Predicting Vulnerable Components: Software Metrics vs Text Mining

James Walden
Department of Computer Science
Northern Kentucky University
Highland Heights, KY 41076
Email: waldenj@nku.edu

Jeff Stuckman
Department of Computer Science
University of Maryland
College Park, MD 20742
Email: stuckman@umd.edu

Riccardo Scandariato
iMinds-DistriNet
KU Leuven
3001 Leuven, Belgium
Email: riccardo.scandariato@cs.kuleuven.be

Abstract—**Building secure software is difficult, time-consuming, and expensive. Prediction models that identify vulnerability prone software components can be used to focus security efforts, thus helping to reduce the time and effort required to secure software. Several kinds of vulnerability prediction models have been proposed over the course of the past decade. However, these models were evaluated with differing methodologies and datasets, making it difficult to determine the relative strengths and weaknesses of different modeling techniques.**

In this paper, we provide a high-quality, public dataset, containing 223 vulnerabilities found in three web applications, to help address this issue. We used this dataset to compare vulnerability prediction models based on text mining with models using software metrics as predictors. We found that text mining models had higher recall than software metrics based models for all three applications.

I. Introduction

Building secure software is difficult, time-consuming, and expensive. Prediction models that identify software components that are prone to vulnerabilities can be used to focus limited information assurance resources on a subset of the source code, thereby reducing the time and effort needed to mitigate vulnerabilities.

There is an extensive literature on defect prediction in software engineering. While vulnerabilities are a specific type of software defect, the problem of finding vulnerabilities in software differs in significant ways from the more general problem of finding defects. The most obvious difference is quantitative: there are typically many more defects than vulnerabilities in software, as one can see by comparing the number of defects reported in a project's defect tracker to the number of vulnerabilities listed on a project's security web page.

However, the qualitative differences between vulnerabilities and defects may be more important than the quantitative differences. Different skills are needed to find vulnerabilities than to find defects. Finding vulnerabilities requires an understanding of both the software and the attacker's mindset [1]. Furthermore, defects cause problems for users, who then report them to developers, while vulnerabilities provide opportunities to hackers, who therefore often keep their knowledge of vulnerabilities secret. Opportunities presented by vulnerabilities include not only criminal activities but also the sale of vulnerabilities at increasingly high prices [2]. These effects tend to reduce the ratio of publicly available vulnerabilities to defects.

Developers and vulnerability researchers often limit public disclosure of information about vulnerabilities to reduce opportunities for exploitation. There is much debate in the security community about how much information to disclose about vulnerabilities [3]. Descriptions in vulnerability reports frequently lack the details needed to correctly identify which software components and which versions were impacted by the vulnerabilities [4]. Vulnerability fixing commits are often not identified as such in code repositories. As a result of these attempts to limit exploitation, the quality of vulnerability data is typically lower than the quality of defect data found in software repositories.

Due in part to these differences, the field of vulnerability prediction is not as mature as the study of software defect prediction. While hundreds of defect prediction studies have been published, including multiple systematic literature reviews [5], using a wide range of software metrics and predictive modeling techniques, only a few dozen vulnerability prediction studies have been published. Early defect prediction studies focused on individual models and software artifacts, while later studies began comparing models and then developed standard datasets, such as the PROMISE repository [6], and methods of comparing models.

We expect the field of vulnerability prediction to evolve along a similar trajectory. Many papers have been published on models using a single type of predictor data, while a few papers compare models built using different types of predictors. However, there are no standard data sets to use for comparison of vulnerability prediction models. With this paper, we hope to create some debate and momentum in order to spur the transition towards comparing models with standard datasets. This paper makes two contributions:

1) It presents a comparison of using software metrics as predictors with using text mining techniques to build vulnerability prediction models in the context of web applications written in PHP.
2) It includes a novel, hand-curated dataset of vulnerabilities in PHP web applications that will be offered to the community.

This paper is organized into sections as follows. Section II discusses the collection and validation of the data set, while

Section III describes the methodology used in the experiment. Section IV describes results of the experiment. Section V discusses threats to the validity of this study. Section VI provides related work, and Section VII concludes the paper and outlines directions for future work.

II. Data Set

Over the course of this research, we collected a dataset containing a total of 223 vulnerabilities from multiple versions of three open-source web applications written in PHP. We will make this dataset publicly available with the aim of facilitating various kinds of vulnerability research. With this goal in mind, the dataset contains the following:

- Source code for each studied version for each application.
- Source code metrics for each PHP file at each application version.
- A mapping from vulnerabilities to files for each version, tracking the location of each vulnerability as the application evolves.
- Identifiers for each vulnerability from the vendor or CVE, along with the source code of the commits that introduced and fixed the vulnerability (except for vulnerabilities introduced prior to the first version present in the dataset.)

This section describes the process, quality controls, and results of the data collection for each vulnerability and version.

A. *Selecting applications*

Applications were selected for study based on the following criteria:

- The applications must be free, open-source software, ensuring that no impediments would prevent them from being distributed for research purposes.
- The applications must be web applications written in PHP. We needed applications in a single language so we could compare modeling approaches, and we chose PHP as the most widely used open source web applications are written in that language.
- We selected applications satisfying these criteria with a large number of vulnerabilities found in the application itself (not including vulnerabilities found in plugins), ensuring that a large number of vulnerabilities would be eligible for analysis.

The applications selected were Drupal, Moodle, and PHPMyAdmin. Drupal is a widely used content management system, while Moodle is an open source learning management system. PHPMyAdmin is a web based management tool for the MySQL database.

B. *Obtaining source code*

We obtained the source code of each release, where *releases* are defined as the distribution of a software version to the general public. Major (e.g. v2.0), minor (v2.1), and subminor (v2.1.5) versions were collected for each application, excluding release candidates, patch releases, and security hotfixes. Because the redistributable packages for old versions were not always available, and because redistributable packages may contain ad-hoc modifications such as the addition of copyright/version headers, versions were extracted from public Git repositories.

Versions are localized in Git repositories by mapping each version number with a Git commit identifier. It has been noted [7] that automated means of mapping releases to commits are problematic due to the complex branching structures utilized in repositories. We found that the Git repositories for Drupal and Moodle had tags which accurately mapped release versions to commits. In contrast, some tags were missing or inaccurate in PHPMyAdmin because previous migrations of the repository from CVS and SVN had moved tags to unexpected locations. In these cases, we located commits for each release by examining announced release dates, commit comments, and histories of committed changelog files.

