman correlations of file churn metrics with severity quantification metrics for Apache HTTP versions 2.2.x..29

5.4.3 Module Level Burst and Architectural Metrics

Table 19 shows the average Spearman correlations of change burst churn metrics CB(2,2) with severity quantification metrics for Apache HTTP versions 2.2.x..29. We note that CountBursts shows relatively high correlation with AvgScore, which is one of the more conservative estimates of overall severity, since it is averaged over the individuals within the module.

Module metrics for Apache are withheld. Apache HTTP uses many C preprocessor macros and additional add on packages in order to build the software. Due to suspect correlations of 0.5, we posit that conditionally compiled code regions were double-counted. We describe additional detail related to the pitfalls when calculating module metrics that result from the lack of precision in our automated methods in Section 4.5.

Table 17
Average Spearman ρ for file complexity metrics; Apache HTTP 2.2.0..29

Metric	CveCount	AvgScore	MaxScore	TotalScore
AltAvgLineCode	0.036	0.041	0.135	0.079
AltCountLineCode	0.037	0.116	0.144	0.086
AvgCyclomatic	0.011	0.025	0.06	0.008
AvgCyclomaticModified	0.028	0.034	0.06	0.015
AvgCyclomaticStrict	0.01	0.005	0.075	0.003
AvgEssential	0.075	0.092	0.018	0.097
CountDeclFunction	0.043	0.13	0.173	0.097
CountLineCode	0.04	0.152	0.167	0.108
CountLineCodeDecl	0.007	0.05	0.099	0.026
CountLineCodeExe	0.034	0.157	0.158	0.107
CountLineComment	0.133	0.053	0.052	0.082
CountLinePreprocessor	0.017	0.143	0.072	0.082
CountStmt	0.016	0.14	0.156	0.082
CountStmtDecl	0.007	0.036	0.087	0
CountStmtExe	0.041	0.151	0.153	0.105
MaxCyclomatic	0.144	0.129	0.065	0.051
MaxCyclomaticModified	0.118	0.115	0.059	0.042
MaxCyclomaticStrict	0.085	0.141	0.113	0.012
MaxEssential	0.032	0.182	0.11	0.106
MaxNesting	0.028	0.197	0.173	0.147
RatioCommentToCode	0.113	0.067	0.084	0.119
SumCyclomatic	0.056	0.172	0.188	0.133
SumCyclomaticModified	0.05	0.162	0.18	0.127
SumCyclomaticStrict	0.05	0.158	0.173	0.118
SumEssential	0.039	0.171	0.159	0.11

Table 18
Average Spearman ρ for file churn metrics; Apache HTTP versions 2.2.x..29

Metric	Cve Count	Avg Score	Max Score	Total Score
CountTouches	0.185	0.134	0.065	0.044
CountTouchesP1	0.168	0.109	0.061	0.050
CountTouchesP2	0.168	0.108	0.064	0.052
HCPF	0.175	0.083	0.007	0.069
LinesAdded	0.180	0.091	0.020	0.073
LinesModified	0.166	0.061	0.010	0.076
LinesDeleted	0.138	0.087	0.014	0.061
TotalTouches	0.040	0.156	0.134	0.083

Table 19
Average Spearman ρ correlations for burst metrics; Apache HTTP versions 2.2.x..29.

Metric	Cve Count	Avg Score	Max Score	Total Score
CountTouches	0.378	0.352	0.189	0.315
MaxSeqByDate	0.364	0.332	0.189	0.315
CountBursts	0.277	0.252	0.143	0.249
CountFilesChanged	0.275	0.251	0.158	0.247
NetAddedInBurst	0.252	0.218	0.149	0.244
TotalChurnInBurst	0.259	0.224	0.152	0.25

5.4.4 Discussion

An outstanding result of our experiments with Apache HTTP is the high correlation between the burst metrics and our severity quantification metrics (i.e., AvgScore, MaxScore, and TotalScore). This result is especially interesting because the correlation is assessed on a subset of modules already known to be vulnerable (i.e., Vuln = 1). The correlation values we are presenting in this particular discussion *do not* reflect the respective metrics' *classification* ability, but are presented as a *relative quantification of severity*. In particular, CountFilesChanged and TotalChurnInBurst show rank order correlation values as high as 0.90 with our aforementioned severity quantification metrics.