We obtained 95 releases for PHPMyAdmin, 71 releases for Moodle, and 30 releases for Drupal. Releases were collected through the end of 2013. All Moodle public releases are included in the data set, and all releases of the Drupal 6.x series are included. PHPMyAdmin releases before 2.2.0 were not in the Git repository and could not be found. An additional number of PHPMyAdmin releases could not be located because information on the releases was unavailable or the released code was only present on a branch which had since been lost.

Only the files containing server-side PHP code were used for vulnerability prediction. We did not examine any vulnerabilities reported in client-side JavaScript code. In addition, we excluded files belonging to test suites, which were inaccessible to users, and third-party libraries, which were maintained by third parties but packaged into the main software releases.

C. *Mining vulnerabilities*

Next, we mined security vulnerabilities for each application from vulnerability databases. For Moodle, the data source was the National Vulnerability Database (NVD) [8], while for Drupal and PHPMyAdmin, we used the security announcements maintained by those projects. Analysis of PHP source code to locate vulnerabilities was performed by graduate students with experience in PHP web application development and secure programming patterns. Vulnerability locations were verified by the authors.

We chose a random sample of vulnerability advisories for Moodle and PHPMyAdmin. We did not attempt to discover or include any vulnerabilities which were not present in these public databases, which would have prevented a fair random sample from being taken. Due to the small number of vulnerability advisories for Drupal, we used all of the Drupal core vulnerability advisories. Starting with the samples for each project, we attempted to obtain the information necessary to add each advisory to the dataset. We found that some advisories contained multiple vulnerabilities, sometimes of different types. For example, Drupal's 50 advisories corresponded to 97 vulnerabilities.

Using information present on the developer's website or other security-related online resources, we located the Git commit that fixed the vulnerability. To find the versions of Moodle and PHPMyAdmin where those vulnerabilities originated, we traced backwards through revisions until the commit that introduced the vulnerability was located. Obtaining information on exploits and fix commits was straightforward for PHPMyAdmin; in contrast, security advisories for Drupal and Moodle were vague or unavailable to the general public, and some fixes were apparently buried in larger commits with non-security changes.

A commit was deemed to have *introduced* a vulnerability to a particular file if the fixed exploit (or a substantially similar exploit) was impossible before the commit but possible afterward. If a security advisory encompassed multiple instances of the same exploit (such as multiple cross-site scripting vulnerabilities in multiple files), each file with related introductions was counted as a separate exploit to ensure that vulnerabilities could be mapped to files, regardless of the decision to combine similar vulnerabilities into a single advisory. Some vulnerabilities were present in a source file both before and after a complete rewrite, such as two cross-site scripting vulnerabilities affecting parameters having the same purpose. In these cases, the dataset considers the newer vulnerability to be identical to the older one.

D. The collected vulnerability dataset

The resulting vulnerability dataset contains 75 vulnerabilities for PHPMyAdmin, 51 for Moodle, and 97 for Drupal. Table I depicts the classes of vulnerabilities represented in the dataset for each application. Vulnerabilities were manually categorized using the following criteria:

- **Code Injection**: Vulnerabilities allowing attackers to modify arbitrary server-side variables, modify arbitrary HTTP headers, or execute PHP, SQL, or native code on the server
- **CSRF**: Cross-site request forgery vulnerabilities allowing for outside, malicious HTML to induce the user to perform unwanted actions
- **XSS**: Cross-site scripting vulnerabilities allowing for malicious Javascript to be executed in a user's browser
- **Path Disclosure**: Vulnerabilities allowing for the installation path of the application to be maliciously obtained. (This information is sometimes useful when launching a subsequent attack.)
- **Authorization issues**: Confidentiality, integrity, or availability violations not related to another category, including: Privilege bypass vulnerabilities allowing the attacker to subvert application-enforced user permissions or user logins; Information disclosure vulnerabilities allowing for information from the application's database or the server's disk to be read; Vulnerabilities related to missing or inadequately implemented encryption.
- **Other**: Miscellaneous vulnerabilities related to phishing, man-in-the-middle attacks, or unspecified attack vectors.

TABLE I. CLASSES OF VULNERABILITIES IN EACH APPLICATION

	Drupal	Moodle	PHPMyAdmin
Code Injection	2	7	10
CSRF	8	3	1
XSS	32	9	45
Path Disclosure	0	2	12
Authorization issues	39	28	6
Other	16	2	1

We found that different applications had different distributions of vulnerability categories, even though (in the case of PHPMyAdmin and Moodle) the vulnerabilities covered approximately the same time frame. Cross-site-scripting vulnerabilities dominated in PHPMyAdmin, while authorization vulnerabilities dominated in Moodle. This reflects the fact that Moodle was designed as a multi-user application from the beginning, possibly leading to more awareness of attacks such as cross-site scripting but more opportunities to implement authorization mechanisms improperly. Drupal, being a multi-user application, also had a concentration of authorization vulnerabilities.

Although different applications in our dataset had different dominant categories of vulnerabilities, the same major categories were represented in all of them. Many of these categories, such as cross-site scripting, would not appear in datasets of non-web vulnerabilities or non-security defects. This emphasizes the importance of collecting and studying datasets covering multiple platforms and fault types, because these categories of defects would otherwise not be represented in prediction research.

Lifetimes of vulnerabilities: Figure 1 shows the number of days between the introduction of vulnerabilities (the first versions when they were introduced) and their fixes being committed. Many vulnerabilities remained in the code for an extended period of time, with the median lifetime of a vulnerability being 871 days and one vulnerability not being fixed until 3993 days after it was introduced.

Accuracy of vulnerable versions data: Vulnerability advisories often contain information on the version ranges of programs that are vulnerable. The final versions in these ranges (representing the most recent vulnerable version) are usually accurate; however, the first versions in these ranges (representing the version where the vulnerability first appeared) are often incorrect. To study this phenomenon, we manually identified the first vulnerable release for each PHPMyAdmin and Moodle vulnerability. We compared these version numbers with those listed in sources of vulnerability advisory data.

As shown in Table II, the vulnerable-versions data in the security advisories was mostly incorrect[1]. In the cases marked as *inspection earlier*, some of the earliest vulnerable version numbers were missing from the advisory. Most likely, the

[1] Whenever a vulnerability might have first appeared in an old release with unavailable source code, that vulnerability was omitted from this table.