Initial prediction experiments with using complexity features for vulnerability prediction in Apache HTTP showed mean precision consistently below 0.40, with large accuracy values (> 0.30) required to achieve the 95% confidence interval. This suggests a prediction model built with complexity features alone would perform poorly for predicting vulnerabilities in Apache HTTP.

Noting the high correlation between the number of files changed within a burst, and a known vulnerable module, we added an additional score feature used to train our prediction model. When training the model, we cross-reference the file to the module of which it is a member. We then query the TEDB for the burst features associated with that module, extract CountFilesChanged, extending our feature set. Over 100 iterations of a ten-by-ten cross validation, our data show that this change results in improved mean precision, while also narrowing the standard deviation across samples of predicted results in versions 3 through 7. Narrowing the standard deviation improves our 95% confidence interval around the mean precision. Table 20 shows the comparison between mean precision evaluated with 10x10 cross fold evaluation with an ANOVA selection filter set for the top five features. Group B (MP_B and Accuracy B) reflects the module metric enhancement.

The result is compelling because the single addition of CountFilesChanged is a simple change, yet results in noticeable prediction performance improvement. One can envision much more elaborate and well designed scoring functions that make better use of our severity quantification measures during training. Moreover, due to the high correlation between CountFilesChanged and severity, as quantified by the sum of CVSS scores from the respective module, it

Table 20
Comparison of mean precision and its accuracy over early Apache versions; 2.2.3..2.2.10.

Version	MP_A	Accuracy A	MP_B	Accuracy B
3	0.57	0.35	0.59	0.25
4	0.55	0.46	0.60	0.37
5	0.58	0.26	0.61	0.21
6	0.54	0.46	0.59	0.37
7	0.46	0.31	0.53	0.26
8	0.55	0.25	0.49	0.28
9	0.55	0.23	0.49	0.35
10	0.38	0.18	0.58	0.25

Table 21
MySQL Database project statistics

Version	LOC	Count Functions	Count Methods	Count Files
5.5.0	741,042	27,292	17,510	2,015
5.5.28	670,181	28,196	17,480	2,058
5.5.54	673,662	28,293	17,509	2,061

logically follows that correctly predicted results will be of greater significance, as argued throughout this work.

5.5 Case Study 3: MySQL Database Server

We studied Oracle MySQL Database Server versions 5.5.0 to 5.5.54, representing a time span of approximately seven years from December, 2009 to November, 2016. Table 21 provides an overview of size measurements across the first, middle, and last versions studied. The measurements shown here were collected from the SciTools Understand tool and exclude documentation and test folders in the project's source code tree.

Due to differences in the way in which the vulnerability table was built from the VCS mined entries for MySQL, we discovered the results produced to be inaccurate upon analysis. Final results are withheld for a future addendum.

5.6 Discussion

We provide evidence here that metrics better reflecting a software project's inherent architectural properties, and that also correspond with earlier lifecycle development activity, are better at approximating that project's *Vulnerability Signal*. We define the Vulnerability Signal as a current pre-image of future security fixes likely to be needed for a project, following from that project's true vulnerability density. Every time a security fix is made to a software project, we learn new information that leads us closer to approximating the true vulnerability density. We reason that metrics "locking on" to a projects vulnerability signal at earlier stages will aid earlier detection.