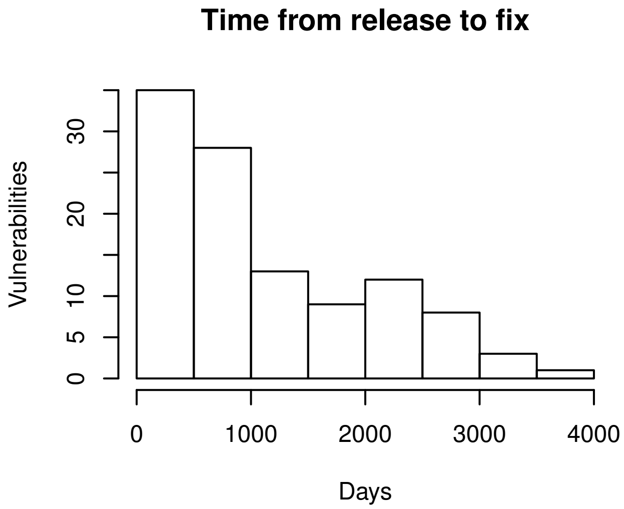

Fig. 1. Histogram depicting the number of days between the first release of a vulnerability and the vulnerability being fixed in the version control repository

TABLE II. COMPARING EARLIEST VULNERABLE VERSIONS IN VULNERABILITY ADVISORIES WITH THOSE FOUND BY CODE INSPECTION

	PMA (Vendor)	PMA (NVD)	Moodle (NVD)
Database earlier	26	29	18
Inspection earlier	28	25	20
Versions match	14	14	13

earlier versions were never checked for the vulnerability's presence because there was little benefit to doing so – users who are concerned about security will probably not be using old versions anyway. In other cases, marked as *database earlier*, the advisory marked early versions as being vulnerable when they actually were not. This usually happened when the vendors asserted that all old versions of a product were potentially vulnerable – in reality, very few vulnerabilities affected all previous versions of a given product.

Although these discrepancies would probably not affect the typical consumers of security advisories, some vulnerability research requires more accurate version information. Data from sources such as the NVD [9] are unsuitable for such research. For this reason, all results in this paper are based on vulnerable-versions data that was manually reconstructed from Git.

E. Reasons for excluding vulnerabilities from the dataset

Not all vendor or NVD security advisories yielded vulnerabilities that could be included in this dataset. Some advisories were discarded for the following reasons:

- 4 PHPMyAdmin and 4 Moodle advisories were excluded because we were unable to determine the location or nature of the vulnerability.
- 2 PHPMyAdmin and 2 Moodle advisories were excluded because there was no direct way to exploit the weakness hypothesized by the developers.
- 4 PHPMyAdmin and 1 Moodle advisories were excluded because they represented a pervasive issue affecting the entire program; for example, every page being susceptible to a cross-site framing attack.
- 1 PHPMyAdmin and 8 Moodle advisories were excluded because they are caused by third-party products, such as the platform that hosts the application or the libraries included with the application.

In addition, some vulnerabilities were mined from security advisories but disregarded in this paper's analyses:

- 5 PHPMyAdmin, 2 Moodle, and 3 Drupal vulnerabilities were excluded because they affected files that did not contain PHP code, such as Javascript files.
- 1 PHPMyAdmin and 3 Moodle vulnerabilities were excluded because localizing the vulnerability was problematic, such as when different files were affected by the vulnerability's introduction and fix. For example, one such Moodle vulnerability was introduced when a non-functional access control setting was added to a control panel but fixed when the authorization step was actually implemented elsewhere in the program.
- 3 Moodle vulnerabilities were excluded because they resulted from code clones of vulnerabilities already included in the dataset.

F. Localizing vulnerabilities to files and versions

Aside from locating the commits where vulnerabilities were introduced and fixed, training vulnerability predictors requires vulnerabilities to be *localized*, or associated with a set of (filename, version) tuples which are susceptible to the particular vulnerability. In isolation, the commit that introduced the vulnerability does not provide sufficient information to localize the vulnerability for several reasons:

- Vulnerable code is not necessarily introduced into a released version as soon as it is committed. Hence, the commit date cannot be used to estimate the first release that is susceptible.
- The revision graphs in Git seemed useful for determining when development branches containing vulnerable code were released; however, we found that branches and merges were not always explicitly recorded over long-lived version histories. In some cases, this was due to the past use of CVS or SVN repositories where branches were represented differently than in Git. For example, in Moodle, many Git commits resulted from merges whose parents are not recorded. These missing merges prevent individual commits (most notably, some which introduce vulnerabilities) from automatically being traced to the versions they affected.
- Vulnerable code sometimes moved between files over the vulnerability's lifetime, due to mass renames of

files or the vulnerability being preserved across refactorings.

We localized vulnerabilities by searching for *indicator strings* which were present in vulnerability commits[2]. To create an indicator string for a vulnerability, we identified a string present in the modified code of a vulnerability's commit (or a nearby commit) which persisted until the next release of the product, revealing the first release where a vulnerability actually affected the codebase. The following process is followed to localize vulnerabilities in each susceptible version:

1) **Identify the main branch**: Developers sometimes produce maintenance releases of older versions when a new, major version has recently been released. These branches are excluded from localization so a linear progression of the vulnerability can be tracked over time. Maintenance branches were identified by examining release dates and the sizes of inter-version differences, which can help find the point of greatest similarity where new, major revisions diverge from maintenance branches.
2) **Locate the first version where the indicator string appears**: Files in each version were searched to locate the earlier point where the indicator appeared. If the indicator was never found (which happens when vulnerabilities are refactored prior to release), a new candidate indicator string was chosen and the search repeated.
3) **Track renamed files across releases until the vulnerability is fixed**: Indicator strings cannot be used to localize vulnerabilities over their entire lifetimes because lines of code containing vulnerabilities are frequently modified without being fixed. (The use of detection scripts [10] could compensate for this, but they would be time-consuming to develop.) After the introduction of a vulnerability, the vulnerability was first assumed to remain in the same file across all future versions until the date when it is fixed. File renames are detected by running `git diff` on each version and using the results of its rename detection heuristic.
4) **Double-check locations of vulnerability fixes**: The previous step assumes that vulnerabilities remain in the same places throughout their lifetimes. This assumption is then checked by comparing the apparent location of the vulnerability at the time when it is fixed with the name of the file actually modified by the fix commit. If these names are not identical, or if the rename detection heuristic in the previous step lost track of the vulnerable file, the vulnerability was localized manually. This was accomplished by identifying additional indicator strings which allow the entire version history to be covered.