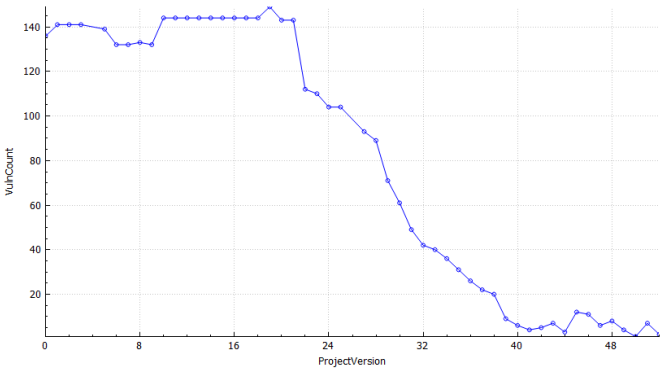

Figure 5. Historical vulnerability density in MySQL DB versions 5.5.x, 0..50

Metrics selected based on linear correlation (i.e., Pearson) or rank order correlation (i.e. Spearman) RawCount, suffer to capture the true residual vulnerability density and corresponding severity of vulnerabilities latent in the released software product. This shortcoming in conventional approaches may be heightened when using automated techniques for model building as we have described in this work. Recall that the vulnerability signal, as we define it, is a property of the overall *true* vulnerability density of a software.

While we can never really know the true vulnerability density, assuming we have the appropriate repositories available, we can better approximate a software's true vulnerability density as security fixes are made over time. That is, every time a released software product is fixed for a security vulnerability, we learn new information about the source code entities involved. Therefore, we view measurement and evaluation of vulnerability prediction metrics better estimated with their relationship to overall severity, and have provided an example of one approach using CVE scoring data to better quantify severity.

5.7 Threats to Validity

The primary threat to the validity of our results for greater generalization to other projects rests with the fact that we used large, mature, and well-known open source projects. It could be argued that vulnerabilities reported such projects are more frequently audited by expert reviewers and that said reviewers naturally focus on modules of greater significance. The result of such expert review and focused attention has the potential to enrich the dataset with additional information that would not translate to small, newly developed projects that have not had similar exposure.

Another caveat with respect to generalization is the validity and applicability of the Previous Vulnerability Assumption. We remind the reader that we arrived at this assumption through careful analysis of the available data only for the subjects we studied. The technique used in this study worked well because of the maturity and stability of the projects used. As used in this study, the Previous Vulnerability Assumption enabled us to simplify the construction of file level vulnerability data. That is, the number of vulnerabilities present in a file at a particular version of the project. For better generalization, rather than assuming a particular vulnerability was present in previous versions of the file, the table building logic would stop application when the affected function was introduced into the file, or affected lines were last modified, rather than the last file modification that wasn't party to a vulnerability fix.

6 Contributions and Future Work

This Section summarizes our research problem: *the residual vulnerability problem*, or the latent persistence of security related defects, termed *vulnerabilities*, remaining in software after its release. We review the problem's significance as well as the significance of our work in this context. We also highlight the creative, original, and novel aspects of our research contribution and findings. We conclude this Section with caveats and suggestions for future research.

6.1 Contributions to Residual Vulnerability Prediction in Software

Our emphasis on more accurately characterizing the residual vulnerability problem and focus on practical automated methods for better quantifying the severity of vulnerability impact, leads directly to several of our significant contributions. A brief description of each follows, with references as appropriate for additional detail.

A new approach for metric selection based on correlation with quantified measures of severity, for use in software vulnerability prediction models implemented with supervised machine learners. Through careful explanation and reasoning about the residual vulnerability problem, we provide a completely new ontology that better *characterizes the relative value of predictions by evaluating their correlation with quantified measures of severity* based on CVE data.

A practical and automated approach to training set construction is presented according to the new approach for metric selection, demonstrating the feasibility of of automating the construction of a training set that is more representative of the characteristic relationships between measurable features and our ontology for learner evaluation. Specific techniques are discussed in Sections 4.3, 4.4, and 4.5.

New insights revealed by our mined data lead to new ways of thinking about vulnerability density over time in a software product. Hence, entirely new concepts and additional explanations for previously observed phenomena emerge on review of mined data (i.e., mined according to our approach for automated training set construction). Specifically, this work discusses the following new concepts:

- **Previous Vulnerability Assumption** introduced as a new concept to aid training set construction, through careful study of CVE vulnerability data available from the NIST National Vulnerability Database for the software projects studied in this work.
- **Vulnerability Signal** introduced as a new concept in 4.4, and empirically demonstrated in Section 5, as the top edge defining a software project's historical vulnerability distribution that appears when visualizing the count of unique CVE entries against affected versions of a software product over time.
- **Fix Signal and Vulnerability Oracle** introduced as a new observed phenomena in sections 4.4.1 and 5 that unify observations from past vulnerability prediction research; specifically the phenomena observed in our study better explain the need for LOC correction and cross-correlation compensation when performing prediction studies.