G. Ensuring quality of the dataset

Due to the large amount of manual effort involved in collecting the vulnerability and version data, several steps were taken to check the integrity of the data. Any anomalies detected during a check were investigated and resolved.

- Vulnerabilities were only excluded if they met the criteria described in Section II-E
- Because we sometimes had to manually disambiguate release branches from development branches, the source code of each version was checked for suspicious patterns of changes, which would indicate that the wrong branch was chosen. These suspicious changes include cases where the same file repeatedly appears and disappears from one version to the next.
- To ensure that the vulnerability commits and indicator strings were chosen properly, the proper order in which these artifacts appear was enforced. Vulnerable code must be committed before it is released, and vulnerabilities must be released before they are fixed.
- As described previously, the apparent location of a vulnerability in the version immediately preceding its fix was verified against the files affected by the fix commit.
- Information found in vulnerability databases was independently verified during data collection, comparing the behavior described in the vulnerability report with the actual source code.

III. Methodology

In order to compare the two prediction techniques, we used a fairly standard experimental setup (i.e., cross-validation) that has been used in many previous works, such as [11]–[14].

We considered one version for each of the three applications that we analyzed in this paper: Drupal 6.0, PHPMyAdmin 3.3, and Moodle 2.0. For each version, we retained files containing PHP source code, while discarding dependencies to third party software like external libraries. The source files were labeled as either "vulnerable" or "clean". The vulnerability status of a file was the dependent variable in this study. Independent variables for source code metrics based models included such metrics as as fan-in and fan-out and lines of code, while independent variables for text mining models were the term frequencies.

We performed a cross-validation experiment twice on each application version. In the first experiment, we used software metrics as predictors, while in the second experiment, we used term frequencies. We compared the quality of the predictions in both cases using several performance indicators, which are described below.

A. Dependent variable

As described in Section II, we mined vulnerability reports from vulnerability databases and localized each vulnerability to the file where it resided during the version that we analyzed. These mappings from vulnerabilities to files were used to define the dependent variable for a binary prediction (or classification) problem. Files where at least one vulnerability was present were labeled vulnerable, while files where no vulnerabilities were present were labeled clean.

[2] This method was not applied to the Drupal vulnerabilities, which were localized with a manual process.

Because our dataset spans multiple major versions of PHPMyAdmin and Moodle but only one particular version of each was tested for vulnerability prediction, not all vulnerabilities for those two applications were included in this study. Some vulnerabilities were introduced after the analyzed version was released, while others were fixed before the release of the analyzed version. For this reason, only 31 out of the 75 vulnerabilities in PHPMyAdmin and 25 out of the 51 vulnerabilities in Moodle were mapped to files. Furthermore, because some files contained multiple vulnerabilities, the number of vulnerable files in Table IV is smaller than the number of vulnerabilities.

B. Independent variables

1) Software metrics: We found that there were no tools available that could compute a range of size, complexity, and coupling metrics on PHP source code. This posed difficulties for vulnerability prediction that are not encountered when analyzing Java or C++ programs, as there are commercial tools that compute such metrics for those languages. For this reason, we developed a metrics computation tool which is based on the PHP compiler front-end developed by Vries and Gilbert [15].

One problem encountered when computing metrics for scripting languages, such as PHP, is resolving the targets of function and method invocations. Unlike in Java and C, imports and dependencies can be dynamically redefined at runtime by selectively including particular PHP files as part of the program's logic. This makes it difficult to unambiguously determine the file and function that a symbol references, which is an essential task when computing coupling metrics. We address this by simply assuming that all method and function calls potentially reference any method or function in the same program which has the same name. Although this potentially introduces errors (or, effectively, noise) into the coupling metrics, these metrics are computed consistently throughout the training and test phases of the experiment. This consistent treatment ensures that any fitted prediction models remain valid, regardless of the noise that was present in the computation of one set of features.

The following metrics were computed and are included in the dataset:

- *Lines of code*: Number of lines in a source file where PHP tokens occur, excluding lines without PHP tokens, such as blank lines and comments. When tokens span multiple lines, such as in the case of strings with embedded newlines, only one line is counted in the total.
- *Lines of code (non-HTML)*: Same as *Lines of code*, except HTML content embedded in PHP files (content outside of `php` start/end tags) is not considered.
- *Number of functions*: Number of function and method definitions in a file.
- *Cyclomatic complexity*: The size of a control flow graph after linear chains of nodes are collapsed into one. Computed by adding one to the number of loop and decision statements in the file.
- *Maximum nesting complexity*: The maximum depth to which loops and control structures in the file are nested.
- *Halstead's volume*: A volume estimate ($(N_1 + N_2) \log n_1 + n_2$) using the number of unique operators (n_1) and operands (n_2) and the number of total operators (N_1) and operands (N_2) in the file. For the purposes of this metric, *operators* are method names and PHP language operators, while *operands* are parameter and variable names.
- *Total external calls*: The number of instances where a statement in the file being measured invokes a function or method defined in a different file.
- *Fan-in*: The number of files (excluding the file being measured) which contain statements that invoke a function or method defined in the file being measured.
- *Fan-out*: The number of files (excluding the file being measured) containing functions or methods invoked by statements in the file being measured.
- *Internal functions or methods called*: The number of functions or methods defined in the file being measured which are called at least once by a statement in the same file.
- *External functions or methods called*: The number of functions or methods defined in other files which are called at least once by a statement in the file being measured. (When the target of a function or method call is uncertain, all possible call targets are considered to have been called.)
- *External calls to functions or methods*: The number of files (excluding the file being measured) calling a particular function or method defined in the file being measured, summed across all functions and methods in the file being measured.

2) Text mining: The text mining process begins by tokenizing each PHP source file with PHP's built-in `token_get_all` function. Tokens represent language keywords, punctuation, and other code constructs. A short tokenizer program calls this function to tokenize each source file and output a list of tokens and their associated frequencies. This representation is known as a "bag of words." The set of unique terms or tokens is the vocabulary of the developers.

The tokenizer processes the set of tokens to eliminate unnecessary features. It ignores comments and whitespace. String and numeric literals are converted into fixed tokens, e.g. `T_STRING` is used to represent a string literal instead of the contents of the string.

In the "bag of words" model, each source file is represented as a term vector. Term vectors typically have a high dimensionality (up to 18K terms), depending on the size of the application and the richness of the vocabulary used by the developers. These term vectors are then used as the predictors in our models.

C. Cross-validation and performance indicators

We use stratified cross-validation to evaluate model performance, which is a standard technique. The files of an

TABLE III. TERMINOLOGY

Measure	*Definition*
TP	**True positives:** number of files that are predicted as vulnerable and do contain vulnerabilities.
TN	**True negatives:** number of files that are predicted as clean and that do not contain vulnerabilities.
FP	**False positives:** number of files that are predicted as vulnerable but do not contain vulnerabilities.
FN	**False negatives:** number of files that are predicted as clean but do contain vulnerabilities.

application version are randomly divided into three folds of equal size. Each fold has the same percentage of vulnerable files as the entire version (stratification), which is important to preserve a realistic testing condition for the prediction model. We use only three folds (instead of five, or ten) due to the low absolute number of vulnerable components in some applications. For instance, Moodle contains only 24 vulnerable files. It would not make much sense to create, for instance, ten folds with only a couple of vulnerable files in each.

Iteratively, each fold is retained as the testing set. That is, a prediction model is built starting from the samples in the other two folds (training set), and the model is used to predict the class of the files in the testing set.

Based on our experience with vulnerability prediction in previous studies, we use *Random Forest* as our primary machine learning algorithm. In particular, we use the implementation provided by Weka 3.7, with the size of the forest set to 100 trees. All other parameters are set to their default values.

At the end of the experiment, each file has been predicted as either vulnerable or clean. As we know which files contain the vulnerabilities in our dataset, we can compare the predictions to the locations of these vulnerabilities. Accordingly, we judge each prediction to be either correct (true positive or true negative) or erroneous (false positive or false negative).

With reference to the terminology introduced in Table III, we compute performance indicators for each cross-validation experiment. In security, the most important indicator is *recall* ($\mathcal{R}$), which can assume values between 0 and 1. A higher recall means that the model has correctly identified a larger number of vulnerable files. The indicator is defined as follows:

$$\mathcal{R} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

The second key indicator we use to compare the techniques is the file *inspection* ratio ($\mathcal{I}$), which is the percentage of files that one has to consider (e.g., inspect manually) to make it possible to find the true positives identified by the model. This indicator can assume values between 0 and 1 and is defined as follows:

$$\mathcal{I} = \frac{\text{TP} + \text{FP}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

In summary, recall represents the benefit that one gets when following the advice of the prediction model, while inspection is the cost associated with that benefit. For completeness, we also report other indicators like precision, false positive rate, and accuracy. However, these indicators are not used to draw conclusions in the comparison.

Finally, note that each cross-validation is repeated 10 times with a different, randomly chosen partitioning into folds of the dataset.

D. Undersampling

For some of the applications we analyze, the positive rate, i.e., the percentage of vulnerable files, is very low. For instance, the positive rate of Moodle is under 1%. This means that there are overwhelmingly more negatives (clean files) than positives (vulnerable files) in the training set during each iteration of the cross-validation. Training a model from such an imbalanced dataset is often challenging. Therefore, sampling is a technique that is often used to balance the training set, e.g., see Shin et al. [12]. With undersampling, all the positive cases in the training set are retained, while only a subset of the negatives is selected. The sample of negatives is randomly chosen such that the number of positives matches the number of negatives. The result is a perfectly balanced training set, which is used to build the prediction model. Note that the testing fold is never altered, in order to preserve the correct testing conditions. We employed undersampling in all experiments using the implementation provided by Weka, the SpreadSubsample unsupervised filter.

E. Hypotheses

We compare two key performance indicators ($\mathcal{R}$ and $\mathcal{I}$) obtained by the technique based on software metrics (SM) with those of the technique based on text mining (TM). As mentioned above, cross validation is repeated 10 times for each application. The goal of our comparison is to determine whether, on average, the SM technique performs better or worse than the TM technique.

Hence, our null hypotheses are as follows:

$$H_0^{\mathcal{R}} : \mu\{\mathcal{R}_{\text{SM}}\} = \mu\{\mathcal{R}_{\text{TM}}\}$$

$$H_0^{\mathcal{I}} : \mu\{\mathcal{I}_{\text{SM}}\} = \mu\{\mathcal{I}_{\text{TM}}\}$$

To assess whether there is a statistically significant location shift in the performance indicators of the two techniques, we use the Wilcoxon rank-sum test, which is a non-parametric test for independent samples. We use a significance level of 0.05.

F. Replicating the study

We provide a companion website containing our public vulnerability dataset and other material necessary to replicate this study[3]. To support replication, for each application version, we provide the source code of the version, the class (vulnerable or clean) of each file, the metrics extracted from each file, the grammar used to tokenize the files, and the terms extracted from each file.

TABLE IV. DESCRIPTIVE STATISTICS FOR THE APPLICATIONS. MOODLE IS THE LARGEST APPLICATION AND HAS THE LOWEST POSITIVE RATE. DRUPAL IS AT THE OPPOSITE END OF THE SPECTRUM.

	Vulnerable files	*Total files*	*P-rate (%)*	*Text features*
Drupal	62	202	**30.68**	3886
PHPMyAdmin	27	322	**8.39**	5232
Moodle	24	2942	**0.82**	18306

TABLE V. RESULTS OF CROSS-VALIDATION. MEAN (μ) AND STANDARD DEVIATION (σ) FOR THE KEY PERFORMANCE INDICATORS. TEXT MINING PERFORMS BETTER THAN SOFTWARE METRICS.

	Indicator		*SW metrics (%)*	*Text mining (%)*
Drupal	Recall	μ	**76.9**	**80.5**
		σ	2.8	3.3
	Inspection	μ	**45.5**	**43.3**
		σ	2.3	2.3
PHPMyAdmin	Recall	μ	**66.3**	**73.7**
		σ	12.0	9.8
	Inspection	μ	**42.0**	**43.4**
		σ	2.7	4.6
Moodle	Recall	μ	**70.4**	**80.0**
		σ	10.8	6.1
	Inspection	μ	**32.1**	**28.7**
		σ	4.1	3.7

IV. RESULTS

Table IV provides summary information about the three applications, including number of files, number of vulnerable files, percentage of vulnerable files (P-rate), and the total number of text features. This information is helpful in interpreting the results of the experiment. The three applications are heterogeneous, providing a rich set of testing conditions for the prediction models. Drupal and PHPMyAdmin are similar in terms of size; however, both of these applications are much smaller than Moodle. The larger size of Moodle also translated into a much larger number of tokens (i.e., text features) used in the code base. Moodle has the smallest positive rate (P-rate). As the positive rate gets smaller, the task of predicting which files are likely to contain vulnerabilities is more challenging, as vulnerable files are rare. Hence, better prediction performance in case of a smaller positive rate is more valuable due to the larger number of files that a human reviewer would need to examine. For the sake of comparison, the positive rate of Moodle is similar to the rates reported for projects like Mozilla Firefox [12], Google Chrome or Microsoft Windows, which have been used in the past to test vulnerability prediction techniques. On the other hand, Drupal has an elevated positive rate, and this type of application has never before been used to test vulnerability prediction models.