Empirical data supporting the value of historical evaluation is provided by way of explanation and experiments in Section 5. Primarily, there are more samples on which to train a machine learner because older code has more exposure to use and inspection than newer code. We argue that under certain conditions, this approach better approximates the true population of vulnerabilities in the software, and by extension, provides a better test environment in which to select features in order to build and evaluate vulnerability prediction models.

6.2 Future Work

There are several future directions for this work, however the most fundamental deal with evaluation of ranked results, further validation of the approach, and more clearly establishing the contexts and caveats characterizing it's suitability an applicability.

As presented, our approach is entirely new. The more modern and novel repository-mined change-based frequency metrics, along with architectural module metrics, showed mixed results for class discrimination largely dependent upon the particular version slice of the respective software project in which they were evaluated. Our original premise for the application of repository mined change and architectural metrics was for use in a ranking stage within model pipelines where established classification metrics would discriminate predictions to be ranked. This speaks to the need for additional validation beyond our initial evaluation of Spearman correlation measurements against our own CVE based quantification of severity.

We are excited to further explore the intended application for scoring. To this end, we also note that CVE-based CVSS scores offered a convenient method for quantifying severity, but the ideas presented are just as applicable to better severity quantification techniques.

References

[1] UnderstandC++. Scientific Toolworks, Inc. https://scitools.com/features/, accessed Nov 21, 2016.
[2] UnderstandC++ getting started with the API: Part 2. Scientific Toolworks, Inc. http://scitools.com/blog/api/api-2-entities-references-and-filters, accessed June 1, 2013.
[3] UnderstandC++ Python API manual. Scientific Toolworks, Inc. https://scitools.com/documents/manuals/python/understand.html, accessed June 1, 2016.
[4] Muhammad Anan, Hossein Saiedian, and Jungwoo Ryoo. An architecture-centric software maintainability assessment using information theory. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(1):1–18, January 2009.
[5] Apache. Apache httpd security report for 2.2. Apache Foundation, online. http://httpd.apache.org/security/vulnerabilities_22.html, accessed November 22, 2016.
[6] Varadachari S. Ayanam. Software security vulnerability vs. software coupling: A study with empirical evidence. Master's thesis, Southern Polytechnic State University, December 2009.
[7] Steven M. Bellovin. On the brittleness of software and the infeasibility of security metrics. *IEEE Security and Privacy*, 04(4):96, 2006.
[8] Mehran Bozorgi, Lawrence Saul, Stefan Savage, and Geoffrey M. Voelker. Beyond heuristics: Learning to classify vulnerabilities and predict exploits. In *Proceedings of the Sixteenth ACM Conference on Knowledge Discovery and Data Mining (KDD-2010)*, pages 105–113, 2010.
[9] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.
[10] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, March 2011.

[11] Dan DaCosta, Christopher Dahn, Spiros Mancoridis, and Vassilis Prevelakis. Characterizing the 'security vulnerability likelihood' of software functions. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, Washington, DC, USA, 2003. IEEE Computer Society.

[12] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, October 2005.

[13] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, Washington, DC, USA, May 2009. IEEE Computer Society.

[14] Ahmed E. Hassan and Tao Xie. Software intelligence: the future of mining software engineering data. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering research*, FoSER '10, pages 161–166, New York, NY, USA, 2010. ACM.

[15] Daniel Hein and Hossein Saiedian. Secure software engineering: Learning from the past to address future challenges. *Information Security Journal: A Global Perspective*, 18(1):8–25, 2009.

[16] WA Jansen. NIST IR 7564: Directions in security metrics research, National Institute of Standards and Technology, US Dept. of Commerce, Gaithersburg (2009).