Table V reports the results obtained from the cross-validation experiment described in Section III-C. This table directly compares the two techniques (software metrics and text mining) over the two key performance indicators (recall and inspection). Mean values (μ) were obtained by averaging the performance of the prediction models over ten executions of the cross-validation experiment. Standard deviation (σ) is also reported for both indicators.

On average, the prediction technique based on text mining performed better. The recall was higher in the case of Drupal (+3.6 percentage points), PHPMyAdmin (+7.4) and Moodle (+9.6). This difference was statistically significant for Moodle and Drupal. The file inspection ratio was approximately the same in the two cases, with text mining performing slightly better in the case of Drupal (-2.2 percentage points) and Moodle (-3.4), and metrics performing slightly better in the case of PHPMyAdmin (+1.4). We can conclude that text mining provided more benefits (i.e., better recall values, up to about 10 points) with comparable costs (i.e., same inspection rate) overall. The performance of text mining also improved significantly in the more challenging cases, i.e., when the P-rate is smaller, as with PHPMyAdmin and, particularly, Moodle.

TABLE VI. ADDITIONAL PERFORMANCE INDICATORS FOR CROSS-VALIDATION.

	Indicator	*SW metrics (%)*	*Text mining (%)*
Drupal	Precision	52.0	57.1
	FP rate	31.6	26.9
	Accuracy	71.0	75.4
PHPMyAdmin	Precision	13.2	14.3
	FP rate	39.8	40.6
	Accuracy	60.7	60.6
Moodle	Precision	1.8	2.3
	FP rate	31.7	28.3
	Accuracy	68.3	71.8

For the sake of completeness, Table VI shows additional performance indicators which are often reported in related work. We found that text mining had better precision (up to 5 percent points), and the difference was statistically significant in the case of Drupal and Moodle. However, we did not observe major differences between the two types of predictors in the other performance indicators.

A. Cross-project prediction

We also tested the potential for using a model built on one application to predict vulnerable components in other applications. This case is particularly interesting, as it could present the possibility of building a single prediction model that could be used for multiple applications. With this in mind, we independently trained three prediction models based on all files of each application[4]. Iteratively, each model was tested by predicting the labels of the files in the other two applications, for a total of six prediction sets. We repeated the experiment using both software metrics and text features as predictors.

Cross-project prediction performance in these experiments was generally poor. The only case where the performance indicators assumed acceptable values was a model based on software metrics, which was trained on Drupal and tested on Moodle, which had a recall of 70% and an inspection ratio of 36%. The same model, however, was not effective when tested on PHPMyAdmin (recall of 66% and inspection ratio of 49%). In general, models using software metrics performed slightly better in the cross-project case.

Low cross-project performance has also been observed in other contexts when using within-project models for cross-project prediction, and specialized training techniques have

[3] http://seam.cs.umd.edu/webvuldata

[4] We used Random Forest and undersampling as before

been proposed [16] to work around this. In our case, it may be that cross-project prediction was hindered by the unequal distribution of vulnerability categories between applications. As seen in Section II-D, the distribution of vulnerabilities by category differed strongly between applications. In future work, we plan to examine the impact of vulnerability categories on predictive models.

V. Threats to Validity

We discuss threats to construct, conclusion, internal, and external validity.

Construct validity. Some vulnerabilities had to be excluded from the data set, because we were unable to determine the location or nature of the vulnerability or because different files were affected during the lifetime of the vulnerability. We depended on the heuristics used by the Git version control system to track renames of files across versions.

The dynamic nature of dependencies in PHP led us to a method of computing coupling metrics that differs from the methods of computing such metrics in static languages. However, this technique was applied consistently across all applications in this study.

Conclusion validity. Threats to conclusion validity relate to issues that affect the validity of statistical inferences. We used standard techniques for our statistics and modeling, and we used well recognized tools for these purposes, including Weka and R.

Internal validity. We attempted to avoid selection bias by either using all of the vulnerabilities in an application or by using a randomly selected subset of vulnerabilities. However, the applications were not selected randomly, but were chosen by finding applications with high numbers of vulnerability advisories for their core components. We did not attempt to control the categories of the vulnerabilities that we sampled, possibly introducing heterogeneity that would impact the prediction results.

When labeling files as *vulnerable* or *non-vulnerable*, files containing vulnerabilities not present in our dataset (either because they are still-undiscovered or because they were not sampled) would have been labeled as non-vulnerable. This inevitably introduces inaccuracies into the performance indicators, although using the inspection ratio performance indicator (instead of precision) partially mitigates this.

External validity. Our results might be specific to the set of web applications that we selected. While we selected applications from different domains, all three applications were open source. Further studies with a broader set of web applications, including both commercial and open source applications, would be needed to generalize the results to the entire class of PHP web applications. Similar reservations apply to generalizing this study to web applications written in other languages or to other types of software, such as desktop or mobile applications.

VI. Related Work

The presence of undiscovered vulnerabilities represents a serious risk to users of software, as an adversary could discover a vulnerability and exploit it widely while users are still unprotected. Machine learning techniques can be used to develop models identifying portions of source code (such as files or classes) which are more likely to contain previously unknown vulnerabilities, with the aim of increasing the likelihood that these vulnerabilities will be discovered during development or testing. One practical application of such a model was developed by Microsoft [17] to screen commits which had an elevated risk of introducing defects.

A. Vulnerability prediction

The field of *vulnerability prediction* uses machine learning to estimate the likelihood of security vulnerabilities, which represent a subset of software defects in general. Shin et al. [12] compared several source code and process metrics for their ability to discriminate between vulnerable and non-vulnerable code components. These metrics were also evaluated in the context of a notional *code inspection* task, measuring the degree that using the metrics would reduce the amount of code that must be inspected to cover a certain proportion of vulnerabilities. Going beyond traditional metrics, other studies utilized features such as the developers' organizational structure [18], text mining [19], static analysis alerts [20], and shared inter-component dependencies [11].

Past work in vulnerability prediction has primarily focused on C and Java applications (including desktop and mobile [21] variants of Java). Many vulnerabilities have been found in web applications written in languages such as PHP or Ruby. It is an open question whether the prediction techniques above are similarly effective for such applications, especially as many types of vulnerabilities found in web applications are unique to the web environment. Shar and Tan [22], [23] predicted vulnerabilities with tailored features related to dataflow and sanitization, and Smith and Williams [24] targeted SQL-related code for the same purpose. Walden et al [25] examined correlations between code metrics and the vulnerability density of a variety of PHP web applications.

B. Defect prediction

Vulnerability prediction can be viewed as a specialization of defect prediction, where much attention has been placed on proposing new features for prediction and refining the performance of existing predictors. Because of the similarities between defect and vulnerability prediction models [26], general defect prediction is also of interest for those who predict vulnerabilities. Many features that were used for vulnerability prediction are similar to those used for general defect prediction, such as metrics [16], [27]–[29] and inter-module or inter-developer relationships [30], [31]. In addition, the predictive power of uncommon features such as file popularity [32] and coding standard adherence [33] has been explored. Related work has also compared a wide variety of machine learning techniques (or models), which can generally be chosen independently from the features. For example, Menzies et al. examined how predictive performance is impacted by the choice of machine learning model and evaluation criteria [29], emphasizing that experiments must be carefully designed with an appropriate model and performance criteria for predictors to function effectively.

Relatively few studies have compared the performance of dissimilar features (such as comparisons between metric and non-metric approaches). The large volume of past defect prediction research makes the need for synthesis and comparison apparent, in order to help practitioners and researchers leverage past studies and choose models and features which have been proven to work. Mizuno and Hata [34] compared metrics and text features (which are also used for vulnerability prediction in this work), and Rahman et al. [35] compared static analysis with statistical prediction for finding bugs. A more systematic effort by D'Ambros et al. [27] compared code metrics, change metrics, and previous bugs, releasing a public dataset which enables others to test their own features and techniques. In this work, we compare two types of feature for vulnerability prediction, and we release a curated vulnerability dataset, serving as a resource for future researchers in a similar manner.

C. Vulnerability datasets

Reliable replication of defect prediction studies has been identified as a beneficial but elusive goal [36]. Without such replication, achieving a comparative understanding of vulnerability prediction techniques would be unattainable, as no single study could examine every conceivable machine learning model and feature on its own. Although publishing a copy of the underlying data used in a study is not *sufficient* to guarantee reproducibility [36], it is certainly *necessary*. Aside from facilitating replication, public defect and vulnerability data sources can accelerate the development of new techniques by eliminating the need to collect new data for each study.

The PROMISE data repository [6] offers several datasets for defect prediction research, including machine learning features and defect counts. Similar datasets have been released as part of individual defect prediction studies [27], [37]. However, a similar repository for vulnerability prediction research does not yet exist. Some curated collections of vulnerability and/or exploit data have been released to support specialized tasks such as IDS evaluation [38], buffer overflow detection benchmarking [39], or dynamic exploit detection and prevention [40]. Other data sources [41], [42], including the National Vulnerability Database [8], compile information on vulnerabilities; however, these data sources lack structured information such as names of vulnerable files, making them inadequate for vulnerability prediction research in their current form. In this study, we release curated datasets including vulnerabilities, metrics, and source code of three open-source PHP web applications, allowing for future research leveraging this data to evaluate new predictors beyond those examined as part of this study.

VII. Conclusion

In this paper, we presented a dataset of open source web applications, including software metrics, source code, and vulnerability locations, that can be used for developing and testing vulnerability prediction models. This dataset contains 223 vulnerabilities found in three web applications written in PHP: Drupal, Moodle, and PHPMyAdmin. Because available PHP analysis tools computed a limited set of metrics, we developed a custom tool to compute a wider variety of metrics, which are included in the dataset. This dataset complements other publicly available defect prediction datasets, providing coverage for a domain (security vulnerabilities) and a context (web applications) that are not well explored.

We used this dataset to compare the vulnerability prediction effectiveness of two modeling techniques: one based on software metrics and the other based on text mining using a "bag of words" model for the source code. Both models were built using a Random Forest machine learning technique, undersampling to compensate for the small percentage of vulnerable files in most applications.

We summarize the results of our study below:

> **Predictive value of text features**: We evaluated the models using stratified cross-validation, finding that text mining provided significantly better recall performance with almost the same cost, as measured by the file inspection ratio. Recall and inspection ratio values can be found in Table V.
>
> **Cross-project vulnerability prediction**: We found that vulnerability prediction models developed on one project did not effectively predict vulnerable files in other projects. Only a single model had reasonable results for predicting vulnerable components in another application. A model built using Drupal was able to predict vulnerable components in Moodle, with a recall of 70% and an inspection ratio of 36%.
>
> **Web application data mining and prediction**: We demonstrated that vulnerability predictors, which have largely been used on server and desktop applications in the past, can successfully be used to develop vulnerability prediction models for web applications.

In the future, we plan to study the ability of vulnerability prediction models to predict the locations of new vulnerabilities in future versions of the same project. We also plan to examine the effect of vulnerability categorization on prediction and apply machine learning techniques which can improve cross-project performance. Finally, we intend to examine additional predictors beyond source code metrics and text mining features. We invite others to replicate our results and compare them with their own predictors by leveraging the data and scripts that we published at the same time as this paper.

Acknowledgment

We thank the United States Office of Naval Research for its partial support for this research under contract N000141210147.

References

[1] G. McGraw and B. Potter, "Software security testing," *IEEE Security & Privacy*, vol. 2, no. 5, pp. 81–85, 2004.

[2] S. Frei, "The known unknowns: Empirical analysis of publicly unknown vulnerabilities," Dec 2013. [Online]. Available: https://www.nsslabs.com/reports/known-unknowns-0

[3] A. Arora and R. Telang, "Economics of software vulnerability disclosure," *IEEE security & privacy*, vol. 3, no. 1, pp. 20–25, 2005.

[4] F. Massacci and V. H. Nguyen, "Which is the right source for vulnerability studies?: an empirical analysis on Mozilla Firefox," in *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, 2010, p. 4.

[5] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, “A systematic literature review on fault prediction performance in software engineering,” *Software Engineering, IEEE Transactions on*, vol. 38, no. 6, pp. 1276–1304, 2012.