[17] D. Janzen and H. Saiedian. Test-driven development: Concepts, taxonomy, and future direction. *IEEE Computer*, 38(9):43–50, August 2005.

[18] D. Janzen and H. Saiedian. A leveled examination of test-driven development acceptance. In *Proceedings of the 29th ACM International Conference on Software Engineering*, pages 719–722. ACM, May 2007.

[19] D. Janzen and H. Saiedian. Does test-driven development really improve software design quality? *IEEE Software*, 25(2):77–84, March/April 2008.

[20] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan. Detection of software modules with high debug code churn in a very large legacy system. In *Proceedings of the The Seventh International Symposium on Software Reliability Engineering*, ISSRE '96, Washington, DC, USA, 1996. IEEE Computer Society.

[21] BuiltWith Pty Ltd. Apache usage statistics. BuiltWith Pty Ltd., online. https://trends.builtwith.com/Web-Server/Apache, accessed January 9, 2017.

[22] P. K. Manadhata and J. M. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, May 2011.

[23] Peter Mell, Karen Scarfone, and Sasha Romanosky. *CVSS: A Complete Guide to the Common Vulnerability Scoring System Version 2.0*. FIRST: Forum of Incident Response and Security Teams, June 2007.

[24] MITRE. Common vulnerabilities and exposures: About CVE. MITRE Corporation, online. https://cve.mitre.org/about/, accessed November 22, 2016.

[25] David Moore. *Introduction to the Practice of Statistics*. Freeman, New York, 2 edition, 1993.

[26] J. C. Munson and S. G. Elbaum. Code churn: a measure for estimating the impact of code change. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 24–31. IEEE Comput. Soc, 1998.

[27] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5):423–433, May 1992.

[28] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 309–318. IEEE, November 2010.

[29] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 284–292, New York, NY, USA, 2005. ACM.

[30] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 452–461, New York, NY, USA, 2006. ACM.

[31] NIST. NVD: National Vulnerability Database. National Institute of Science and Technology, online. http://nvd.nist.gov/, accessed November 22, 2016.

[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[33] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 9(63):1278–1308, 1975. http://web.mit.edu/Saltzer/www/publications/protection/.

[34] S. Sarkar, G. M. Rama, and A. C. Kak. API-based and information-theoretic metrics for measuring the quality of software modularization. *IEEE Transactions on Software Engineering*, 33(1):14–32, January 2007.

[35] Bruce Schneier. Information security and externalities. In *Social and Economic Factors Shaping the Future of the Internet*, pages 19–20. NSF/OECD, January 2007. http://www.oecd.org/sti/ieconomy/37985707.pdf.

[36] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 18–27, New York, NY, USA, 2006. ACM.

[37] Yonghee Shin. *Investigating Complexity Metrics as Indicators of Software Vulnerability*. PhD thesis, North Carolina State University, Raleigh, North Carolina, 2011.

[38] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, November 2011.

[39] F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz. What we have learned about fighting defects. In *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, pages 249–258. IEEE, 2002.

[40] Kenneth R. van Wyk and John Steven. Essential factors for successful software security awareness training. *IEEE Security and Privacy*, 4(5):80–83, 2006.

[41] Greg Wilson and Jorge Aranda. Empirical software engineering. *American Scientist*, 99(6):466+, November 2011.

[42] Tao Xie, Jian Pei, and Ahmed E. Hassan. Mining software engineering data. In *Software Engineering - Companion, 2007. ICSE 2007 Companion. 29th International Conference on*, pages 172–173, Washington, DC, USA, May 2007. IEEE.

[43] Tao Xie, Suresh Thummalapenta, David Lo, and Chao L. Liu. Data mining for software engineering. *Computer*, 42(8):55–62, August 2009.

[44] A.A. Younis, Y.K. Malaiya, and I. Ray. Using attack surface entry points and reachability analysis to assess the risk of software vulnerability exploitability. In *High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on*, pages 1–8, Jan 2014.

[45] Awad Younis, Yashwant K. Malaiya, and Indrajit Ray. Assessing vulnerability exploitability risk using software properties. *Software Quality Journal*, 24(1):159–202, March 2016.