[6] T. Menzies, B. Caglayan, Z. He, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan. (2012, June) The promise repository of empirical software engineering data. [Online]. Available: http://promisedata.googlecode.com

[7] J. F. Shobe, M. Y. Karim, M. B. Zanjani, and H. Kagdi, “On mapping releases to commits in open source systems,” in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 68–71.

[8] NIST, “National vulnerability database,” http://nvd.nist.gov/, 2013.

[9] V. H. Nguyen and F. Massacci, “The (un)reliability of NVD vulnerable versions data: An empirical experiment on google chrome vulnerabilities,” in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ser. ASIA CCS ’13. New York, NY, USA: ACM, 2013, pp. 493–498.

[10] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates, “When a patch goes bad: Exploring the properties of vulnerability-contributing commits,” in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, ser. ESEM ’13, 2013, p. to appear.

[11] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, “Predicting vulnerable software components,” in *Proceedings of the 14th ACM conference on Computer and communications security*, ser. CCS ’07. New York, NY, USA: ACM, 2007, pp. 529–540. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315311

[12] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, “Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities,” *Software Engineering, IEEE Transactions on*, vol. 37, no. 6, pp. 772–787, 2011.

[13] I. Chowdhury and M. Zulkernine, “Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities,” *Journal of Systems Architecture*, vol. 57, no. 3, pp. 294–313, 2011.

[14] V. H. Nguyen and L. M. Sang Tran, “Predicting vulnerable software components with dependency graphs,” in *International Workshop on Security Measurements and Metrics (MetriSec)*, 2010.

[15] E. de Vries and J. Gilbert, “Design and implementation of a PHP compiler front-end,” Technical report, Trinity College Dublin, Ireland, Tech. Rep., 2007.

[16] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, “On the relative value of cross-company and within-company data for defect prediction,” *Empirical Software Engineering*, vol. 14, no. 5, pp. 540–578, 2009.

[17] A. Tarvo, N. Nagappan, and T. Zimmermann, “Predicting risk of pre-release code changes with checkinmentor,” in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, 2013, pp. 128–137.

[18] T. Zimmermann, N. Nagappan, and L. Williams, “Searching for a needle in a haystack: Predicting security vulnerabilities for Windows Vista,” in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. IEEE, 2010, pp. 421–428.

[19] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden, “Software vulnerability prediction using text analysis techniques,” in *Proceedings of the 4th international workshop on Security measurements and metrics*, 2012.

[20] M. Gegick, L. Williams, J. Osborne, and M. Vouk, “Prioritizing software security fortification through code-level metrics,” in *Proceedings of QoP ’08: 4th ACM workshop on Quality of protection*. New York, NY, USA: ACM, 2008, pp. 31–38.

[21] R. Scandariato and J. Walden, “Predicting vulnerable classes in an android application,” in *Proceedings of the 4th international workshop on Security measurements and metrics*, 2012.

[22] L. K. Shar, H. B. K. Tan, and L. C. Briand, “Mining sql injection and cross site scripting vulnerabilities using hybrid program analysis,” in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 642–651.

[23] L. K. Shar and H. B. K. Tan, “Predicting common web application vulnerabilities from input validation and sanitization code patterns,” in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*. IEEE, 2012, pp. 310–313.

[24] B. Smith and L. Williams, “Using sql hotspots in a prioritization heuristic for detecting all types of web application vulnerabilities,” in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*. IEEE, 2011, pp. 220–229.

[25] J. Walden, M. Doyle, G. A. Welch, and M. Whelan, “Security of open source web applications,” in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 545–553. [Online]. Available: http://dx.doi.org/10.1109/ESEM.2009.5314215

[26] Y. Shin and L. Williams, “Can traditional fault prediction models be used for vulnerability prediction?” *Empirical Software Engineering*, vol. 18, no. 1, pp. 25–59, 2013.

[27] M. DAmbros, M. Lanza, and R. Robbes, “Evaluating defect prediction approaches: a benchmark and an extensive comparison,” *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 531–577, 2012.

[28] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, “Defect prediction from static code features: current results, limitations, new approaches,” *Automated Software Engineering*, vol. 17, no. 4, pp. 375–407, 2010.

[29] T. Menzies, J. Greenwald, and A. Frank, “Data mining static code attributes to learn defect predictors,” *Software Engineering, IEEE Transactions on*, vol. 33, no. 1, pp. 2–13, 2007.

[30] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu, “Putting it all together: Using socio-technical networks to predict failures,” in *Software Reliability Engineering, 2009. ISSRE’09. 20th International Symposium on*. IEEE, 2009, pp. 109–119.

[31] T. Zimmermann and N. Nagappan, “Predicting defects using network analysis on dependency graphs,” in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 531–540.

[32] A. Bacchelli, M. DAmbros, and M. Lanza, “Are popular classes more defect prone?” in *Fundamental Approaches to Software Engineering*. Springer, 2010, pp. 59–73.

[33] C. Boogerd and L. Moonen, “Evaluating the relation between coding standard violations and faultswithin and across software versions,” in *Mining Software Repositories, 2009. MSR’09. 6th IEEE International Working Conference on*. IEEE, 2009, pp. 41–50.

[34] O. Mizuno and H. Hata, “An integrated approach to detect fault-prone modules using complexity and text feature metrics,” in *Advances in Computer Science and Information Technology*. Springer, 2010, pp. 457–468.

[35] F. Rahman, S. Khatri, E. T. Barr, and P. Devanbu, “Comparing static bug finders and statistical prediction,” in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 424–434.

[36] T. Mende, “Replication of defect prediction studies: problems, pitfalls and recommendations,” in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. ACM, 2010, p. 5.

[37] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting defects for Eclipse,” in *Proceedings of PROMISE’07: ICSE Workshop on Predictor Models in Software Engineering*, May 2007, p. 9.

[38] K. Kendall, “A database of computer attacks for the evaluation of intrusion detection systems,” Ph.D. dissertation, Massachusetts Institute of Technology, 1999.

[39] K. Ku, T. E. Hart, M. Chechik, and D. Lie, “A buffer overflow benchmark for software model checkers,” in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ser. ASE ’07. New York, NY, USA: ACM, 2007, pp. 389–392. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321691

[40] G. Nilson, K. Wills, J. Stuckman, and J. Purtilo, “Bugbox: A vulnerability corpus for PHP web applications,” in *Presented as part of the 6th Workshop on Cyber Security Experimentation and Test*. USENIX, 2013.

[41] “OSVDB: The open source vulnerability database,” http://www.osvdb.org/, 2013.

[42] Offensive Security, “Exploit DB,” http://www.exploit-db.com/.